Title A novel parallel algorithm for surface editing and its fpga implementation

Name Yukun Liu

A NOVEL PARALLEL ALGORITHM FOR SURFACE EDITING AND ITS FPGA IMPLEMENTATION

YUKUN LIU

Ph.D.

September 2013

UNIVERSITY OF BEDFORDSHIRE

A NOVEL PARALLEL ALGORITHM FOR SURFACE EDITING AND ITS FPGA IMPLEMENTATION

by

YUKUN LIU

Ph.D.
Institute for Research in Applicable Computing

A thesis submitted to the University of Bedfordshire in partial fulfilment of the requirements for the degree of Doctor of Philosophy

September 2013

ABSTRACT

Surface modelling and editing is one of important subjects in computer graphics. Decades of research in computer graphics has been carried out on both low-level, hardware-related algorithms and high-level, abstract software. Success of computer graphics has been seen in many application areas, such as multimedia, visualisation, virtual reality and the Internet. However, the hardware realisation of OpenGL architecture based on FPGA (field programmable gate array) is beyond the scope of most of computer graphics researches. It is an uncultivated research area where the OpenGL pipeline, from hardware through the whole embedded system (ES) up to applications, is implemented in an FPGA chip.

This research proposes a hybrid approach to investigating both software and hardware methods. It aims at bridging the gap between methods of software and hardware, and enhancing the overall performance for computer graphics. It consists of four parts, the construction of an FPGA-based ES, Mesa-OpenGL implementation for FPGA-based ESs, parallel processing, and a novel algorithm for surface modelling and editing.

The FPGA-based ES is built up. In addition to the Nios II soft processor and DDR SDRAM memory, it consists of the LCD display device, frame buffers, video pipeline, and algorithm-specified module to support the graphics processing.

Since there is no implementation of OpenGL ES available for FPGA-based ESs, a specific OpenGL implementation based on Mesa is carried out. Because of the limited FPGA resources, the implementation adopts the fixed-point arithmetic, which can offer faster computing and lower storage than the floating point arithmetic, and the accuracy satisfying the needs of 3D rendering. Moreover, the implementation includes Bézier-spline curve and surface algorithms to support surface modelling and editing.

The pipelined parallelism and co-processors are used to accelerate graphics processing in this research. These two parallelism methods extend the traditional computation parallelism in fine-grained parallel tasks in the FPGA-base ESs.

The novel algorithm for surface modelling and editing, called Progressive and Mixing Algorithm (PAMA), is proposed and implemented on FPGA-based ES's. Compared with two main surface editing methods, subdivision and deformation, the PAMA can eliminate the large storage requirement and computing cost of intermediated processes. With four independent shape parameters, the PAMA can be used to model and edit freely the shape of an open or closed surface that keeps globally the zero-order geometric continuity. The PAMA can be applied independently not only FPGA-based ESs but also other platforms.

With the parallel processing, small size, and low costs of computing, storage and power, the FPGA-based ES provides an effective hybrid solution to surface modelling and editing.

DECLARATION

I declare that this thesis is my own unaided work. It is being submitted for the degree of Doctor of Philosophy at the University of Bedfordshire.

It has not been submitted before for any degree or examination in any other university.

Name of candidate:

Signature:

Date:

DEDICATION

To my parents

To my husband, Jie Geng

To my son, Hao Geng

To all who gave me a hand to reach this achievement, I will always say:

Thank you.

ACKNOWLEDGEMENTS

Issac Newton said, “If you say that I see far, it is because I am standing on the shoulders of giants”.

This PhD research would have been a huge, unachievable piece of work if there had not been the facilities available in the surroundings, which come from different companies and organisations, and, of course, without the Internet. Thanks for all the open or commercial resources. With them, the research has been accomplished at the foundation built from the facilities and sources. Most of them have been quoted as references at the end of the thesis. It is just like “standing on the shoulders of giants”.

It is a trial-and-error process. It is a dark tunnel always with a definite light in front of me, far or near. It is a lonely journey full of joys and sadness. It is my PhD journey. It is just a scientific exploration, I think. Thanks for all the help from different places and my deep heart to support me.

I would like to express my thanks to my team of supervisors, Prof. Yong Yue, Prof. Carsten Maple, and Prof. Jiaomin Liu, for their encouragement, criticism, guidance, suggestions, and support during the research.

I would like to thank Mrs. Carol Connett and her family for their help and good time along with them during my stay in the UK.

I would like to thank Qixiang Wang for his suggestions and advice in my research.

I would like to thank my colleagues in my working university, Hebei University of Science and Technology, Prof. Liwei Guo, Chunru Huang, Guojiang Gao, and Shiquan Qiao, for their help in my work during my PhD research.

Thanks to Prof. M. James C. Crabbe, Dr. Dayou Li, Dr. Jon Hitchcock, Dr. Herbert Daly, Dr. Xiaohua Feng, Shijuan Pan, Hui Wei, Shuang Gu, Beisheng Liu, Tao Cao, and others, for providing help when I needed.

Finally, my very special thanks are due to my beloved family, parents, husband and son, whose encouragement has been invaluable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D	Two-Dimensional
2D DCT	Two-Dimensional Discrete Cosine Transform
3D	Three-Dimensional
ANSI	American National Standards Institute
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BSP	Board Support Package
C^0	Zero-order Parametric Continuity
C^1	First-order Parametric Continuity
C^2	Second-order Parametric Continuity
CAD	Computer-Aided Design
CAGD	Computer-Aided Geometric Design
CFI	Common Flash Interface
CMOS	Complementary Metal-Oxide-Semiconductor
DDR SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
DMA	Direct Memory Access
DMIPS	Dhrystone Million Instructions Per Second
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
E^3	Three-dimensional Euclidean Space
EDS	Embedded Design Suite
ES	Embedded System
ESDK	Embedded System Development Kit
FFT	Fast Fourier Transform
FIFO	First-in First-out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
G^0	Zero-order Geometric Continuity
G^1	First-order Geometric Continuity
G^2	Second-order Geometric Continuity
G^r	r-order Geometric Continuity
GCC	GNU Compiler Collection
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface

HAL	Hardware Abstraction Level
HDL	Hardware Description Language
HSMC	(Multimedia) High Speed Mezzanine Card
I-GL	Implementation of OpenGL
I/O	Input/Output
IDE	Integrated Design Editor
IP	Intellectual Property
IRQ	Interrupt Request
JTAG	Joint Test Action Group
LAB	Logic Array Block
LCD	Liquid Crystal Display
LE	Logic Element
LUT	Lookup Table
MIMD	Multiple Instruction Multiple Data
MMU	Memory Management Unit
MPU	Memory Protection Unit
NCO	Numerically Controlled Oscillator
PAMA	Progressive and Mixing Algorithm
PCB	Printed Circuit Board
PE	Processing Element
PIO	Programmed Input/Output
PLL	Phase-Locked Loop
R^d	d-Dimensional Riemannian Space
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SBT	Software Build Tools
SGDMA	Scatter-Gather Direct Memory Access
SIMD	Single Instruction Multiple Data
SOPC	System on a Programmable Chip
SPMD	Single-Program Multiply-Data
UART	Universal Asynchronous Receiver/Transmitter
VBU1	Varying Beta-u-one
VBV1	Varying Beta-v-one
VBU2	Varying Beta-u-two
VBV2	Varying Beta-v-two
VGBU1	Varying Group Beta-u-one
VGBV1	Varying Group Beta-v-one

VGBU2	Varying Group Beta-u-two
VGBV2	Varying Group Beta-v-two
VGP	Varying Group Position
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
VMB1	Varying Mixing Beta-one
VMB2	Varying Mixing Beta-two
VP	Varying Position

LIST OF NOTITIONS

b_0 , …, b_i ,…, b_n	Control points of a Bézier-spline curve
$[b_0, b_1, b_2, ..., b_n]$	Control polygon of a Bézier-spline curve of degree n
$b_k^r(u)$	Intermediate de Casteljau points at the r step
$b_{k,l}$	Control points of a Bézier surface of degree (m,n)
$\breve{b}_k$	Coefficients of a v-isoparametric curve of the Bézier surface *x(u,v)*
$\hat{b}_l$	Coefficients of a u-isoparametric curve of the Bézier surface *x(u,v)*
b_{ij}^{rq} , $b_{i,j}^{r,q}$	Intermediate de Casteljau points at the (r,q) step of computation of a point on a bi-cubic Bézier surface
$B_k^n(u)$	Bernstein polynomials of degree n
$\beta_1(u)$, $\beta_2(u)$	Shape parameters of a Beta-spline curve
$\beta_{u1}(u,v)$, $\beta_{u2}(u,v)$	Shape parameters along the u direction of a Beta-like-spline surface
$\beta_{v1}(u,v)$, $\beta_{v2}(u,v)$	Shape parameters along the v direction of a Beta-like-spline surface
$\Delta^{1,0}b_{k,l} = b_{k+1,l} - b_{k,l}$	Differential operation performed on the first subscript k of control vertices
$\Delta^{0,1}b_{k,l} = b_{k,l+1} - b_{k,l}$	Differential operation performed on the second subscript l of control vertices
$\Delta^{1,1}b_{k,l} = b_{k+1,l+1} - b_{k,l+1} - b_{k+1,l} + b_{k,l}$	Twist vectors
$\gamma(i)$	Ratio parameters of lengths of two line segments, *[V(i),W(3i+1)]* and *[W(3i+1),W(3i+2)]*, of a Beta-spline curve
$\gamma_u(i, j)$	Ratio parameters along the u direction of lengths of two line segments, *[V(i,j),W(3i+1,3j)]* and *[W(3i+1,3j),W(3i+2,3j)]*, of v-isoparametric Bézier curves of a Beta-like-spline surface
$\gamma_v(i, j)$	Ratio parameters along the v direction of lengths of two line segments, *[V(i,j),W(3i,3j+1)]* and *[W(3i,3j+1),W(3i,3j+2)]*, of u-isoparametric Bézier curves of a Beta-like-spline surface
S(w), T(u), x(u)	Parametric curves
S(w,t), T(u,v)	Parametric surfaces
$u \mapsto u(w,t)$, $v \mapsto v(w,t)$	Reparametrisations
V(i)	Original control points of a curve

V(i,j)	Original control points of a surface
W(k)	Interpolated control points of cubic Bézier curve segments with Beta constraints
W(k,l)	Interpolated control points of bi-cubic Bézier patches with the PAMA
[W(i),W(j)]	Line segment between two control points of *W(i)* and *W(j)* of a curve
[W(i,j),W(k,l)]	Line segment between two control points of *W(i,j)* and *W(k,l)* of a surface
[W(3i),W(3i+1), W(3i+2),W(3(i+1))]	Control polygon of a cubic Bézier curve segment
[W(3i,3j), W(3i,3(j+1)), W(3(i+1),3(j+1)), W(3(i+1),3j)]	Control net of a bi-cubic Bézier patch
$W_{k,i}, \overline{W}_{j,l}, \tilde{W}_{k,i}, \hat{W}_{j,l}$	Intermediate points of control polygons of isoparametric Bézier curves
$\overline{W}(3i,3j)$	Intermediate points with $1:\beta_{u1}(i,j)$, the ratio of lengths of two line segments, *[W(3(i-1)+2,3j)* $\overline{W}(3i,3j)$ *]*, and *[*$\overline{W}(3i,3j)$*,W(3i+1,3j)]*
$\tilde{W}(3i,3j)$	Intermediate points with $1:\beta_{v1}(i,j)$, the ratio of lengths of two line segments, *[W(3i,3(j-1)+2),* $\tilde{W}(3i,3j)$ *]*, and *[*$\tilde{W}(3i,3j)$*,W(3i,3j+1)]*
x(u,v)	A Bézier surface of degree (m,n)
$x_{i,j}(u,v)$	Bézier patches
$x_g(u)$	A generating curve of a tense-product Bézier patch

Chapter 1 Introduction

1.1 Background

The research in this project is a comprehensive process involving software, hardware and parallelism. These issues have to be discussed individually.

1.1.1 Software and Hardware in Computer Graphics

In computer graphics, methods of software and hardware have been successfully implemented and applied to a number of areas. The applications can be found in many important fields, such as the Internet, multimedia, visualisation, and virtual reality. Many application areas have benefited from computer graphics, for examples, automotive and computer industries, science and engineering, education, entertainment, information visualisation, pharmaceutical research, 3D (three-dimensional) medical imaging, and earth and weather modelling (Bliss 1980, Krone et al 2009, Losh 2006, Martinez and Chalmers 2004, Nahum 1996, Schröder et al 2012, and Yoo et al 2012).

Computer graphics covers a wide range of research subjects. They include animation, colour, display algorithms, geometric algorithms, lighting, interaction techniques, morphing, object modelling, picture and image generation, rendering technologies, representations and editing of curve, surface, solid and object, shading, shadowing, texture, 3D graphics and realism, virtual or augmented reality, and others (House 1996, Pan et al 2000, and Sathyanarayana and Kumar 2008).

In the surface modelling and editing, studies are also plentiful (Barsky and DeRose 1990, Cheng and Goshtasby 1989, Duchamp and Stuetzle 2003, Farin 1993, Hearn et al 2011, Hoppe et al 1994, Huang et al 2006, Joe 1990, Lawrence and Funkhuser 2003, Loop and DeRose 1989, Loop 1994, Müller et al 2010, Nasri and Abbas 2002, Perez et al 2003, Sederberg et al 2003, Sorkine et al 2004, Wang et al 2006, Welch and Witkin 1994, and Yu et al 2004). For 3D rendering, it is one of purposes to make the geometrical images as smooth as possible. There are three mainstream schemes to maintain the local control during modelling and editing. They are subdivision, deformation and shape parameterisation with a specific-order geometric continuity (shorten as shape

parameterisation in the rest of the thesis). According to the number of researches and application results that have been published, the first two have achieved more success than the third one.

Since this research aims at a hybrid solution of software and hardware to computer graphics, resources of the hardware system to support graphics applications have to be considered. As algorithms for surface modelling and editing are applications on the top of the system architecture, their storage and computing costs can increase the requirements of hardware resources. Compared to the first two methods, shape parameterisation methods have lower storage and computing costs because intermediated processes required by the first two methods are eliminated. Shape parameterisation methods can be used to construct a composite curve or surface with geometric continuities that can yield the smoothness in the geometric sense, rather than with the agreement of parameter derivatives of parameter continuities. Parameter continuities have more restrictive conditions than geometric continuities do. This research proposes an algorithm, Progressive and Mixing Algorithm (PAMA), which belongs to the third one. It is detailed in Chapter 7.

1.1.2 Hardware Solution in Computer Graphics

Compared with existent studies on software in computer graphics (Barsky and DeRose 1990, Botsch and Sorkine 2008, Cheng and Goshtasby 1989, Duchamp and Stuetzle 2003, Hoppe et al 1994, Huang et al 2006, Lawrence and Funkhuser 2003, Loop and DeRose 1989, Loop 1994, Müller et al 2010, Nasri and Abbas 2002, Perez et al 2003, Sederberg et al 2003, Sorkine et al 2004, Wang et al 2006, Welch and Witkin 1994, and Yu et al 2004), there are fewer hardware accomplishments published in this field. Related implementations were initiated mainly by Silicon Graphics Inc. (Kilgard 1997, Kilgard and Akeley 2008, Luebke and Humphreys 2007, and Patashev et al 2000). Some publications are related to simulation and performance evaluation for general-purpose computer systems and only a few of them are for graphics applications. Others are implemented with GPUs (Graphics Processing Unit) and also used in general-purpose computer surroundings (Nickolls and Dally 2010, Owens et al 2008, and Vuduc and Czechowski 2011).

The hardware accomplishment for abstract algorithms of the graphical pipeline requires the support from the computer system to which the hardware implementation is integrated. If an entire graphical system is built up from the hardware, individual parts of the graphical pipeline system can be easily compatible with each other and it also means that a lot of construction work is necessary. But when a single hardware solution to an abstract algorithm is embedded into an existent system, the compatibility with the original system environment requires much attention. It requires a large amount of time and effort to

bridge the new part with the original system. Or, the new part can be incompatible with the original hardware. This issue can reduce the attraction to hardware when researchers look for solutions to graphics algorithms and make a choice between hardware and software (Chua and Neumann 2000, Guthe et al 2005, Kazakov 2007, Li et al 2006, and Weyrich et al 2007).

In the evolution history of OpenGL, it can be seen that there are always some hardware-dependent and some hardware-independent sections involved in the OpenGL pipeline. The OpenGL pipeline consists of two joined-together sub-pipeline, geometric pipeline (or vertex pipeline) and fragment pipeline (pixel pipeline) (Hearn et al 2011, and True et al 2004). The OpenGL is also a state machine with a fixed topology and orthogonal state variables. From the input of OpenGL API (application programming interface), an object rendering has to be transmitted with a set of state variables and carried out through the OpenGL pipeline. The OpenGL state machine performs all the processing in a fixed order that follows the geometric and fragment pipeline, and finally displays the object on a device screen.

Embedded systems (ES) are defined with a computing core and are intended for in-field applications rather than general-purpose computing. ESs cover a major fraction of the digital systems market, and act as a key technology in the automotive, consumer electronics, industrial automation, military and aerospace applications, office automation, and telecommunication and data-communication industries (Green and Edwards 2000, Henzinger and Sifakis 2007, Konrad et al 2004, and Zave 1982). When computers become more and more popular in daily life, ESs dominate areas of controlling communication, transportation, and medical systems (Henzinger and Sifakis 2007).

Compared with general-purpose computers, except for computation requirements considered in general-purpose computers, control requirements are more significant for ESs.

As FPGAs (field programmable gate array) provide a platform to configure hardware systems, FPGA-based ESs allow designers to build up their system not only with software blocks but also via hardware modules. They give designers more freedom and opportunity for options.

Naturally, the upstream part of OpenGL pipeline that is close to the graphics applications at the top of a system tends to a software solution. The downstream part that approaches frame buffers is prone to acquire the hardware support. Therefore, in this project, a hybrid method combining hardware-related procedures and software-related algorithms is adopted to implement the pipeline in an overall and progressive way. Because of the flexibility of the programmable hardware and system design combining with software and hardware, an FPGA-based ES is chosen as the research platform for

this project.

1.1.3 Parallelism in Computer Graphics

As known, a program solving a typical large problem is usually composed of parallelisable and non-parallelisable parts. For this kind of problem, Amdahl's law (Amdahl 1967) can be roughly described as that the non-parallelisable part of a program will limit the overall speed-up via parallelisation of the program. It can also be further described as that the maximum speedup with parallelisation of a sequential program is reciprocal of the fraction of running time the program spends on its non-parallelisable parts.

For the same type of problem, Gustafson's law (Gustafson 1988) tells that for the overall speedup, optimising the large running-time portion of a program will have a greater effect than making effort on speeding up the much smaller part of the program.

Parallelisation of algorithms, however, relies on their data dependencies as well. In most algorithms, there are dependent and independent calculations in parallel. There are one or more critical paths that are the longest chain of dependent calculations in algorithms. The study of Bernstein 1966 offers the conditions that two program fragments can be executed in parallel.

There are different types of parallelism, including bit-level, instruction-level, data, and task parallelisms (Grama et al 2003). Since it adopts a hybrid method of software and hardware based on FPGA, this project makes the best use of task parallelism. If viewed from a system standpoint, as applications are executed on FPGA-based ESs, co-processors play a key role (Cevik 2004, Cheng and Goshtasby 1989, and Sridharan and Priya 2004). From an algorithm implementation's point of view, because of its graphics pipeline goal, the parallel attribute is deconstructed naturally into the pipeline accomplishment of FPGA.

1.2 Research Motivation

OpenGL architectures with FPGA-based hardware realisations are not within the scope of most computer graphics researchers (Constantinides and Nicolici 2011, Kilgard 1997, Kilgard and Akeley 2008, Luebke and Humphreys 2007, and Patashev et al 2000). The FPGA hardware has not attracted the same passion and effort in graphics researches as software in the past decades. It is still an area waiting for exploation where the OpenGL pipeline, from the hardware system through the whole ES up to applications, is implemented in an FPGA chip. Thus, this project presents a hybrid solution to graphics pipeline with software and hardware based on an FPGA ES in order to enhance the overall system performance in computer graphics applications.

On the other hand, a lot of research results in computer graphics have been applied to other fields, such as automobile industry, archaeology, biomedicine, chemistry, earth mantle convection analysis, and videogames and virtual reality (Bliss 1980, Krone et al 2009, Losh 2006, Martinez and Chalmers 2004, Nahum 1996, Schröder et al 2012, and Yoo et al 2012). Thus, with the features of small size, low power cost, and parallelism processing, this research can be promoted in many in-field applications, such as robots, measuring fields, crafts designs, and medical surroundings.

1.3 Research Aims and Objectives

This research aims to develop a novel parallel algorithm for effective surface modelling and editing, which is incorporated with hardware support by an FPGA implementation in parallel. Following comprehensive literature reviews and in-depth analysis of existing algorithms, a novel algorithm has been developed for surface modelling and editing in 3D computer graphics. The enhancement and parallelism of the algorithm with FPGA support have also been done in this work.

The objectives of the research have been fulfilled as follows,

- The realisation of the computer graphics pipeline has been divided effectively and efficiently between hardware and software to enhance the overall performance of the computer graphics system in an integrated way. The hybrid solution has been carried out with an FPGA-based ES.
- During the development and implementation of FPGA-based ES with a hybrid approach, the parallelism has been considered and adopted in the task processing. Co-processors and pipelines have been used to further enhance the entire performance of the FPGA-based ES.
- An implementation of Mesa-based OpenGL has been completed for the FPGA-based ES. In addition to specifications of the OpenGL ES standard, this implementation consists of the Bézier-spline curve and surface algorithms that support the surface modelling and editing. To meet requirements of limited storage and logic elements of the FPGA chip, this implementation adopts the fixed point arithmetic, which provides a satisfactory accuracy for the 3D rendering. The fixed point arithmetic is composed of designated processes for multiplication, division, dot production, cross production, square root, linear interpolation, and trigonometric functions.
- The new algorithm for surface modelling and editing, PAMA, has been developed and enhanced for effective and flexible applications not only in the general-purpose computer environment but also in the FPGA-based ES environment. In the

FPGA-based ES, the PAMA is more practical because of its lower requirements on storage and computation than two other methods, subdivisions and deformations. The results of its hybrid implementation verify that the PAMA can be applied in the shape change in an interactive way.

- For a rigorous mathematic derivation based on the differential geometry, parameter continuities and geometric continuities have been investigated. The geometric properties of Bézier-spline curves and surfaces have been also inspected. Above these, continuities of the PAMA have been explored and summarised mathematically.
- Research results have been collected and systematically analysed via the comparison between two environments of the FPGA-based ES and general-purpose computer.

1.4 Research Methodology

This project proposes a hybrid approach to investigating both software and hardware methods. This approach can bridge the gap between methods of software and hardware, and thus enhance the overall performance for computer graphics.

Since it investigates both software and hardware to stay up-to-date, a comprehensive research method is adopted in this work. This research method includes four aspects.

- **To merge different technologies into a technology programme**. As a new research usually crosses two or more disciplines, it is necessary to apply different technologies of different groups, such as individuals, companies, and academic institutes, to a comprehensive and complex project. To make these technologies work effectively and efficiently for a specific complicated goal, it is critical to know each of them and bridge related parts with their interfaces and correct techniques. These techniques must be based on principles of each of relevant disciplines and also bridge the gap between them.
- **To attempt different means and theories until a solution is obtained**. Because of uncertainties of a new research, a trial-and-error method is required. As regards a hybrid solution of hardware and software, an adjustment on one part can lead to changes of many related parts. It is normal to do many times of adjustment or revision during attempts and explorations of different methods. Before the correct result comes out, it can make one keep re-searching and re-verifying. During this process, it is based on the rational analysis and rich experiences to make a decision about which direction one should move in at the next step.
- **To shift timely from an obsolete technology to an up-to-date one**. As

technologies develop quickly, old technologies are always replaced with new ones. When a technology is not available for some reason, such as obsolete, a substitute one has to be found. This alternative can not only make a project proceed but also provide a space for the project to develop. This is also based on rich knowledge and experiences that help one to gain the ability of making a right decision about the shifting direction.

- **To flexibly use the existent knowledge and concepts**. Any existent technology model can be broken to yield a new one in order to meet new needs of the real society. The existing knowledge and concepts can give one a method of thinking about old models. One must produce new ideas and concepts for new requirements during the research process.

1.5 Thesis Organisation

This research includes four parts: the construction of the FPGA-based ES, Mesa-OpenGL implementation for the FPGA-based ES, parallelism processing, and a novel algorithm for surface modelling and editing in computer graphics. From a system perspective from the top to the bottom, the implementation of the algorithm for surface modelling and editing is an application of computer graphics at the high level and has induced a series of research activities, all the way through the OpenGL implementation, ES construction, and integration of the parallelism context down to the hardware system setup with FPGA.

The system structure of this project can be shown in Figure 1.1. In a system view, this structure is composed of five main parts from the top of application to the bottom of hardware.

- On the top, the algorithm for surface modelling and editing is the application using Mesa-OpenGL and auxiliary functions to edit surfaces through user interactions and displaying the resulting images on the display device.
- In the middle, the Mesa-OpenGL implementation carries out functions of the OpenGL pipeline.
- The connection between Mesa-OpenGL and hardware abstract layer is the auxiliary interface of the OpenGL to communicate with the FPGA-based ES.
- The FPGA-based ES is a hybrid set of software and hardware resources that comprises the hardware abstract layer and the configured FPGA supported by Altera Embedded System development board of Cyclone III (Altera 2008a).
- The parallelism is applied to the speedup of the graphics and video processing of the system. It includes pipelines and co-processors.

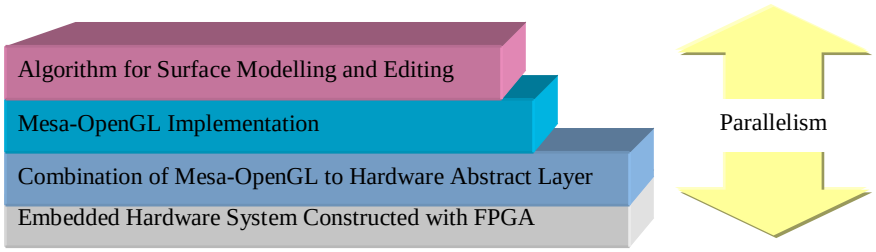


Figure 1.1 Structure of the Project

The thesis is organised as follows.

- After the introduction of Chapter 1, literature reviews are presented in Chapter 2. Chapter 2 encompasses the contents of ES, FPGA, parallelism, and computer graphics.
- An FPGA-based ES is built up. In addition to Nios II soft processor and DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory) memory, the ES consists of the LCD (liquid crystal display) display device, frame buffers, video pipeline, and algorithm-specified module to support the graphics processing. These are discussed in Chapters 3 and 4. Chapter 3 explores an integrated hybrid ES. Chapter 4 focuses on the FPGA-based embedded hardware system for graphics applications.
- An OpenGL implementation based on Mesa is carried out for the FPGA-based ES. It has been ported successfully to the FPGA-based ES. These are detailed in Chapter 5.
- The parallelism implementation in the FPGA-based ES is explored in Chapter 6.
- A novel algorithm for surface modelling and editing, PAMA, is proposed and implemented. It is detailed in Chapter 7. Applications of the PAMA in a general-purpose computer environment are presented in Chapter 7.
- Results of surface modelling and editing with PAMA on the FPGA-based ES are presented in Chapter 8. Comparisons between applications on two environments of the general-purpose computer and FPGA-based ES are also discussed in Chapter 8.
- The future work is discussed in Chapter 9.
- Conclusions are presented in Chapter 10.
- Continuities of the PAMA and the mathematical context are explored rigorously in Appendix.

Chapter 2 Literature Reviews

Since this research aims at bridging the gap between software and hardware solutions for computer graphics with a hybrid approach, it is a study crossing two disciplines of computer science and electronics engineering. The principles in both disciplines have to be followed and a scheme integrating the individual features of them into one application goal must be created in order to find a feasible and effective solution. Researches on both disciplines have been done in this project. According to four parts of this project, as shown in Figure 1.1 and Section 1.5, these being the construction of a novel FPGA-based ES, a new implementation of OpenGL based on Mesa for the FPGA-based ES, parallel processing, and a new algorithm of surface modelling and editing – PAMA, the literature reviews on them will be discussed in different sections and in the order of the thesis organisation and system architecture from the bottom (the hardware system) up to the top (applications).

2.1 Related Studies of ESs

ESs, as a younger discipline than computer science, have been flourished in many application fields, such as life science, nano-engineering, controlling communication, transportation, and medical systems (Chen et al 2008, Ghasemzadeh et al 2013, Gorski et al 2010, Liu et al 2012, Massey et al 2007, Saddem et al 2011, and Surducan et al 2010).

Since ESs are usually designed directly for special consumers or specific application environments, the factors of cost, performance and power consumption have more effect on them than general-purpose computers. Each unit of an ES has its cost and benefit, and contributes to the entire cost and benefit of the system. To lower the cost and power consumption and enhance the system performance, it is necessary to encourage a detailed consideration of implementation options for each unit of the system. For example, a floating-point algebraic computing system can be implemented with a software or hardware unit. The software implementation can have a lower price but a slower processing speed. The designated hardware unit can speed up but at a higher price. Thus, an ES design must make a choice by balancing between costs and system performance.

The ES design takes the system as a whole (Green and Edwards 2000). It is necessary to evaluate implementation options between software and hardware in the context of the overall system, rather than according to whether or not a single unit implementation is optimum.

On the other hand, for developers of high-level applications in a computer system, compared with hardware facilities, available software resources give more support to the application implementations. For example, software resources have plentiful specified libraries and user-friendly APIs of operating systems. Thus, to find answers to new problems, developers more likely turn to the software to search algorithm implementations for answers, rather than to the fixed obscure hardware. In the computer system, the software is more open and easier to expand than the hardware.

Behind this phenomenon, it is the computer science background of developers that leads them to software. The computer science holds a general framework for a system design that is highly abstract and logical in order to make the design device-independent, user-oriented and computation-targeted. The design and construction of ESs, however, are device-dependent, application-oriented and operation-targeted. This is the difference between the computer science and electronics engineering. ESs combine the computer science and electronics engineering by using both the computing ability of computer science and the engineering capability of electronics engineering. The former leads to mathematics, logic and science while the latter faces construction, engineering and technologies. Researchers in these two fields think in different ways. Researchers with the computer science background think in the way of the discrete mathematics of computer science while ones with the electronics engineering background have to sample and process the signals that continuously change (Henzinger and Sifakis 2007). In fact, it is just like the difference of digital and analogue signals. The former can be one and zero while the latter can be a range of voltage values.

These require that ESs, as an emerging discipline, increase and improve their knowledge framework for common designs and constructions to meet the diverse and varied application needs. ESs have been setting up a bond between computer science and electronics engineering for both technologies and applications in a definitely interdisciplinary way. In addition, ESs provide a solution of hardware/software co-design of a system for applications. This is unachievable for general-purpose computers.

In this project, since the fixed hardware environment of general-purpose computers cannot meet the needs of combining hardware and software to solve the graphics pipeline problems, ESs are chosen as the platform. Especially, with FPGA technologies, ESs become more feasible and flexible. ESs will be explored deeply in Chapter 3 and the FPGA detailed in Chapter 4.

2.2 Investigation of Hardware Graphics Applications

As a branch of computer applications, the graphics applications are also facing the need of speeding up with hardware solutions. Because of the spread of the games market, the acceleration of processing has become more crucial in graphics applications than many other applications. Compared with other technologies in the computer graphics, GPUs (Graphics Processing Unit) can be one of the most attractive devices.

2.2.1 GPU Applications

GPUs have been gradually merging into the graphics design to speed up the processing and also playing a promising role in scientific computing applications (Baladron et al 2012, Choe et al 2013, Nickolls and Dally 2010, Owens et al 2008, and Vuduc and Czechowski 2011). Over the past several years, many researches have attained rich results in games, biophysics, and neuroscience.

In games, GPUs save the host CPU from complex graphics tasks and improve the performance of the overall system. With the parallelism, large throughput, and high computational ability, GPUs have an effective pipelining mechanism for graphics pipeline. The synergy of GPU architecture is obtained by an array of computing units in the fine-grained and closely coupled parallel. The upstream of the computer system gives the GPU the input of a list of vertices of 3D coordinate system. These vertices are grouped in geometric primitives. For example, a triangle primitive can be composed of three vertices with three coordinate data for each vertex. Through several steps, including vertex operations, primitive assembly, rasterisation, fragment operations, and composition, these geometric primitives are shaded and mapped onto a display screen. GPUs bring games into an unprecedented speed-up. It is a proof that the hardware is more effective at the speed-up than the software. In the design and development of a target application, hardware can be more flexible and direct in the system speed-up than software. The details of the graphics pipeline will be explored in Chapter 5. The parallelism will be discussed in Chapter 6.

In biophysics, GPUs fulfil their potential for solving computationally complex, large problems. Also with the parallelism, large throughput, and high computational ability, GPUs have produced a considerable performance in protein sequence search (Xiao et al 2011), dynamic protein structural comparison (Bonnel and Marteau 2012), and molecular modelling (Daga et al 2011, and Lampe et al 2007).

The neuroscience is another field that embraces the GPUs for the solution to its huge computation in simulation (Baladron et al 2012). In this field, to describe brain activities, scientists design models of neurons and networks. They want to test hypotheses about

how the brain operates. The number of model neurons must be huge enough to simulate a real biological brain. It is an extremely computational challenging. GPUs have supported to do the simulation of spiking neural networks (Yudanov and Reznik 2012), visual neurons (Egashira et al 2012), and stochastic dendritic neurons (Karol et al 2011).

The basic idea beneath these applications is that the programmable unit of the GPU adopts a single-program multiply-data (SPMD) model. In this model, a single program is executed in all the elements. Each element does its task in parallel independently of others and does not communicate with others.

From the above discussion, it is obvious that GPUs have the strength in speeding up computation in parallel. But the GPUs evolved from the model with a fixed-function special-purpose processor. The fixed-function model with special-purpose functionality is the main characteristic of GPUs even though a lot of effort has been put in to make their processing programmable. It can also be seen that GPU applications go into two directions: one is a full speed-up solution to the graphics pipeline; the other is the computing speed-up for large computation-intensive problems. These mean that GPUs have limitations on the flexibility in the hardware structure building and the applications in heterogeneous problems.

Compared to commercial video designated cards that adopt GPUs mostly, FPGAs give alternative solution to graphics applications with their flexibility in the hardware structure building.

2.2.2 FPGA Applications

At the very beginning, FPGAs were mostly used in signal and image processing, neural networks (Nagendra et al 1993, and Van den Bout et al 1992).

After that, FPGA-based applications expanded quickly. Many fields have adopted FPGA technology, such as aerospace system, ASIC (Application-Specific Integrated Circuit) prototyping, automotive, consumer electronics, bioinformatics, cryptography, computer hardware emulation, DSP (digital signal processing), industry control systems, medical imaging, optimisation, portable applications, software-defined radio, speech recognition, and others (Chelton and Benaissa 2008, Chrysos et al 2012, DeGroat et al 2008, Gao and Long 2008, Kasik and Chvostkova 2013, Liu et al 2009, Lopez-Ongil et al 2005, Melnikova et al 2009, Monmasson and Cristea 2007, Smith and De La Torre 2006, and Turqueti et al 2010). The FPGA technology has assisted and supported these researches to stay at the leading edge in their fields.

In the DSP, numerous units for integer multiply-accumulate operations in a high-end FPGA provide the ideal building blocks for high data-rate DSP. Several design flows for DSP-oriented input specifications have been formed by some manufacturers' design tools,

for example, National Instruments LabView and Xilinx System Generator (Simulink) flows.

In scientific computing, the potential of FPGA technology for computing acceleration has been well understood, but there is still a mismatch between the design tools supported by FPGA manufacturers and the compilation flows expected by FPGA-based application developers. Another obstacle in the scientific computing is that the numerical analysis methods, which are used often in the scientific computing, have not been mapped effectively into FPGAs. It is also the similar case for other complicated algorithms in the scientific computing, such as the wavelet methods for time series analysis and analysis on fractals. Before scientific applications can be ported to FPGAs, it is the key to find the operations that are simple and common in the scientific computing, and can be transformed into reusable building blocks in FPGA devices.

In the computer graphics, since GPUs have attracted most attentions of the academia, industry and governments, GPUs have dominated the entirely speed-up of graphics pipelines in general-purpose computers. FPGAs are developed only in the image processing. With the similar performance on the parallelism, throughput and computing ability as GPUs, FPGAs have not acted as an appropriate role in this field. To inspect this phenomenon, let us make a deep analysis between FPGAs and GPUs.

2.2.3 FPGAs vs. GPUs

FPGAs have the similar features of high density and excellent parallelism as GPUs. Both of them benefit from the constant progress in semiconductor technologies. In all the academic, industrial and governmental societies, however, GPUs are more attractive than FPGAs. In these two decades, more activities of research, development, and investment are promoted to enhance conditions for mapping scientific applications onto GPUs. The specification languages, design environments and compiler technologies for GPUs have been formed quickly (Constantinides and Nicolici 2011). In spite of having a two-decade evolution history, FPGAs do not have an open, completed, and generic platform for the application design and development. Thus, the research, development and investment on FPGAs are separate and indifferent.

Because of the inconvenient design methods and tools for FPGA-based accelerators, many applications turn to general-purpose graphics processing unit (GPGPU) for their solutions. In fact, since their configurability ability, FPGAs have more potential to provide a large number of processing engines in parallel on a silicon die than GPUs. FPGAs need new tools to equip developers and engineers who are used to using compilers such as C compiler for microprocessor programming in order to migrate to FPGA platforms without needs to learn new languages and design environment.

GPUs dominate in the research, development and investment by productivity rather

than performance reasons (Constantinides and Nicolici 2011). In fact, GPUs dominate the graphics speed-up in general-purpose computers, but do not in ESs. Because the full potential of FPGAs has not been realised, FPGAs should be invested more attention and effort in development and applications in order to fulfil their potential in ESs. For the above reasons, an FPGA platform is chosen as the platform for this research.

In addition, in general-purpose computers, CPUs are the central microprocessors of systems for computing. Without the speed-up GPUs, CPUs have to manipulate graphics pipelines. This architecture is still adopted by low-end computers. Thus, let us make another comparison between FPGAs and CPUs.

2.2.4 FPGAs vs. CPUs

Compared with CPUs, solutions with FPGAs have advantages and disadvantages. Table 2.1 displays the comparison.

Table 2.1 Comparison between FPGAs and CPUs

	CPUs	***FPGAs***
Power Consumption	High	Low
Parallelism	Instruction Processing in Sequence	Inherent Operation Parallelism
Design Complexity	Low	High
Increasing Speed in Capability	Low	High

2.2.4.1 Lower Power Consumption

Modern FPGAs exhibit a lower order of magnitude of power consumptions than CPUs. For example, Altera Cyclone III FPGAs with up to 200K logic elements claim to consume less than 0.25 watts whereas Intel Core i5-460M, a high end dual core CPU for laptops, requires 35 Watts.

2.2.4.2 Higher Parallelism

Because of its architecture, a microprocessor tackles an application as a sequence of instructions while the logic blocks in an FPGA can be configured to operate in parallel. Compared to the software executed on a CPU, the inherent parallelism of FPGA logic resources can offload time intensive operations from the CPU.

2.2.4.3 Higher Design Complexity

The design complexity of FPGA solutions is higher than software solutions by the CPU. It is an obstacle for promoting FPGA solutions to problems. A minor change in the software

can take several minutes to re-compile with a compiler, but a minor adjustment in the hardware may take several hours to re-build on the FPGA design platform.

2.2.4.4 Higher Increasing Speed in Capability

According to studies of Arden and Awad (Arden 2002, Awad 2009, and Kahng 2013), FPGA technology would be ahead of microprocessors in increasing speed for the recent 15 years because FPGAs follow the International Technology Roadmap for Semiconductors, rather than the microprocessors roadmap. The FPGA density is growing at the rate of advanced CMOS (complementary metal-oxide-semiconductor) technology, which makes its size decrease by a factor of 1.26 per year. With time, as the FPGA community settles CAD (computer-aided design) tools limitations, FPGA will affect even more on most digital logic designs and implementations.

Predicted by Awad (Awad 2009), a hybrid system with multi-core CPUs and FPGAs operating in tandem with parallel-core GPUs can be expected to offer an enhanced hardware performance and programmability for supercomputing platforms.

2.3 Introduction of OpenGL, OpenGL ES and their Implementations

In the computer graphics, the OpenGL is a widely-accepted standard for 2D (two-dimensional) and 3D graphics applications with user interactions. In the OpenGL, the graphics pipeline is a state machine that can be implemented with an OpenGL-capable computer (Kilgard 1997, and Kilgard and Akeley 2008).

The graphics pipeline consists of two sub-pipelines, geometric pipeline and fragment pipeline (Hearn et al 2011, and True et al 2004). The geometric pipeline processes the vertex coordinates of displayed objects through a sequence of coordinate transformations, including transformations from vertex coordinates to eye coordinates, to clip coordinates, to normalised coordinates, and finally to window coordinates. After rasterisation, the fragment pipeline processes the pixels in frame buffers to make the images of displayed objects rendering on a display screen. The fragment pipeline includes steps of texel generation, depth test, stencil test, alpha blending, and logical operations. These steps can be enabled or disabled according to whether or not they are required to perform during rendering.

The OpenGL ES (OpenGL for Embedded Systems) is one of OpenGL standards specified for ESs (Angle and Shreiner 2008, and KHRONOS 2013). It can be applied to automobile digital parts, hand-held gadgets, and mobile phones. Since there are wide applications in ESs, the OpenGL ES consists of versions and profiles for different applications (KHRONOS 2013).

Because there are a broad range of display devices and platforms in embedded

markets, the OpenGL ES has two versions for two different development requirements and platforms, respectively. These are OpenGL ES 1.X and 2.X. The OpenGL ES 1.X is specified for the fixed function hardware in order to decrease the memory bandwidth, to enhance image quality and graphics performance, and to improve hardware acceleration. The OpenGL ES 2.X is set for programmable hardware. It consists of specifications for a programmable 3D graphics pipeline. They include the definitions of constructing shader objects and programming vertex and fragment shaders in the OpenGL ES Shading Language.

Three profiles of the OpenGL ES have been released, which are Common, Common-Lit, and Safety Critical Platforms. The Common Profile is designated for consumer handhold devices, such as PDAs (personal digital assistant), cell phones, game consoles, television set-top boxes, and others. The Common-Lite profile primarily concentrates on a simpler class of graphics system with a requirement on even less footprint. It only supports the fixed-point calculations. The Safety Critical Profile is specified for consumer and industrial surroundings that have high requirements on being certifiable and reliable. It can be used in 3D graphics applications of safety certifications, and avionics and automotive displays.

No matter the OpenGL or OpenGL ES, they must be implemented on different platforms. Since the hardware system of one platform is different that of another, the implementations on them are different. Companies, such as Intel, Imagination Technologies, ARM, Apple, and NVIDIA, have their own implementations of OpenGL ES (KHRONOS 2013). Mesa (Paul 2013) has many implementations of OpenGL for different general-purpose computer platforms, which are free and open-source.

Since the existent implementations of the OpenGL ES do not encompass one specified for FPGA-based ESs and the algorithms of Bézier curves and surfaces are not included in specifications of general OpenGL ES, a Mesa-based implementation of OpenGL is carried out in this research in order to meet the needs of surface modelling and editing with the PAMA.

2.4 Investigation of Traditional Computation Parallelism

As the hybrid way is adopted in the construction of FPGA-base ES in this project, the parallelism is considered naturally to enhance the performance of the system. Thus, it is necessary to investigate the traditional computation parallelism and to find a root to expand the concept of the traditional computation parallelism to meet new requirements of FPGA-based ESs and hybrid methods.

At the beginning, the emphasis of parallel algorithm design was on messaging and loop-based parallelism, and on precise mapping of tasks to specific topologies such as

meshes and hypercubes. It has evolved into the programmability and portability of parallel algorithm design and implementation (Grama et al 2003).

In computer systems, the principle at the heart of parallel algorithms, which is the locality of data reference, provides a solution to cache-friendly serial algorithms. It has been extended to the development of out-of-core computations.

2.4.1 Data Parallelism: SIMD and MIMD

Traditionally, the parallel architecture can be classified in two large classes: single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD) (Jonker and Vogelbruch 1997, Krishnakumar et al 2011, Nomoto et al 2011, Pitas 1993, and Wang and Ziavras 2006). They have been developed in digital imaging processing, computer vision, neural networks, and other fields. Such machines exploit data level parallelism.

The SIMD is a class of parallel computers in the Flynn's taxonomy. It describes computers with multiple processing elements (PEs) that perform the same operation on multiple data simultaneously.

SIMD machines are the first ones that appeared in applications of digital image processing and computer vision. Most of them exploit cellular logic arrays. They are composed of large arrays of simple one-bit PEs. All the PEs form a processor grid by connecting each of them to its immediate neighbours. Instructions are broadcast to all PEs. Using local data transfers, it can perform local neighbourhood operations synchronously. The advantage of cellular logic arrays is to make the best use of both geometrical and neighbourhood parallelisms. However, it has some disadvantages. One is that it can process an array with a small size of 128 X 128 pixels simultaneously. An image with a typical size of 1024 X 1024 pixels must be split in segments and each segment must be processed independently. When multiple local operations must be applied in pipeline, it can result in border effects that may seriously influence the image performance. The second disadvantage is that it can bring a heavy I/O load to the system. The third one is its restricted ability in high-level vision.

In computing, the MIMD is another technique employed to achieve parallelism. In contrast to the SIMD, in the MIMD, any PE is able to execute a different program independently of the others. Machines using the MIMD have a number of processors that function asynchronously and independently. At any time, different processors may execute different instructions on different pieces of data. MIMD architectures may be used in a number of application areas such as CAD, simulation, modelling, communication switches, image processing, and computer vision. In digital image processing applications, massive MIMD machines have been developing for at least three decades.

MIMD machines can be divided in two categories: common shared memory and

distributed memory. This classification is according to how MIMD processors access memory.

In common shared memory architectures, all processors access the same memory through the bus. They are extendable with a hierarchical structure. Their advantage is that since data are stored in the common shared memory, any processor can access them at any time. The disadvantage is that the competition in memory access among processors can occur.

In distributed memory architectures, each processor has its own local memory and communicates with others via a common bus, communication link, or both. A hypercube or mesh interconnection scheme may be adopted in distributed memory architectures. In this case, data must also be divided into small chunks and distributed among processors. It means that data exchanges between processors are required when data processing algorithms execute. Data exchanges can aggravate the communication load and reverse the speedup expected. If a common bus is used, bus congestion may occur as well. If the serial communication links are used, typically in DSPs, the small bandwidth of serial links for most data processing applications can be a bottleneck. A modified scheme is to combine a high-speed common bus with communication links. The former is specified for the data transfer; the latter are used for the message passing.

In the image processing, for example, the SIMD architecture is used in the low-level processing while the MIMD is applied in the high-level processing. In the middle, it can be a mixed architecture that is a hybrid or mixed one of SIMD and MIMD (Siegel et al 1981).

2.4.2 Operator Parallelism

An alternative parallelism approach is operator parallelism. In this architecture, each operator can be optimised for specified tasks and be linked together to accomplish a sequence of tasks (Pitas 1993, and Ramasubramanian et al 2002). Thus, it is also called as pipelining even though it has different functions from the pipeline that will be discussed in detail in the next section. There is a special architecture in the operator parallelism, the operator parallelism within loops. To form a loop, the starting processor in the pipelining can be linked to the end processor. This loop structure can be used in a situation where the same operations must be performed on the data cyclically and repeatedly.

In DSPs, the operator parallelism has been widely used in the implementation of low-level digital signal or image processing algorithms. It has been proved that the operator parallelism is very efficient for the morphological image processing. In special-purpose integrated circuit boards, heterogeneous pipelines are successfully used in operating on a high-speed image bus. With integrated circuit board drivers, the pipelining function can be controlled by a host computer.

2.4.3 Pipelined Processors

Pipelined processors for real time low-level image processing have been developed for three decades as well (Kelly and Hsu 1998, Kim H. et al 2012, Kim J.K. et al 2012, and Pitas 1993). These processors receive data in the raster scan order and pass them to the processing pipeline sequentially. Each stage in the processing pipeline performs the same pre-specified operation on every element of the data in sequence. Pipelined processors have been applied in processing algorithms for pixel-wise and local images. The advantage of pipelined processor is to process the image in real time. Incoming data can be accepted from the image sensing device directly. The intermediate processing results are not necessary to store, which simplifies communication between stages. Their disadvantage is the lower flexibility on the hardware devices. Once specified and built up for a special algorithm, devices cannot be changed for other algorithms.

2.4.4 Cache

Compared to the above parallelism technologies, it is not obvious that the caching technology is included in the parallelism. It is known that there is a speed mismatch between processor and DRAM (Dynamic Random Access Memory). One reason for this is the memory latency, a period of time needed by the memory device for preparing the data to transmit; the other reason is the low bandwidth of memory, which is the rate of data transmission between the processor and memory device. The memory bandwidth is determined by the size of memory blocks and the bus bandwidth of the memory. Memory blocks are the smallest units that can be physically fetched to the cache each time.

The cache, which acts as a temporary low-latency high-bandwidth storage, is used to speed up the memory access with the reuse and locality of code and data (Grama et al 2003). The reuse means that a section of code or data can be re-accessed more than once. The locality is the feature that the access to a section of code or data can be kept for a while when a program is executed. If several of consecutive and contiguous blocks are pre-fetched to the cache in subsequent bus cycles when the first word is retrieved, the subsequent consecutive blocks can save the memory latency. Both of the reuse and locality are features that the programming mostly has.

There are varied types of cache. The lower latency and higher bandwidth a cache has, the more expensive it is. Thus, a system can have multiple levels of caches. The lower-latency and higher-bandwidth cache is put closer to the processor. The system's performance is improved by raising the rate at which data can be transmitted into the CPU, rather than by increasing the processing rate of the CPU.

This structure does not belong to typical traditional parallelism architectures. As the multiple levels of caches are cascaded in between the processor and DRAM, it can be

treated as an implicit pipeline.

2.4.5 Promotion for New Concept Introduction to Parallelism

Driving the parallelism evolution is needed to speed up the computation, which is always required by advanced applications. Since there is no limitation for the requirement of speeding up computation, the parallelism in computer systems and applications becomes diverse, flexible, and changeable. Researchers have attempted many different ways in different hardware platforms to extend and even break through the conventional parallelism concepts. Different application situations can spawn different methods, and lead to different solutions.

For example, a pipeline has many variants in different parallelism contexts, which will be further discussed in Chapter 6. Applications in graphics and visualisation can use multiple rendering pipelines and PEs of different levels in parallel in order to compute and render realistic scenes with millions of polygons in real time. This parallelism simulates but does not exactly map rendering tasks to specific topologies of processors.

Another issue is the capability of hardware system where a parallelised algorithm is implemented may influence the speed. The effect of speeding up of the parallelised algorithm relies on whether or not the capability of a hardware platform is sufficient for this parallelised algorithm. This may result in modifying the parallelised algorithm in order to speed up the computation at that hardware platform. The parallelism can produce an effect in that system context, which is different from those in other system contexts.

In graphics and video processing, the pipeline, however, can be transformed to have a cache-like function. Just like the mismatch between processor speed and the memory latency needs a hierarchy of successively faster caches to compensate, the mismatch between frame buffers in memory and pixels processed and displayed line by line on a device screen has to be pipelined to display images smoothly on the screen. Therefore, a new type of parallelism can be modified and synthesised.

These new concepts of parallelism are applied to this project and will be detailed in Chapter 6.

2.5 Related Studies in Surface Modelling and Editing

According to Cunningham 2008, and Hearn et al 2011, many methods and technologies of other areas are needed in the computer graphics, such as the geometrics, numerical analysis, approximation theory, and interactive computer systems, to support the representation, manipulation and display of free-form surfaces. These free-form surfaces can represent varied objects. But the computer graphics has its own features and rules to

model objects in order to display them on computer screens.

For the 3D rendering, it needs to go through the whole graphics pipeline that an OpenGL implementation can provide. From the application perspective, it includes modelling, several coordinate transformations from the local coordinates to scene world coordinates, to viewing coordinates, and to projection coordinates, clipping with view volume, and mapping viewport coordinates to screen device coordinates.

In order to display an object properly on the screen, it is fundamental to model the object in a controllable, flexible, and effective way at the beginning. The object can be modelled with a surface or mesh. The surface can have a set of control vertices that are used by the user interaction for surface editing (Barsky 1984, Cunningham 2008, Hearn et at 2011, Hoppe et at 1994, and Hoschek and Lasser 1993).

In many applications of computer graphics, for instant, computer-aided geometric design (CAGD) and CAD, the manipulation of surfaces or meshes is necessary because users expect to edit and modify the object shapes according to their design intentions. Therefore, for surface/mesh editing, it is important to model objects in a controllable and flexible way.

There are many methods for object modelling and editing in 2D or 3D, which have been published in these three decades (Abbasinejad et al 2013, Cheng and Goshtasby 1989, Duchamp and Stuetzle 2003, Hoppe et al 1994, Huang et al 2006, Loop and DeRose 1989, Loop 1994, Nasri 1987, Perez et al 2003, Sederberg et al 2003, Sorkine et al 2004, Wang et al 2006, Welch and Witkin 1994, and Yu et al 2004).

For the 3D rendering, there is a general purpose, which is to make the geometrical images as smooth as possible. For this goal, there are three mainstream schemes to maintain the local control during surface modelling and editing, these being subdivision, deformation, and shape parameterisation.

The multi-time reuse of subdivision in different levels can create a limit surface with visual smoothness (Catmull and Clark 1978, Deng and Ma 2013, Doo and Sabin 1978, Kazakov 2007, Lin et al 2008, Müller et al 2006, Müller et al 2010, Nasri and Abbas 2002, Patney et al 2009, and Sederberg et al 1998). Since the limit surface is the convergence of infinite-time applications of subdivision, there is always a compromise between computation cost and surface smoothness during practising subdivisions. Depending on the smoothness and local geometric details to be attained, this compromise varies and the number of subdivisions to be applied is empirical. That is, when using the subdivision method, designers may use fewer times of subdivisions if the situations allow the lower smoothness and fewer local geometric details. Or, they may use more times of subdivisions if the cases need the higher smoothness and more local geometric details.

Deformations of solid primitives can be assembled in a hierarchical structure to create complex objects (Barr 1984, Botsch and Sorkine 2008, Clack and Keyser 2013, Lawrence and Funkhuser 2003, Sederberg and Parry 1986, and Sumner et al 2007). In order to change the shapes of primitives, deformations have several hierarchical solid modelling operations to support user interactions. Deformations provide a flexible way to construct 3D geometrical shapes. The disadvantages are that the assemblies of different levels of primitives can result in the unexpected non-smoothness between different primitives. The more complicated the target object structure is, the more assembly layers are required, and the more calculations are involved.

The shape parameterisation can be used to get local control with the predicable surface smoothness and without multi-time reuses to reach the smoothness (Barsky and DeRose 1989, and Barsky and DeRose 1990). The local controls are not divided into any hierarchical structure. Seen from publications, the aforementioned two schemes involve a lot of researches of different individuals and groups, but the last one does attract much less attention after the study of Barsky and DeRose 1990.

Table 2.2 displays comparisons among these three methods. As the above discussion, the multi-time reuse is required by subdivision methods, rather than by the two others. The assembling sub-models are needed by deformation methods, but not by the two others. Without primitive assembles of deformation methods, the surface parameterisation methods offer a definite smoothness. The reason for this is that the geometric continuities on the common boundary of two connected patches are the conditions that are used to construct the algorithms for surface parameterisation methods, which will be explained later. Without intermediated processes, the surface parameterisation methods have lower storage and computing cost than subdivision methods. Since the multiple applications and primitive assembles are eliminated, the computation cost of surface parameterisation methods are lower than the other two.

In addition, for the surface modelling and editing, the user interactions are necessary and require the operations with the input and output system. The times of user interactions caused by the surface editing engender the same times of input/output operations. The latter leads to the computation cost and makes the time cost even more because the time cost is the sum of the time cost by both the computation, and the input and output system. Thus, the time cost of surface parameterisation methods is much lower than two other methods. Moreover, considering that it is applied to an FPGA platform, which is limited in the storage space and computing ability, a shape parameterisation method is even better than the two other methods.

The novel algorithm, PAMA, proposed in this thesis, can be classified into shape parameterisation.

Table 2.2 Comparisons among Three Surface Modelling and Editing Methods

	Subdivision	*Deformation*	***Shape Parameterisation***
Multi-time Reuse	Required	None	None
Assembling Sub-models	None	Required	None
Smoothness	Varied, Empirical	Non-smoothness Between Primitives	Smooth
Hierarchical Structure	Depending on Geometrical Details	Required	Flatted in One Level
Storage Cost	High	High	Low
Computation Cost	High	High	Low
Time Cost	High	High	Low

Among many existent spline algorithms, B-spline (Cheng and Goshtasby 1989, Forsey and Bartels 1988, Liu et al 2009, Loop and DeRose 1990, and Martin et al 2008) and Bézier-spline (Efremov et al 2005, Hagen 1986, Li et al 2006, Loop and DeRose 1989, and Si and Guenter 2010) can smoothly approximate an irregular control mesh. They are often used as a basis, and proved simple, efficient, and of a polynomial form.

Bézier-spline curves and surfaces provide a convenient method for interactive design applications. A curve or surface formed with two Bézier sections or patches can be established with a zero-order and first-order parametric continuity (C^0 and C^1) at the common boundary point or line (Barsky and DeRose 1989, and Hohmeyer and Barsky 1989). Since the degree of the Bézier curve is determined by the number of all control points to be approximated and their relative positions, Bézier-spline curves and surfaces have a limited local control on control points. Parametric continuities can shrink the set of parameterisations by excluding ones that can generate geometrically smooth curves. The reason for this is that the condition of a given order of parametric continuity is stricter than one of the same order of geometric continuity, which will be explained later in this section.

Beta-splines have many merits (Barsky and DeRose 1990, Farin 1993, Hearn et al 2011, and Hohmeyer and Barsky 1989). They can acquire constructed curves that are smooth and fit control points. They can also add Beta-constraints to a curve so that Bézier curve segments can be joined together with geometric continuities and more controls can be provided with shape parameters for users to edit curves interactively. For each control point of the curve, each of shape parameters has its own efficacy for changing the curve shape, which is different from changing the position of control point.

Cubic Beta-splines can give the second-order geometric continuity (G^2), which is more relaxed than the second parametric continuity (C^2) (Barsky and DeRose 1989, and Hohmeyer and Barsky 1989). With shape parameters that ensure Beta-constraints, Beta-spline curves can match parametric derivatives and lead to local controls on curves. In the study of Barsky and DeRose 1989, the authors give an application of stitching cubic Bézier-spline curves together with the first-order and second-order geometric continuity (G^1 and G^2), which is a development of cubic Beta-splines with G^2. The scheme of joining cubic Bézier-spline sections together with the G^1 and G^2 conditions (or Beta-constraints) is first proposed by Farin (Farin 1982), enhanced by Boehm (Boehm 1985), and applied to Beta-spline curves by Barsky and DeRose (Barsky and DeRose 1989). It has palpable geometrical meanings and is feasible to construct the polygons of cubic Bézier-spline sections with G^2 common boundaries, which will be explored in Appendix. Following these thoughts, the PAMA attempts to extend the geometrical construction scheme to Bézier-spline patches of a composite surface with the common boundary curves that hold specific-order geometrical continuities.

With Beta-splines, the curve construction is given by several researchers, such as Barsky and DeRose (Barsky and DeRose 1990). For the surface construction, a special case is presented in the article of Barsky and DeRose (Barsky and DeRose 1985), which has the same shape parameters in two parameterisations. In the article of Joe (Joe 1990), Beta-spline surfaces are constructed with an algorithm, in which one of the shape parameters (β_{u2}) is set to one for all control points, and the computing cost is reduced while the control of these shape parameters is lost. There has been no recent published work on comprehensive extensions of surface modelling with Beta-splines or Beta-like-splines of four shape parameters changing independently in two parameterisation directions.

Considering the tension product of Beta-splines surfaces can result in a large computational cost of multiplication and the shape parameters of each control point should be manipulated independently for the design purpose by user interactions, this research proposes and applies a Beta-like-splines of four shape parameters to surface modelling and editing through user interactions. The Beta-like-splines proposed in this research are not deduced directly from the tensor product of Beta-spline blending functions, but derived with a progressive and mixing way, which takes account of both the geometric continuities of surfaces and the shape parameters of two parameter directions naturally adding to the surface construction. These lead to the novel algorithm, PAMA, which will be detailed in Chapter 7. Open and closed spline surfaces have been constructed and edited with the PAMA through user interactions and are also shown in Chapter 7. In addition, the applications of the PAMA in two environments of the general-purpose computer and

FPGA-based ES are presented in Chapters 7 and 8, respectively. For the coherence of the thesis, a lucid exposition of the rigorous mathematical deduction of the PAMA and its continuities will be presented in Appendix.

2.6 Chapter Summary

In this chapter, five aspects of literature reviews are presented, which are relative to five aspects of this project. The first aspect is Section 2.1, the related studies of ESs. The second one is Section 2.2, the investigation of hardware graphics applications. The third one is Section 2.3, the introduction of OpenGL, OpenGL ES and their implementations. The forth one is Section 2.4, the investigation of traditional computation parallelism. The last one is Section 2.5, the related studies in surface modelling and editing. The contents in these sections are followed by the rest chapters of the thesis, respectively. Chapter 3 discusses an integrated hybrid ES following the discussion in Section 2.1. Chapter 4 explores the FPGA-based embedded hardware system for graphics applications, which deepens the investigation in Section 2.2. Chapter 5 details integrating the Mesa-OpenGL implementation into FPGA-based ES, which follows the introduction in Section 2.3. Chapter 6 explores the parallelism implementation in FPGA-based ES that deepens the investigation in Section 2.4. Chapter 7 focuses on the novel algorithm for surface modelling and editing, PAMA, which deduces from the studies of Section 2.5.

Chapter 3 An Integrated Hybrid Embedded System

The discussions of Sections 2.1 and 2.2 clearly suggest that to obtain a hybrid solution of software and hardware for graphics applications, we need to choose a platform that allows us to construct a system as a whole by considering both hardware and software. This has the merit that the ES outstrips general-purpose computers. In addition, since the ES is known as an interdisciplinary field, we need to investigate the features of ES design and gradually explore how to use an ES model to construct a goal system for graphics applications and allow the PAMA to execute on this system.

3.1 Features and Principles of ES Design

According to the study of Henzinger and Sifakis (Henzinger and Sifakis 2007), an ES is an engineering artefact involving computation that is subject to physical constraints. In other words, even though involved in the computation, the ES is a physical system that has to meet the physical requirements and be implemented with engineering methods.

3.1.1 Physical Requirements for ESs

The physical requirements for ESs are twofold. One is responding to its physical environment; the other is performing operations on its physical platform. These operations can be, but are not limited to, computations. They can be the moving of actuators, receiving sensor signals, transmitting signals, and many other controls. The ES mainly interacts with its environment. The interactions can be from human beings' control with consoles and to a monitor, or making a response to a sensor signal, such as turning a robot around when it is about to bump into a wall. Both the response ability and operations can significantly influence the ES performance.

3.1.1.1 Response

Requirements for response include deadlines, throughput, and jitter performance.

Because of real-time requirements, the response of ES to an outside event has a start

or completion time as a deadline. Beyond this deadline, the response is pointless. In other words, the real-time response must be able to keep up with the event with which it is concerned and be started or completed before this deadline. Otherwise, it will lead to a fatal failure to the system or undesirable damage.

The throughput has a different meaning for an ES from a general-purpose computer. For a general-purpose computer, the throughput indicates the amount of data that the computer can accept and process without missing any of the data. For an ES, except for the amount of data to be processed, the throughput usually means the processing ability of the ES. That is, the ES must tackle all the specified operations in real time without missing any of them. In other words, the operations can comprise not only a large amount of data to process but also a sequence of responses to a series of events.

Jitter problems are natural but undesired for electronic circuits. They can occur when the switches in electronic circuits are turned on and off and lead to instantaneously undulation of signals. Without properly protection, they can cause ESs to malfunction. Thus, jitter performance is critical for ESs. Protection measures for jitter problems can come at the cost of system resources, however.

3.1.1.2 Operations

In ESs, the requirements of operations involve the speeds of on-board processors and the capabilities of system resources. The processors and system resources are related to the price for which the end products can be paid, the sizes and capabilities of chips and boards, and power capabilities. Hardware failure rates can also have an influence on operations. These requirements affect each other.

Usually, when considering the higher cost restriction and less computation, ESs use slower processors than general-purpose computers do. In general, for general-purpose computers, for which computation is the main task, the higher the speed of their microprocessors, the better their system performances. For ESs, because of the cost restriction, a proper option can be a processor with a speed fast enough for response to their application environment, or a processor with a speed sufficient for the completion of specified operations before the deadline. Similarly, the size of memory, which is one of the most important system resources, is also restricted to an appropriate range for specified applications of ESs.

The sizes and capabilities of chips and boards can impose restrictions on the on-board processor option. Because of their application environments, ESs often do not have large space for accommodation. A small size is the regular choice. Higher capabilities of chips or boards mean higher costs. An unrealistic high-quality chip or board is not compatible with the limited budget of a mass electronic-consumer product.

Because of the limited space, the power units of ESs cannot be large. Small sizes lead to power restrictions. Greater power means higher cooling requirement. Usually, rechargeable batteries are alternative power sources for ESs. For rechargeable batteries, a large size is inappropriate.

Lower hardware failure rates are often required, especially for the applications in life science, medical instruments, and avionics. Decreasing hardware failure rates means increasing the cost in terms of enhancing the check and protection units of systems.

Roughly, to decrease hardware failure rates, there are two levels for safety-critical requirements in ESs, hard and soft. For hard safety-critical ESs, the necessary computing power must be guaranteed at all times. The needs of the static scheduling and worst-case execution time analysis must be met in real time. Hard deadlines must be met. System reliability must be guaranteed with massive redundancy technologies and measures for failure detection and recovery. These ESs are usually used in life science and avionics.

Soft safety-critical ESs make a soft demand on service quality. The efficient use of resources is the goal for this type of system design. The applications for this type of systems can tolerate some service degradation or even temporary service shortage. Soft deadlines are not too serious and allow a small amount of being missed. Thus, the best-effort mechanism can be adopted. The best-effort mechanism tries to balance system performance and cost when systems go into degenerated situations. It can get feedback from the degenerated situations and adjust some parameters to maintain an acceptable performance or to recover from the degenerated state at the run time.

Communication networks and multimedia systems belong to the soft safety-critical ESs. In these systems, the best-effort scheme is often used to guarantee a lower failure rate. Even more, since the workloads of these systems vary dynamically at the run time, the best-effort scheme can be used to adjust performance according to different workloads. It uses different users with different priorities or data traffic to balance performance levels of the whole systems during different periods of time.

All the above requirements decide the implementation options on ESs. Since they influence each other, it is necessary for ESs to find a balance among these requirements. Control theory and computer engineering can help to balance the capabilities of response and operation of ESs with all the requirements. Therefore, ESs need to carry out a design that can meet a designated set of requirements on a given implementation platform.

3.1.2 Analysis for ES Design

Since an ES design involves both hardware and software design, it is necessary to make an in-depth analysis on regular paradigms of software and hardware designs, respectively. Then it can be seen clearly how to progressively build up a new way with current design

conditions for the ES design.

3.1.2.1 Software Design Paradigm

In software engineering, a system design employs an abstract representation of the system requirements. The abstract system representation is a model that follows the standards and rules of software engineering and some high-level language syntax. The model can give a description of a target system design. A designated language compiler uses the model to generate automatically a system that meets the given requirements. This system is a piece of software that is usually the collection of the machine codes and can be executed on a computer.

Software design is usually carried out on a virtual machine. This machine is an abstract one defined with a real interface via which application software can interact with a real machine. No matter which machine it is, the interface is real and generic for all types of machines. Since it consists of different functional units, the standard input and output system is divided by functions and defined in specific blocks. These blocks are available for software calling via the interface. Therefore, software design does not necessarily analyse the content of the virtual machine but only knows how it works. A real machine can be uncertain before the software is ported to it. But the real interface can guarantee software portable among real machines with the generic interface.

On the virtual machine, the software design is set up. The software is conventionally designed as a model that consists of sequential blocks. These blocks can be instructions and routines representing threads and objects. The implementation of each block is changeable depending on the actual algorithm, but the function and interface of each block must be clarified and consistent. Designers can add and delete individual blocks according to their target system's requirements. Designers can define the blocks' layout by specifying the control flow among them.

Software programming usually does not treat timing as seriously as hardware construction does. One reason for this is that the computation result is often more important than the computing process. The other reason is a side-effect of high abstraction. High abstraction encapsulates and hides CPUs and memories from the applications in a computer system. It also hides how the computation result is being attained. On the other hand, the hardware knows that all the processing in a computer system is done with a series of actions of processors controlled by the system clock, no matter whether the processing is computing or controlling.

Time-sharing systems have led to greater efficiency of computer systems by making the CPU busy manipulating several jobs concurrently. In this architecture, one CPU's working time is shared by several different jobs. This sharing idea has been extended in many other applications, such as sharing different system resources among different tasks or

users in telecommunication networks and multimedia systems. These sharing models attempt to satisfy every task or user by sharing and averaging services among them and can result in some resources being temporarily unable to serve some users. They do not let any of them occupy a resource exclusively. These models can be realised with just a system counter to schedule the tasks. There is no critical task that is so pressing to require timing and have an absolute priority that it occupies the system resources exculsively. Thus, serial or concurrent processing can meet most of demands.

In general, a piece of software is a computation process to execute on a machine. It is important whether its result is correct or not. It does not matter how the computation is accomplished. Thus, it can be implemented in sequence. It can be highly abstractive, device-independent, and transparent. The implementations of a software design can vary.

3.1.2.2 Hardware Design Paradigm

Hardware design is a totally different scenario. Every component has to be verified before execution in order to avoid the target system getting out of control because each of them has its own necessary task and makes a contribution to the whole system, which is deduced from the system requirements.

Even though there are different abstraction levels to describe building blocks in hardware design, each block has an explicit definition of its components, for example, logic gates and transistors. The abstraction blocks can be functional blocks, such as adders, multipliers, and others, or architectural ones, such as processors, multi-core architectures with caches, etc. There are fewer options for replacing one type of device component with another in hardware than software. Since the signals that represent different data transmit along different parallel branches in a complex electronic circuit, the parallel feature is inherent in hardware design.

As regards top-down design, a design model can consist of dataflow diagrams and netlists. The dataflow diagrams can represent transfer functions of a target system. The netlists can be a description of connections between different blocks.

As regards the hardware implementation of a system, FPGA devices can be adopted, which will be detailed in the next chapter. The basic idea is that with a computer-aided design (CAD) tool, such as Altera Quartus, Xilinx ISE, Mentor Graphics HDL Designer, or Synopsys Synplify, a system representation written with a HDL (hardware description language), such as VHDL (VHSIC HDL, very high speed integrated circuit hardware description language) or Verilog HDL can be programmed and synthesised the target system circuit and downloaded into an FPGA chip. With a CAD tool, the representation can be translated into a RTL (register transfer level) schematic of a target circuit. The generated schematic can be verified by using simulation software to show the waveforms of inputs and outputs of the circuit. The verified schematic provides a lucid illustration of

the blocks and connections of the target circuit. The VHDL or Verilog HDL model is translated into the hardware description information about the gates and their connections. Then, the information is mapped onto an FPGA device. The FPGA device is configured with the model and ready to function as the target system.

In hardware design, the abstraction technique is used as well. For instance, real-valued transistors can be abstracted as Boolean-valued gate-level models, and now system-on-a-programmable-chips (SOPCs) provide even more abstractive models.

Compared with software design, timing is critical to hardware design. Timing is hard to express in building blocks of a model. Viewed in the bottom-up way, with gates, it can be clearly seen how many system clocks a process has to accomplish. But when the design is abstracted into different building blocks, unless the clocks that have to be cost by a unit are marked explicitly, it is hard to maintain exactly timing. On the other hand, in some control applications, such as those with hard deadlines, timing is critical. When a system is large, such as VLSI (very large scale integration), with millions of gates, the manual identification of timing is not realistic and timing analysis becomes necessary. In reality, given the FPGA background, the synchronisation of processing is encouraged more than asynchronous operations because synchronous units are easier to control in circuits than asynchronous ones. This is why the timing analysis is very important for hardware design, as well as logic analysis.

Many hardware designs are applied to a designated control or operation, often exclusively. They have one critical task, which occupies all the system resources in its critical time. It does not share the resources with others. This can result in the under-use of system resources. The deadlines of the critical task must be met. To guarantee the task's accomplishment without any known risk, massive redundancy in system resources sometimes has to be adopted. This is the case with hard safety-critical ESs, as mentioned in Section 3.1.1.1

In addition, it is relatively easier to express a logic statement by a specified language than to describe a hardware action, especially an action that is a series of operations controlled by the system clock. Hardware synthesis tools are more difficult to understand than automatic model transformations of software compilers. In hardware design, some stages, such as adjustments according to the results of logic analysis and timing analysis, cannot be done automatically by some tools. Without some expertise, skills, and experience, it is almost impossible for newcomers how to handle these tasks. The more human-guided decisions need to be involved, the more difficult it is to be understood. Therefore, hardware design needs more specialised training and practice.

Unlike a piece of software, which is a temporary execution process, a hardware system is a permanent physical machine that is expected to execute different tasks, which may be

a program or a operation of an actuator, at any time. Its error case may always exist and be tested after it starts to run. Therefore, the demands put on a hardware system are different from those put on software, and the verification process passed by a hardware system is more serious and detailed than software.

In the above two paradigms, the compilers or CAD tools are quite mature. The process of compiling or synthesising is automatic without too much manual work. It saves designers a lot of energy and time. The design formula has a well-organised structure. It allows not only the bottom-up style but also the top-down way to design. Two ways can be combined for making, adjusting, and refining the design models.

3.1.2.3 Comparison between Hardware and Software Developments

In the design and development of a system with a hybrid way, it is harder to add new functions to the hardware part of the system than to the software part. During the implementation, the variation and extension in the hardware part can cost designers and developers more time and energy than the software. Compared with software development, hardware development faces several restrictions.

- For regular developers, understanding hardware implementation is more difficult than understanding that of software. As the logical feature of software is more readable than the electronics engineering notation of hardware design, software implementation is easier to master.
- If the hardware is changed, more work related to the system reconstruction has to be done by the developers. Many facilities are unavailable, such as the API and device driver of a new hardware addition.
- The design and development platforms for the hardware and its software accessories (for example APIs, device drivers, and integration and porting to the system) are not as complete as the ones for the software. Unlike software development tools that can compile programs automatically, the compile, build and link tools are separated and intertwined with a lot of manual work. The infrastructure for the hardware development is not as developed and popular as that for the software.
- Different hardware devices, especially from different suppliers, are not compatible with each other whereas software code programmed with one language, such as C or C++ language, can merge more easily with other parts of the software system. The reuse of modules in hardware design and implementation is lower than for software codes, especially regarding different devices from different suppliers.

Despite these constraints, the hardware solution has some features that its software counterpart does not.

- The hardware accelerates the operations more effectively and directly than the software. The hardware can be sped up by lowering the number of system clocks by optimising the hardware design. Since the hardware is the final actuator in a complete ES, its acceleration can make the software running on it more quickly.
- Parallel processing is achieved more easily and reasonably with hardware than via software. The multiple branches of data flow in the hardware implementation are inherently parallel.
- The pipeline of a sequence of dependent operations or instructions can be implemented more straightforwardly with hardware modules than via the software code. It can also make a substantial contribution to the speeding-up of the system.

3.1.2.4 Current Conditions of ES Design

No complete and systematic tool for ES design is available yet. Since ESs consist of hardware and software, it is difficult to integrate a hardware design tool into a software environment, vice versa, in a short period of time, especially as researchers from two different backgrounds are involved. Researchers involved in software study and development follow the rules of software engineering in computer science whereas researchers in hardware design and development follow the standards of electronics engineering. Their design tools are developed for different goals. It is difficult to convince people in a short while that combining these two together could be effective. After all, it needs a lot of more work to build up a complete design environment for ESs.

The good thing is that there are already design tools that can be used in software or hardware separately. The Altera Embedded System Development Kit (Altera 2008a) provides a good prototype for FPGA-based ES design environment even though there is still a lot of work to do. For designers who want to take an ES design as a coherent and systemic process, the challenge is the energy and time required to learn and become familiar with two different design styles for software and hardware. It is also necessary for these designers to figure out how to make two design tools collaborate well to meet a unified ES design goal.

There are still differences between ES design and single software or hardware design. Single software has a closer relationship with the computing-intense processes. The input/output-intense processes have to access the input/output peripherals of a system, which are usually set up by operating systems and standard libraries. Software designers do not necessarily think about it much. The high abstraction in computer science does not work for ES. Such high abstraction is generated by significant separation between computation and devices, and central enabling management of resources. It hides the system resources from applications. Since ESs are often device-dependent, application-oriented, and operation-targeted, ES designers must know how to

communicate with devices and harness them well.

On the other hand, single hardware designers may consider how a hardware unit works and which hardware units should be used to set up a target system. They do not think much about how their implementations are to be used in an application environment. It is usually the responsibility of the application developers of a hardware system to establish the facilities to control and harness the hardware, and make a smooth connection between the hardware and software over and above the hardware system. It needs considerable effort to do this.

From the above discussion, it can be seen that hardware design goes in a totally different direction from software design. One cannot simply move from software to hardware design, or the reverse, to form an ES design. In other words, ES design has to fill all the gaps and establish a complete and systematic methodology to integrate control theory with fundamental prototypes in hardware and software design.

3.1.3 Challenges for ES Design

The challenges for ES design can be summarised as follows.

- There is a big gap between the two model-based methods in computer science and electronics engineering. Different models constructed by different tools meet different system requirements and support different design processes. Apart from these requirements and processes there is a difference between the nature of software and hardware, and a distinction between the digital and analogue domains. Objects processed in the former are digital and discrete whereas signals sampled by the latter are analogue and continuous to time. It is necessary to look for an effective method to connect them to make them work together to server applications.
- ES design has a wider scope of applications than software programming when roughly viewed in the diversity of applications. It has a smaller pool of human resources and more distributed force than software design, as shown in Figure 3.1. Two factors contribute to this layout. One is the stronger specialty of ES technologies than software programming. For this reason, the education investment of a good ES designer is greater than that of a good software programmer. A hardware development platform is usually more expensive than a software one. This makes software programming more attractive to raw recruits than hardware development. The other factor is the greater diversity of ES applications than those of general-purpose computers. The wider diversity of applications divides the small pool of researchers and developers in ESs into many groups and makes each group even smaller. The narrow strip of ES, as shown in

Figure 3.1, illustrates the division effect of the application diversity. These two factors lead to more researchers maintaining the design environment for software programming than those for ESs. Furthermore, they result in the better design environment for software programming than that for ESs.

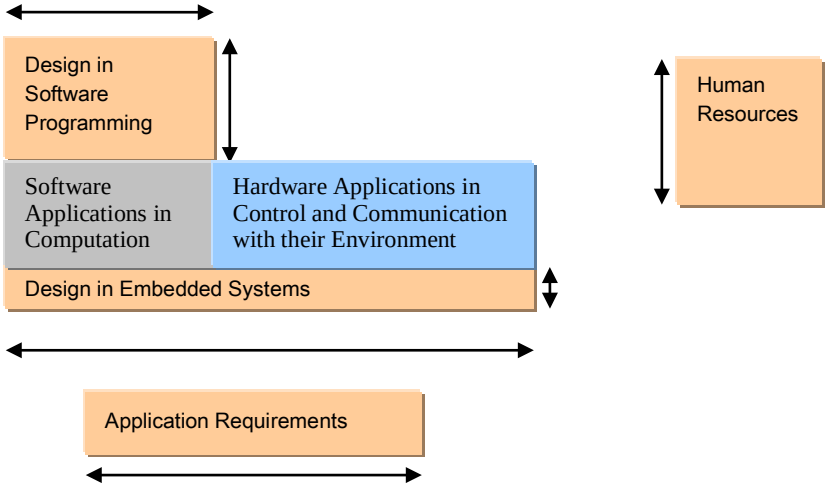


Figure 3.1 Comparisons of Application Requirements and Human Resources between ES and Software Programming Designs. Horizontal Widths Represent Application Requirements; Vertical Heights Represent Human Resources for Research and Development. The Wider Width of the Rectangle of Design in ESs Shows the Wider Application Diversity of ESs. The Thinner Height of the Rectangle of Design in ESs Shows the Smaller Separate Human Resources in ESs.

- Another problem related to human resources for ESs is the lack of researchers and engineers who are expert in both two fields, software programming in computer science and technology and hardware engineering in electronics engineering. These two fields have different theories and research methods to support, which have been discussed in Section 3.1.2.4. This results in the separate researches in two fields.
- These two domains, software programming and hardware engineering, are equipped with different mathematics systems. In one system, there are specified notations for operands, operations, and restrictions. As a system, it is expected to be complete or self-contained. Inside their own domain or system, operations and operands can be transformed and executed in a right way. Outside their domain, they cannot be guaranteed to behave in a correct way. ES design includes the two in order to cope with any system that has to consist of both hardware and software. A mathematics system designated for ESs has to be established, just like computer science has discrete mathematics. It can fully express and analyse the ES domain.

Since software programming concerns itself with discrete and digital data and hardware engineering is required to transform continuous analogue signals into discrete and digital data, it is reasonable to suggest a new mathematics system combining or comprising the digital and analogue domains.

- ES designs need a sufficient library of building blocks to support wide and changeable modelling. This library takes time to accumulate.
- A transforming means is required to compile and (or) synthesise a constructed model into a style which the underlying systems can accept and execute. The means cannot be single but is complex. It can be interwoven with many user manual interactions.

3.1.4 Prospective Principles for ES Design

Some prospective principles for ES design can be summarised as follows.

3.1.4.1 A Rational Combination of Functional and Computational Models

Conventional functional and computational models are also applicable in ES design. They have different benefits, meet different system requirements, and are suitable for different targets.

Functional models are adept in the situations that involve in concurrency tasks with quantitative restriction conditions. They are not good at dealing with sophistic mathematics algorithms and uncertain tasks. They are usually used in hardware design, scheduling and dispatching tasks, and performance evaluation. Therefore, functional models can be used in ESs.

The System Identification Toolkit of National Instruments Corporation is an example of a functional model tool (National Instruments 2007). The toolkit consists of a library of VIs (virtual instruments) and an assistant for developing models of a system. Because of its graphical programming language, one can use the toolkit to construct control models for mechanical engineering, biology, and economics applications. The model can reflect the behavior of a certain dynamic system. Even though it is a software model tool, the models constructed with System Identification Toolkit can be analysed and validated, even used as a controller to hardware.

Computational models are inherently abstract and hierarchical. They are good at dealing with complicated mathematics problems, and partial and changeable tasks. They are not adept at tackling concurrency and quantitative restriction conditions between tasks. Computational models have been successfully used in programs, operating systems, state machines, time sharing systems, distributed systems, and situations where tasks have to be tackled dynamically. The compilers of many high-level languages, such as

GCC (GNU Compiler Collection), can be computational model design tools. Computational models can be applied in ES design to some extent. At a high abstraction level, computational models are not suitable for ESs because of nondeterministic sequential processing. The hardware part of ESs has to retain the ability of parallelism, pipelines and precise cooperative constrains between tasks. Otherwise, the hardware design cannot be accurately mapped to the components of electronic circuits.

The compilers and synthesis tools have limitations. For example, they can produce inefficient codes from a constructed model. The transformation semantics may be misunderstood or its library may not include some functions or computations that designers expect; for example, floating point computation is not always supported. Designers need to change the way they construct a special model and include computing insights to make the compiler or synthesis tools transform correctly. Therefore, the intelligence of designers can improve the effect of compiler and synthesis tools.

For the tools of functional or computational models, the basic idea is to provide an environment where designers work with a designated human-friendly language; the language can be a high-level language or a graphical language. The environment helps to transform or translate a model constructed by a designer into a set of codes that the underlying system can identify. This underlying system can be software, such as an operating system or hardware abstract layer, or hardware, such as FPGA devices or VLSI systems.

In any model-based design method, at the early stage of a design, system components in a modelling language cannot produce a design that is executable or interactive, or can be implemented successfully in a hardware device. Multi-times of model construction, analysis, compilation (or synthesising) and improvement are required for the redesign of systems. Verifications and adjustments of the hardware systems are also required in the engineering context. Therefore, it is a trial-and-error process.

To meet the connection goal, the consistent aspects of the two domains have to be identified for decisions on how to do any ES design in a convenient way that can be promoted in ES world.

3.1.4.2 A Promising Design Platform for Generic ESs

As two models, functional and computational models, exist, a new platform that can include these two models is required. Figure 3.2 shows the structure of the new platform. This structure gives a partitioned solution. As functional and computational models have different application targets, support different design processes, and satisfy different system requirements, it is unrealistic to mix them directly together. Any application target, like ES itself, consists of software and hardware. It involves software programming and hardware construction. Consequently, the system design should start by decoupling the

top coupled models. The top coupled models are obtained from the system requirement analysis. Through decoupling, the functional sub-models and computational sub-models can be obtained. With coupling, an entire ES can be composed with all the sub-models. During construction, the functional sub-models can be built up with building blocks for hardware and software and the computational sub-models can be constructed with building blocks for software. Two pools of building blocks, one for software and the other for hardware, can be accumulated, based on the standard software libraries and hardware component engineering archives. Intermediate modification is allowed. After verification, executable and configure codes for a specified target machine can be generated ready for application. The design can also be optimised to cover the entire ES for the designated applications.

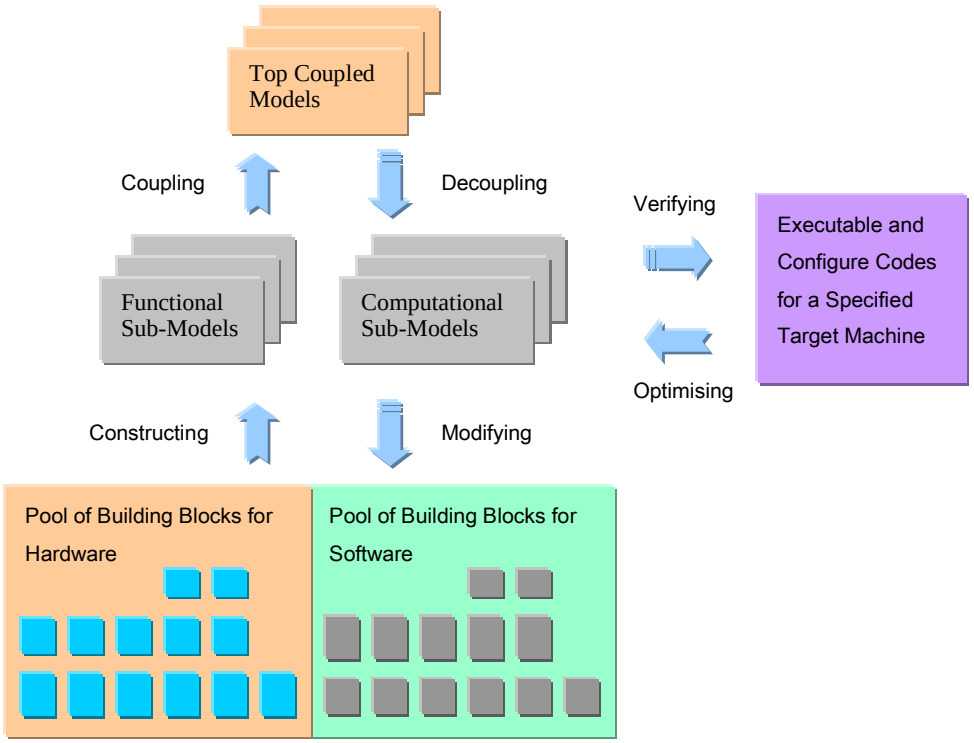


Figure 3.2 Structure of New Platform for ES Design

It is sensible to integrate existing software and hardware design tools into a complete design kit or platform in order to implement a system that includes hardware and software. It is not a simple task, and it will need a lot of more work to make such a system practicable and usable.

To accompany this structure, a hierarchy of design process for ESs has to be defined. Since software is always executed on the top of hardware, it is natural to do the software

implementation after the hardware accomplishment. Figure 3.3 shows the hierarchical design process for ESs. It is a process for a bottom-up design but it can be accomplished in parallel. Hardware construction is expected to be completed before software verification.

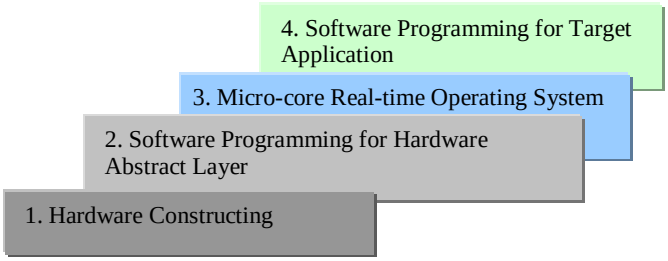


Figure 3.3 Hierarchy of Design Process for ESs

The second layer is the software that is closest to the hardware system. Since any unit of a hardware system must have a section of codes or device driver that can be used by the top software to control or communicate with the hardware unit, the hardware abstract layer must be programmed by designers or be provided by a third-party organisation or suppliers.

The third layer is not mandatory. As regards computational, multi-task, and sharing-time applications, a micro-core real-time operating system is very helpful to manage the tasks and resources. A micro-core operating system is available from some open-source software resources. It is usually necessary to customise it to fit in the target applications. One reason for this is the limited memory space. The other reason is that some required functions may not be available in an open-source operating system. For simple control systems without an operating system, a complete hardware abstract layer is sufficient to handle the issues involved in communicating with the hardware system. Without an operating system, the response to the environment may be even faster and the system performance can be enhanced.

The top level of the hierarchy is the software programming for the specific target applications. For some systems, it may contain a very small amount of software. For others, it may consist of many functions, mechanisms and algorithms. It depends on the special applications that the ESs target.

In fact, even though the design process can apply model-based design methods and no matter how powerful an integrated design kit is, the final stage will be carried out in a bottom-up way because the software has to be executed or verified on a hardware platform. In other words, the hardware implementation should be at least one stage before the software implementation. The layers of design process can be pipelined, which means that some unrelated tasks in different layers can be done in parallel. As there are inherent

links between the layers, the design of ESs has to follow the order of this hierarchy loosely in order to avoid wasting the energy and time of developers on multiple re-designs or re-implementations.

3.1.4.3 A Rational Division of Functional and Non-functional Requirements

In ES design, there are functional requirements for the services and functions that systems should provide. There are also other requirements for performance and robustness, such as the response speed and the recovery ability from the worse case.

The functional requirements are easily mapped into building blocks, but non-functional requirements are not. It depends on the test and verification of the implementation of a design model and even the application of the implementation whether or not the non-functional requirements are met. Some non-functional requirements, such as performance, can be tested by a number of tests, but others, such as resistance to failures and attacks, cannot. The non-functional requirements can influence all design decisions in system construction. For example, to enhance the safety of system, redundancy technology must be used; to raise the security level of system data, some protection measures should be applied in data access and delivery. Therefore, rational anticipation for these non-functional requirements is expected.

3.2 Reasons for Choosing an ES as this Project Platform

There are several reasons for choosing an ES as the platform for this research.

- **Flexibility for construction.** ESs equip the designers with the ability to construct the system from the hardware. In this way, designers can customise the hardware units according to their requirements. For developers of performance-intense application, ESs are more designer-friendly than other platforms. They provide designers and developers with more options for processors, memories, devices and peripherals to enhance the whole system performance including operation speed, power requirement, and system size. The flexibility of ESs can offer more options for graphics applications of this project than the general-purpose computers.
- **Uncertainties.** ESs are an emerging discipline of science and technology, and there are likely to be many possibilities for their use. Although they are becoming more and more prevalent, their new environments are not well known yet, and it may be necessary to make an estimate and then experiment with a trial-and-error approach. The difference between the expected and worst-case situations may be expected. These uncertainties have risks as well as opportunities for exploration. Although GPUs are a popular solution to the graphics speed-up for the

general-purpose computers, the ES may present a new field to be explored for a solution to the graphics units for other applications, such as small handheld gadgets and in-field equipment.

- **Changeable structures.** ESs have been adapted and adopted in many applications. They offer changeable structures by using different unit options and parallel processing schemes to suit the application environment. The changeable structures can provide a systemic strategy to solve the graphics application issues.
- **Room for further development.** Since FPGA and VLSI technologies progress rapidly, more sophisticated components and architectures can be introduced in ES design: for example, pipelines, multi-core CPUs with caches, and speculative execution. These can lead to further changes and development in terms of the ES. The pipelines can readily be applied in graphics for speed-up.

These are part of reasons why the ES is adopted in this project even though most of other researchers in computer graphics are trying to find solutions in GPUs.

The ES is mostly thought as a device embedded into a large system. It is involved frequently in the hardware. The word ‘embedded’ can describe not only the hardware but also the software. Also, a device (like a bank of memory, peripheral, or processor) or a section of codes (such as a function or routine) can be embedded or integrated seamlessly into a target system. It means that ‘embedded’ has connotations of adding to and subtracting from, tailoring, and customising a system. That is, if the infrastructure is available, the application developer can construct an ES with any hardware device module and software element and integrate it into a larger system.

In terms of hardware development, there are several technological difficulties, including the construction of the API and hardware abstract layer of device drivers, and porting and integration of a new additive unit in the system. Furthermore, the fact that there is no existent device that can fit into a specified algorithm needed by the designer can be a serious obstacle. An FPGA can help to accomplish the design and implementation of a hardware module for a specified algorithm. Some advanced FPGA chips have integrated DSPs (Digital Signal Processing) into their elements. For these reasons, an FPGA-based ES is an ideal option for this project that includes a special graphics algorithm, PAMA, to be supported.

Another difficulty is how to link all the design and development processes for hardware and software together and make them function together well. It is the required infrastructure setup for the design and development of an ES with the FPGA support.

The Nios II Embedded Design Suite (EDS) of Altera Corporation (Altera 2008a) gives a comprehensive solution to an ES with FPGA support. Several years of using the FPGA

development platform, Qurtues II, and FPGA device products, have shown that the Altera Corporation's Nios II EDS accords with the research demands of this project. It provides the infrastructure for the research and development of an ES with FPGA support and the flexibility and freedom to explore new research approaches. After full consideration, it was decided to use the FPGA-based ES as the main research platform for this project.

This research has been done on the Nios II EDS and Altera Embedded Systems Development Kit, Cyclone III Edition (Altera 2008a). Figure 3.4 shows a photo of the board. The details will be introduced in the next section (3.3).

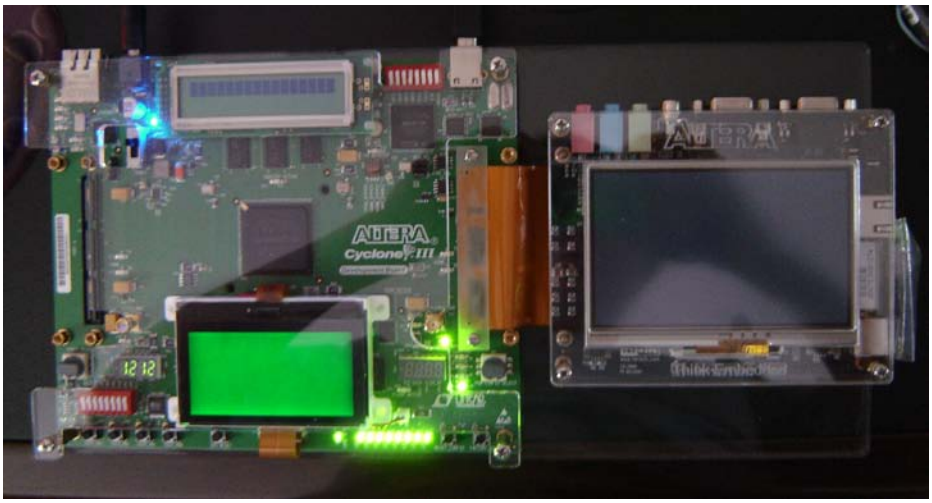


Figure 3.4 Altera Cyclone III ESs Development Board

3.3 Environment Structure of this Research

Since the research goal is to enhance the whole system performance in a hybrid way, both hardware and software have to be involved at the very beginning of system design. The environment structure has to be open to development in both hardware and software, which is one of the prerequisites for the research platform chosen.

The environment structure for this research and its expansion is shown in Figure 3.5. Different from the common computer system architecture, this structure has a clearer boundary between low-level hardware services and high-level applications, and both hardware and software are open to customisation. It is helpful that new modules can be freely added to the target ES. The hardware design and implementation can influence the software. Inversely, the set of applications decide the requirements that may be used to customise the components of the hardware system. Therefore, a top-down design and bottom-up implementation are employed. Moreover, as the design and development are processed in a sequence of relevant and integrated platforms (Altera 2008a), some algorithm coded by C language for an application can be transformed into a hardware part

and merged into the target embedded hardware system.

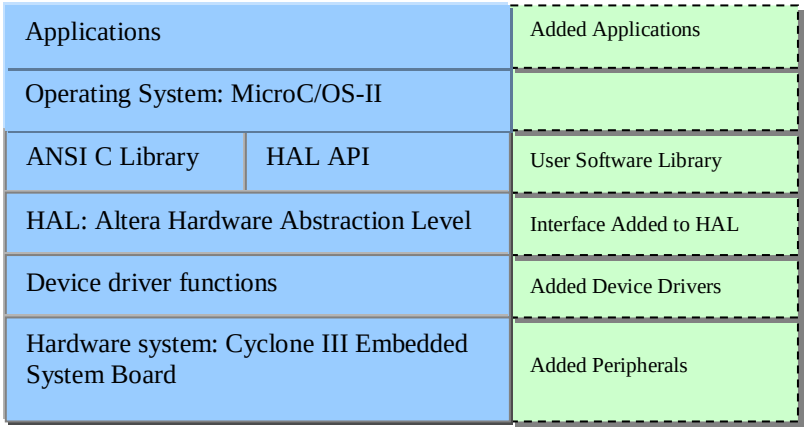


Figure 3.5 Environment Structure of the Research and its Expansion

This structure consists of six levels, from bottom to top, including the embedded hardware system, device driver functions, Altera Hardware Abstraction Level (HAL), ANSI (American National Standards Institute) C Library and HAL API, operating system, and applications.

It is also evident in the dotted-line blocks at the right column of Figure 3.5 that the structure can be expanded. If needed, some hardware peripherals can be added to the ESs no matter whether they are the devices on the board or the configurable parts in the FPGA. The relevant device drivers have to be added to the level of device driver functions. During HAL regeneration, the functions are merged into the original HAL and equipped with the HAL API. Whether the new HAL API is with or without the software addition in the HAL API level depends on whether or not the newly added devices belong to one of the general peripheral classes. For some new applications, new hardware peripherals may not be needed but the user software library that is not available in the Altera embedded systems may be required. The operating system remains unchanged. On the top level, more applications can be supported.

3.3.1 Embedded Hardware System

The embedded hardware system is the Altera Embedded Systems Development Kit (ESDK) (Altera 2008a). The ESDK consists of three parts: the Cyclone III 3C120 FPGA basic board, LCD Multimedia High Speed Mezzanine card (HSMC), and Multi-purpose HSMC for debugging and developing software via the USB and SD card. In this research, all the three parts are used during development; when the graphics applications are executed, only the first two parts are used because the HSMC is meant for debugging and developing software.

The FPGA of Altera Cyclone III EP3C120F780 provides hardware design support. With the SOPC (system-on-a-programmable-chip) facilities provided by Altera Corporation, a soft microprocessor (Nios II) can be incorporated with general peripherals and standard interfaces for most embedded applications. Along with another development tool of the Altera Corporation, the Nios II C-to-Hardware Acceleration (C2H) compiler, the SOPC can also be used to add a new algorithm-specified hardware module to the target ES, which has been designed to include all the regular parts, such as processor (Nios II), memory, and peripherals (for example, LCD controller, video processing pipeline, and LCD touch panel controller). The design details will be introduced in Chapter 4.

The C2H compiler can transform an application algorithm programmed with C language and executed by the Nios II processor into hardware in FPGA if the algorithm software is programmed according to C2H rules. The mapping hardware can improve the execution performance by using parallelism and pipelines. The C2H compiler is able to make the mapping hardware of independent statements operate in parallel and create the pipeline logic to lower the memory access latency. The process is the converse for bottom-up-to-top implementation. It shows the interaction and feedback between the design and implementation in order to enhance the whole system performance. It also reveals that the target embedded hardware system can be customised.

The Nios II soft processor is 113 DMIPS (Dhrystone Million Instructions Per Second) at 100 MHz, with two 32-Kbyte caches for data and instruction, respectively, which uses around 1500 of 119000 total logic elements in the Cyclone III 3C120 FPGA. The memory resources on the board include two banks of 64-Mbyte DDR2 SDRAM memory, 64-Mbyte common flash interface (CFI) flash memory, and 1-Gbit SD card memory. The key peripherals for the research are the LCD controller of the *800 X 480* pixel LCD colour screen display, which are also integrated into the ESDK. A PLL (phase-locked loop) receives the *50* MHz on-board oscillator input as its clock source and outputs two clocks: *100* MHz clock for the CPU and *60* MHz clock for the slow peripherals. A performance counter helps to analyse the system performance. The response unit of four buttons is used for the user interaction. Figure 3.6 shows the block diagram of the ES customised for this research.

As the Altera ES provides the software and hardware support of a Nios II processor and a rich set of peripherals, it saves the developers designing and developing the whole ES from scratch and makes them just focus on areas of interest and add them to the target ES.

3.3.2 Device Driver Functions

For this research, the graphics application is the goal. Therefore, the device drivers and functions include an LCD controller, video pipeline, response unit of four buttons, and

algorithm-specified module.

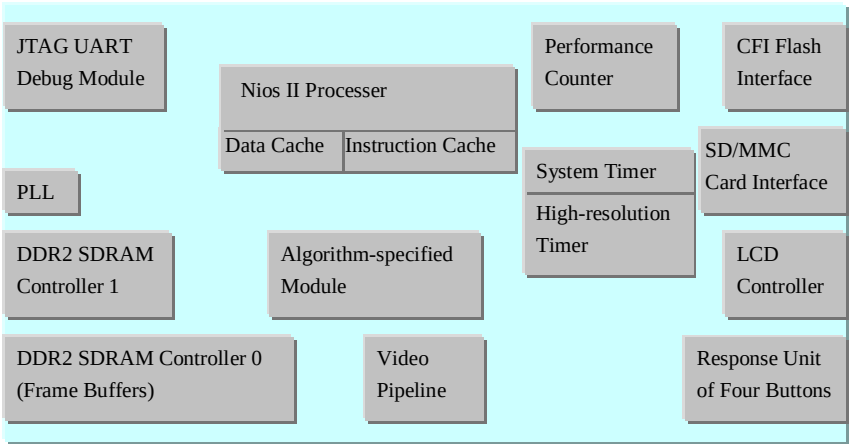


Figure 3.6 Block Diagram of the ES Customised for the Research

The LCD controller software module consists of two parts: an LCD controller driver and the LCD controller software API. The LCD controller driver provides a set of low-level functions for communicating with the LCD module registers. During the system configuration, the LCD module registers have to be configured with the LCD controller software API.

The graphics pipeline is constructed from the resources provided with the ESDK. The algorithm-specified module is created for a novelty graphics algorithm at a high level in the graphics pipeline. Its output data are sent to the frame buffers. Then the data are transferred to the video pipeline. The video pipeline processes the data to meet the pixel and timing requirements of the LCD display stream. Since the frame data in the fame buffer that resides in one of DDR2 SDRAM memory banks do not match the LCD video stream, the video pipeline provided by the ESDK reads the frame buffer, and produces and synchronises the pixel data to match the video stream of the LCD device. Finally, the graphics is displayed on the LCD screen. The process is shown in Figure 3.7.

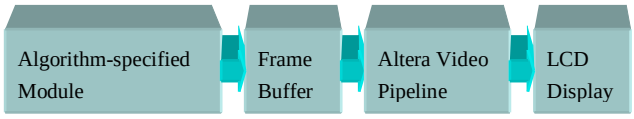


Figure 3.7 Graphics Pipeline in FPGA-based ES

The graphics pipeline software module includes driver functions and API. The driver functions and API can be used to initialise the device and manipulate frame buffers.

The response unit of four buttons receives signals when the user presses one of buttons and transforms them into interruption signals for Nios II processor.

3.3.3 Hardware Abstraction Level

The Altera HAL integrates all the device drivers into the Nios II processor system. The HAL can provide a basic runtime environment for the ES even without an operating system. When the system power is switched on, the HAL performs the system and device initialisation. It has a consistent interface to the device drivers. This interface provides a channel for procedures to control and manipulate the hardware devices. The HAL can be customised as well.

As the HAL defines a consistent interface to the device drivers, the drivers for new peripherals are explicit and concise and can be programmed and integrated into the target ES. For general peripherals, the HAL provides a device model for each class of devices that defines a group of procedures for managing the class. The HAL supports general peripherals including character mode devices, flash memory devices, file subsystems, timer devices, DMA devices and Ethernet devices. For some hardware that does not belong to any of the above classes, its deriver should have a header file and a group of dedicated procedures to access it.

The clear differentiation between device drivers and application software separated by the HAL makes high-level applications reusable when the hardware system changes.

To expand the hardware configuration, during the embedded hardware system generation, the Altera Nios II BSP (board support package) can be configured at the top level of the hardware system by means of the BSP settings. These settings are usually the system-dependent and influence the system performance and functions. These settings include those for the standard input and output devices, system timers, memory regions and section mapping for the linker, and the boot loader enable. The details are as follows.

- **Settings for standard input and output devices.** They can choose and make one of the character mode devices connect to the Nios II processor in the target hardware system.
- **Setting system timers**. They can choose and make one of the system timers connected to the Nios II processor in the target hardware system.
- **Setting default memory region**. The first choice of the default memory region is the largest volatile memory. The second choice is the largest non-volatile memory. But a specific memory region can also be set that is available in the system with the memory region settings. The default memory section mappings consist of .entry, .exceptions, .text, .rodata, .rwdata, .bss, .heap, and .stack, which are the section names of the boot and reset entry, exception service entry, instruction section, and data sections for different functions, respectively.

- **Setting the boot loader**. When the boot loader enable is set, the instruction section mapping and the Nios II reset port will be examined. As regards the system reset function, it will also engender that the boot loader copies the data sections.

3.3.4 ANSI C Library and HAL API

With the HAL, the ANSI C standard library is integrated into the Altera embedded software system, and its library functions are available for the high-level applications. The HAL defines the interface for communication with the embedded hardware system, called HAL API. The hardware access macros of HAL API cooperate with the ANSI C Library functions to allow the applications to access the devices and files. Therefore, when accessing the system resources, applications may pass through the ANSI C library or the HAL API.

3.3.4.1 HAL System-Specific Settings

The ANSI C library, because of its portability, defines explicitly the data width for data types. The HAL is dependent on the embedded hardware system, especially the embedded processor, Nios II processor. It has to give the data width for different data types clearly.

The HAL also provides a basic system interface and environment for executing the application programs without an operating system. This system interface consists of some UNIX-style functions that perform system settings and file I/O operations.

3.3.4.2 HAL Device Management Strategy

The HAL manages the devices in the same way as the UNIX operating system. At the system generation time, the HAL registers devices as nodes in the HAL file system (identifies their path names with the prefix /dev/) and builds the connection between each device and its access functions. Generally, a file descriptor is associated to a device's name when the device is accessed with ANSI C file operation – fopen(). If data are sent to or received from the device, the ANSI C file I/O functions – fread() and fwrite() – can be used.

3.3.4.3 HAL Character Mode Devices

For a character mode device, with the file descriptor associated with the device, a program can send characters to or receive from the character mode device with fread() or fwrite().

The HAL also supports standard input (stdin), standard output (stdout) and standard error (stderr). It gives another channel to access the I/O devices without passing through file descriptors.

3.3.4.4 HAL File System

Like the UNIX file system, the HAL file system mounts a file subsystem on a given mount point, which is a directory under /mnt/ directory. Therefore, it is convenient to access the directory by using the ANSI C file operations.

3.3.4.5 HAL Timer Devices

HAL timer devices count the elapsed number of system clocks and generate interrupt requests periodically. As regards time devices, the HAL have two types of drivers: a system clock driver and a timestamp driver. The former supports alarms and can be used in a thread scheduler. The latter provides high-resolution timing procedures to launch the counter start and return the current value of the timestamp counter, which can be used in the program performance analysis.

3.3.4.6 HAL Flash Devices

As a non-volatile memory, flash memories have a physical nature. A flash memory is often divided into blocks. To erase a block of flash memory one needs to set their values to ones rather than zeros. An entire block must be set to one at the same time. A single address in a block of flash memory cannot be erased individually. To write one bit in a flash memory one has to change it from one to zero. If a change of any bit from zero to one is expected, the entire block where the bit is located must be erased totally.

Generally, if just reading the flash memory, programs can do the reading in the same way as a simple memory without calling the special HAL API. If writing the flash memory, programs have to perform several operations, which require the procedures of the flash device model in the HAL API, such as alt_flash_open_dev(), alt_flash_read_dev(), alt_flash_write_dev(), and alt_flash_close_dev().

The HAL API for the flash device model provides two ways to operate the flash. One is simple flash access; the other is fine-grained flash access. The former writes a buffer to the flash memory at the block unit. If the buffer is less than a full block, the whole block where the new data are written is erased first and then the new data are written to the designated addresses. If the previous data on the addresses around the new data are useful, the writing operation of the simple flash access will cause data corruption.

To solve this problem, the latter way can be used. It can change the data at the designated addresses of flash. It provides three more procedures that can perform operations on the written block: alt_get_flash_info(), alt_erase_flash_block() and alt_write_flash_block(). If the buffer is less than a full block and new data writing is expected without change of the surrounding data, some subtle operations must be performed. First, the whole block where the new data will be written is read to a buffer. Second, the new data are written to the buffer, which leaves the surrounding data

unchanged. Third, the whole block is erased. Fourth, the buffer is written to the block.

3.3.4.7 Video Pipeline

The video pipeline software API provides a set of procedures which control the video pipeline and manage the graphical frame buffers. These procedures include the opening and initialisation of the video pipeline, the manipulation of frame buffers, and the closing of video pipeline.

The opening and initialisation procedure, which is alt_video_display_init(), opens and initialises the video pipeline for graphics display. Its tasks include allocating memory space for all the frame buffers and their SGDMA (Scatter-Gather Direct Memory Access) descriptors and initialising the descriptors for each frame buffer. These descriptors can control the pixel data in each frame buffer and send them into the video pipeline automatically without the Nios II processor's intervention. The memory access of frame buffers and their descriptors can employ absolute or relative address, depending on the parameter setting in the head file of the video pipeline API.

Relative addressing is usually adopted in the heap memory. Its advantage is that the heap procedure in C runtime library can provide and manage the required memory for all the frame buffers and their descriptors, and save designers and developers from having to consider it. Its limitation is that contention can arise when both the Nios II processor and SGDMA have to access to the heap memory.

After the allocation, all the pixels in the frame buffers are replaced with the default black colour or other specified colour. The initialisation procedure can also create a structure that can keep track of the information of all the frame buffers, and return a pointer to this structure that can be used to access the frame buffers later.

The manipulation procedures of frame buffers are used to access the frame buffers. There are two or more frame buffers for the graphics pipeline. One is designated for writing a new frame by the graphics application, as shown in Figure 3.7; the others are for displaying on the LCD device screen. After one frame is written, the frame buffers can be swapped. Therefore, manipulation procedures are useful for handling these issues. For example, alt_video_display_buffer_is_available() is used to acquire a free frame buffer for writing; alt_video_display_register_written_buffer() is used to display the frame buffer on the LCD screen. All the manipulation procedures use the structure pointer that the initialisation procedure returns to require, access, or display the frame buffer.

When the video pipeline is not used any more, the closing procedure of video pipeline (alt_video_display_close()) can be used to stop the video pipeline, close it, and release the memory resource to the system.

3.3.4.8 LCD Device

The LCD software API provides an initialisation procedure and a set of low-level procedures for communicating with the LCD device registers.

The initialisation procedure of the LCD device can be used to configure it. This procedure sets the default parameters for the gamma curve and the positive polarity voltage. It also creates a structure for the LCD device and returns a pointer to the structure. The LCD device is configured with the parameters that specify the resolution of the LCD screen.

The LCD device, as a general purpose I/O peripheral on the FPGA, has a simple three-wire interface. Its configuration registers of enable, clock and data signals can be communicated through this interface. Low-level procedures are used to read and write these registers.

3.3.4.9 Response Unite to Four Buttons

Since there is no keyboard or mouse available on this board, the response unit of four buttons are adopted by this project to implement the response to the inputs of the user interaction for surface editing at high-lever application.

Four user-defined buttons are available on the Cyclone III embedded system development board, as shown in Figure 3.8. They can be also seen at the left-bottom corner in Figure 3.4. They can be configured as four general purpose I/O peripherals. The input ports can catch the rising edge of signals when the user presses one of the buttons and generate an interrupt signal for the Nios II processor.

Figure 3.8 Four User-Defined Buttons on Altera Cyclone III ESs Development Board

The initialisation procedure (init_button_pio()) is used to register the rising edge of signals from each of four I/O ports as an interrupt signal for the Nios II processor by using the procedure of alt_irq_register().

The button interrupt handler (handel_button_interrupts()) can store the value in the button's edge capture register when one of the buttons is pressed by the user. An interrupt service routine is programmed in the user's application to accomplish functions that the user defines. In this project, the user's defined functions are the operations for surface modelling and editing algorithm, PAMA.

3.3.5 Operating System

The operating system is MicroC/OS-II of Micrium Inc., created by Jean J. Labrosse, which is open-source, portable, and scalable. Altera has ported the MicroC/OS-II to the Nios II ESs. In the environment structure of the research, the MicroC/OS-II is a layer on top of the Altera HAL and shares a common structure. It is a multi-threaded runtime environment. It implements a simple RTOS (real-time operating system) scheduler. Its directory structure is a superset of the HAL BSP (board support package) directory structure. With the MicroC/OS-II above the HAL, application programs can be reused when the embedded hardware is changed, and portable to different Nios II embedded hardware systems.

For the HAL, the MicroC/OS software procedures are similar as the device drivers for the Nios II processor. As the HAL is a runtime single-threaded environment, the MicroC/OS scheduler is used inter alia to dispatch the running time of the processor among several tasks to create the multi-threaded environment. The HAL API of hardware devices can also be extended to cover the multi-threaded environment of the MicroC/OS-II.

Altera provides a group of operating-system-independent macros that access operating system facilities. During the system generation, these macros can make a switch between the single-threaded HAL environment and the multi-threaded MicroC/OS-II environment in order to make a decision whether or not using the operating system.

3.3.6 Applications

In this project, the application is the implementation of the surface modelling and editing algorithm, PAMA, with the Mesa-OpenGL support. The PAMA will be discussed in Chapter 7. The application results of the PAMA algorithm on an FPGA-based ES will be presented in Chapter 8. The Mesa-OpenGL implementation for the FPGA-based ES will be discussed in Chapter 5.

3.4 Chapter Summary

Since the general-purpose computer cannot meet the need for combined hardware and software to solve graphics pipeline problems, ESs are chosen as the platform for this project. This chapter has first discussed the ffeatures and principles of ES design. Then the reasons for choosing an ES as the platform for this project are analysed scrupulously. Finally, the environment structure is detailed for the research.

It is shown that ESs equip designers with options for processors, memories, devices and peripherals to enhance overall system performance including operation speed, power requirement, and system size. The novel approach of hybrid hardware and software

design method allows designers to consider the strengths of both hardware and software in an integrated ES. The change from stand-alone hardware and software in conventional system design has great potential to enhance overall system performance. The consequent challenge posed is detailed in the next chapters.

Chapter 4 FPGA-based Embedded Hardware System for Graphics Applications

Conventional building of ESs needs three groups of engineers and developers: hardware engineers, device driver developers, and application developers. There is no such team available for this project. One of the important tasks for this project is to find a feasible strategy for one or few developers to establish an ES for the graphics applications in a whole process by combining both hardware and software designs. It also needs to validate whether this hybrid way is viable for a research and development project. On the other hand, as mentioned in Chapter 3, ES development involves both hardware and software implementations. The expertise and skills required for hardware design and implementations are different from those for software. The knowledge and skills for software programming is not sufficient for a hybrid system construction and more effort and time should be spent in studying the details of hardware implementation. The benefit is that it is known how to make all of them function together well and realise the expected goal under the prevailing conditions.

This chapter discusses the hardware system construction with FPGA. Before that, however, the traditional ES development process is introduced.

4.1 Traditional ES Development

In traditional ES development, hardware engineers, device driver developers, and application developers do their tasks relatively independently. The hardware engineers initially need to know the application goal and take account of the system performance and power and size limitations. They choose the devices required to build the target ES and try to strike a balance between the high performance of computing speed and response ability and the low cost of power and hardware resources.

To create the device drivers to control and manipulate the devices properly, the device driver developers should understand the physical characters of hardware resources that

hardware engineers choose during the hardware system's building. They should also know the standard calibrations of application programming interfaces that the application developers will use to program the applications to interact with the hardware resources.

The application developers program the designated applications according to the standard API and a set of functions that the device driver developers provide. The top applications are constrained by the supply of hardware resources and the integrity of the device drivers and API. The two previous steps can heavily influence the development flexibility of applications. Successful implementation for an application goal largely relies on adequate construction of the embedded hardware system and device drivers.

The separate engineering process blocks direct communication between these three groups of developers and engineers. Since the initial requirement analysis of a goal application can be incomplete and shallow, the embedded hardware system constructed by the hardware engineers may not meet all the requirements of the targeted application. Issues may occur during or after the application development. In addition, in academic researches and new product developments, the requirements of an application can be modified when the research is deepened. These issues require the redesign of the embedded hardware system. In the separate engineering process, it means extra cost as the application developers have to ask the hardware engineers to modify the embedded hardware system. The modified hardware system can result in a change in device drivers. This can also increase the cost. Also, inadequately understanding others' design intentions may degrade the implementation of the target application-specified ES.

This separate engineering process can give rise to another problem: hardware appears more fixed and less flexible than software. There are two main reasons for this problem. One is that a hardware unit or device is harder to be changed once used in the implementation than a section of software because of the differences between the compiling and verification processes for hardware and software. A change of hardware may result in the replacement of physical devices or units whereas a change of software is only the replacement of a section of codes. The other reason is that it is more difficult for an application developer to change a hardware unit or device if s/he thinks it should be replaced with a better one because changing the hardware has to be carried out by the hardware engineer who designed the hardware system.

FPGA-based ES development offers a solution to the above issues since all the tasks of hardware engineers, device driver developers, and application developers can be performed in an integrated design environment by one or a few individuals. The main benefit of FPGA-based ESs is to make most hardware devices available for designs. It can save a lot of work in terms of constructing a system, adding new devices to the system, modifying units of the system.

In fact, even though it needs more time and experience to make a change on a hardware design to meet the newest requirements than to make a change on a software design, the former has become achievable and feasible with FPGAs. The FPGA-based platform provides a design-friendly environment for updating designs to realise new academic ideas and meet new market requirements. The Altera ESDK (Altera 2008a) is a good example. With the utilities and tools that Altera Corporation provides, the development and building of application-specified ESs is a feasible undertaking for one or few people. Section 4.3 will detail the FPGA-base ES development.

4.2 FPGA Device Evolution and Applications

In recent years, the increasing density of chips has provided an opportunity for the development of complex high-performance ESs on FPGA devices. FPGAs have been considered an appropriate solution for many applications that are expected to give high performance at low cost.

4.2.1 FPGA Evolution History

In 1986, when the first commercial SRAM-based FPGA was developed by Xilinx Inc., the products of FPGA technology began to be put on the market (Awad 2009). Many manufacture companies were active in the FPGA field until the early 2000s, but after several acquisitions and mergers, only a few of them are left. Among the biggest ones are Altera, Actel, Lattice, Quicklogic and Xilinx. As the competition among companies is strong, these companies' FPGA products cover a wide range for applications and functional architectures.

In the 1990s, FPGAs were small devices with low computational throughput, simple internal structure, and few components, and could not meet the needs of complex computation and functional applications (Constantinides and Nicolici 2011, and Qasim et al 2009). Along with the sustained progress of VLSI technology, FPGA devices have developed into ones that are composed of multi-million gates and diverse logic components. Thanks to the architectural innovations, the FPGA device density has been improved. Many hardware units specified for some operations, such as multipliers and embedded memory blocks, have been gradually integrated in FPGA devices. In the newest generation of FPGA devices, all the complex blocks, such as multipliers, microprocessors, embedded memory, and fast routing matrices, can be integrated in one silicon die.

4.2.2 FPGA Features and Reprogrammable Technologies

As mentioned above, several manufacturers produce FPGA devices with their own

technologies and each company has several series of FPGA devices. For example, Altera has Cyclone (low-cost), Arria (midrange), and Stratix (high-end) series. Thus, the range of FPGAs is wide and varied. They provide different solutions for their FPGA devices. But the basic idea is the same.

4.2.2.1 FPGA Features

An FPGA device consists of a matrix of configurable logic blocks, configurable input/output (I/O) banks and an interconnect network that is reprogrammable to connect logic blocks and I/O blocks according to a design target. The configurations of logic blocks and interconnection network are dependent on memory cells. By changing the contents of memory cells, the FPGA can be made to fulfill the required applications.

The configurable logic blocks can be used to make up combinatorial, sequential or mixed circuits. Each of them includes several logic elements (LEs) or logic cells. Each LE consists of a four-bit lookup table (LUT), which can be configured either as a combinatorial function, or a (16 X 1) RAM or ROM, as shown in Figure 4.1. A carry-lookahead data path is also included in order to build efficient arithmetic operators. A D-type flip-flop, with its control inputs of synchronous or asynchronous set/reset and enable, allows the output of the LE to be registered. When its registered output is configured as its input, a LE can function as a microstate machine.

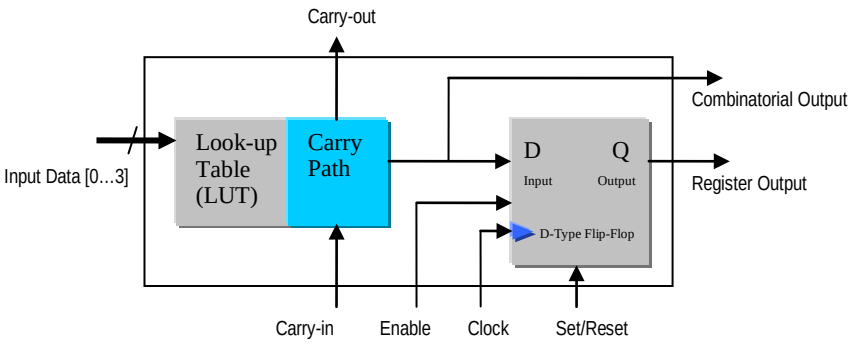


Figure 4.1 LE's Composition

The configurable I/O blocks can have different I/O elements. Each I/O block may contain a bidirectional I/O buffer, one input register, two output registers, and two output-enable registers. Each I/O element can be configured as an input, output, or bidirectional data path.

I/O pins support single-ended and differential I/O standards. Single-ended signalling uses only one signal line, and its voltage potential is referred to the ground. The signal line provides just the forward path and the ground offers the return path for the signal. The differential signalling uses two wires to send two complementary signals, which can improve resistance to electromagnetic noise compared with the single-ended signalling

but occupy one more pin.

The programmable interconnect network consists of switch matrixes and paths, as shown in Figure 4.2. An item of a switch matrix can be programmed to make a row path connect to a column path and make an output of one LE link to an input of another LE. A good layout of an interconnect network of a FPGA design can decrease area, delay and power consumption. Figure 4.3 shows a generic structure of an FPGA device.

Figure 4.2 Programmable Interconnect Network

4.2.2.2 FPGA Reprogrammable Technologies

Several configurable technologies exist, such as Flash, EPROM, SRAM and antifuse. The EPROM and SRAM technologies are just like the common memory uses of EPROM and SRAM in a microprocessor system.

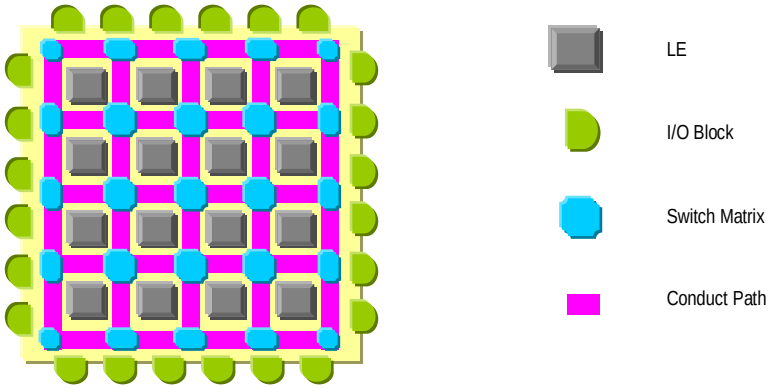


Figure 4.3 Generic Structure of an FPGA

SRAM is by far the most widespread in the FPGA field. SRAM-based FPGA stores LE configuration data in its static memory. Since SRAM is volatile and cannot keep data without a power source, an SRAM-based FPGA reads configuration data from an external Flash memory chip, which is called master mode. It can also be configured by an external processor via a boundary-scan (JTAG, joint test action group) interface, which is called slave mode.

A Flash-based FPGA uses a flash memory as a primary resource for configuration storage. The advantages of Flash-based FPGAs are less power consumption and greater tolerance to radiation effects. When the power is off, the flash memory can preserve the configuration of FPGA. Flash-based FPGAs fit into applications of space and aircraft industries.

An antifuse-based FPGA adopts antifuse technology. An antifuse does not conduct current initially, but can be fused to conduct current. Once an anitfuse-based FPGA is programmed, the process cannot be reversed. Compared with the previous two technologies, which can program FPGAs several times, an antifuse-based FPGA can only be programmed once.

The above manufacturers share the FPGA product market with their different technologies for reprogramming. Altera, Lattice, and Xilinx tend to use the SRAM-based FPGAs; Altera, Latice, and Xilinx also use the Flash-based FPGAs; Actel and Quicklogic adopt the antifuse-based technology.

4.2.3 Architecture Diversity in FPGA-based Systems

The following are some of the current architectures of FPGA-based systems.

4.2.3.1 Stand-alone FPGA-based Systems

Stand-alone FPGA-based systems can be ESs, which are often employed in consumer electronics, industry control systems, and portable applications. They have a typical ES structure, which is described in Chapter 3. After the system power is switched on, they can function well by themselves or interact with their environments, including the user interaction. With a real-time operating system, they can also handle several tasks concurrently. These systems are complete and independent.

4.2.3.2 General-Purpose Computer Systems with FPGA Supports

General-purpose computer systems with FPGA supports have the tightly-connected or loosely-coupled co-processor architectures.

In the co-processor architecture, the general-purpose computer is the host CPU whereas the FPGAs can have their own processors that assist to do specific tasks without the host's intervention. The tightly-connected co-processor model has a board connection between the host and FPGAs. It has a fast communication rate between the host and co-processors. The loosely coupled co-processor model allows the direct communication between the host and FPGAs by using some fast interconnection, such as point to point networks.

Since co-processors in FPGAs work simultaneously with the host CPU, this architecture can provide more parallelism than general-purpose computers. It can also lead to heavy

traffic on the serial bus when the data are transmitted between co-processors and the CPU. As long as the serial bus is not overloaded, this architecture will be more effective computationally than the model of CPU with simple I/O peripherals.

4.2.3.3 Reconfigurable Computing Systems

Reconfigurable computing (RC) systems have become more and more attractive in these last two decades (El-Ghazawi et al 2008, Green and Edwards 2000, and Huang et al 2009). Reconfigurable computing architecture can provide parallel processing at instruction or task levels. A central microprocessor is connected to several FPGA-based boards. The architecture allows scalable connections between parallel systems on FPGA-based boards dynamically. They offer more flexibility in terms of system layout, but they need more complicated design techniques to generate a good target design and implementation.

4.3 FPGA-based ES Development

If the hardware system implementation is selected, designers may target either an ASIC solution, or one based on FPGAs. Programmable hardware solutions are becoming increasingly attractive thanks to recent increases in logic capacities, improvements in performance, and the ability of some devices to be wholly or partly reconfigured during the runtime of a system (Green and Edwards 2000).

In Section 4.1, we discussed the difficulties of separate ES development. If a developer can pass through all three steps, i.e. hardware building, device driver development, and application programming, it will be easier to achieve the goal of the target application-specific ES with meeting all requirements. The basic condition is that the developer has to know what the hardware can and cannot do and also what the software is and is not good at. Even though the necessary expertise is still enormous, the Altera ESDK provides an achievable, systematic and coherent solution. Compared with traditional ES development, FPGA-based development has the following advantages.

- Programmability – the logic functions in FPGA are volatile, configured with a SRAM object file, and programmed electronically with a specified logic design. The embedded hardware system built in FPGA is not fixed and can be changed by a new configuration in a distinct SRAM object file.
- Customisation – via the FPGA, the hardware system can be customised according to the application goal without physically building of devices. The download of the design to the FPGA is just a configuration file. All the components in FPGA-based hardware systems are generated from the FPGA resources, such as LEs, in-chip memories, multipliers, PLLs, etc.

- Microprocessor substitute – since advanced FPGA devices consist of rich resources in one chip, it is practical to implement logic that a complete conventional microprocessor can handle. Even better, the FPGA can provide more processor and peripheral flexibilities in one chip than the conventional microprocessor.
- Complexity and Integration – not only common logic as a single-function peripheral but also sophisticated logic like a soft-core processor can be designed and implemented with FPGAs. For advanced FPGA devices, both soft-core processors and peripherals can be integrated in an FPGA chip.
- Changeability – the hardware system design based on the FPGA can be changed when the target application software cannot be programmed with the resources and device driver functions built initially. As the full design flow is a coherent process and can be repeated, the later step of the design flow can lead to the design modification of the previous step. As it is quicker to modify the design on the FPGA and reconfigure the FPGA compared with ASIC, the modification will not take up a lot of time and delay the design cycle.
- Systemic development – the JTAG interface of the Alter ESDK supports both hardware and software development. The development of a complete ES with hardware and software can be implemented systemically and coherently in one integrated design environment with one design flow and without crossing development platforms. It provides support to application-specific ES development.

The starting point for a design is to build a complete embedded hardware system with a soft-core Nios II processer and basic peripherals in an FPGA chip with necessary external devices. Altera provides an SOPC (system on programmable chip) builder to set up hardware systems with modules of processors, memories, peripherals and connections rather than from the gate level. All the modules have been verified and parameterized for specific hardware development. This saves a lot of time and energy in the development process.

4.4 FPGA Device for this Research

FPGA devices vary widely. Different manufacturers have different series of FPGA products. Altera has Cyclone, Arria, and Stratix series (Altera 2012). Cyclone series are groups of low power and high functionality and are the cheapest in the Altera FPGA product family, which can meet the power and function requirements of ESs. In the Cyclone series, there were five sub-series, Cyclone I, II, III, IV and V, in 2011. The devices in the Cyclone III sub-series consist of more resources in one chip than those in Cyclone I and II. According to the logic and I/O requirements of the design and restrictions of the

FPGA device, the Altera Cyclone III 3C120F780 meets the requirements of the graphics application for this project. The last two sub-series, Cyclone IV and V, were put on the market in 2009 and 2011, respectively. This project started in 2009. Thus, the Altera Cyclone III 3C120F780 was chosen as the main FPGA device of this project.

The basic features of Cyclone III devices include densities ranging from 5,000 to 200,000 LEs, 0.5 Megabits to 8 Mb of memory, and less than 1/4 watt of static power consumption (Altera 2012). Thanks to their low power consumption, Cyclone III devices can prolong the battery life in handheld and portable applications, cut down cooling system cost, and support work in the environment with thermal limitation. Cyclone III devices provide numerous external memory interfaces and I/O protocols that can meet the needs of high-throughput applications. The architecture of Cyclone III devices consists of many logic array blocks (LABs), M9K memory blocks, embedded multiplier blocks, PLLs, global clock networks, and I/O banks. The following describes some but not all of the features that Cyclone III devices have.

4.4.1 LABs and LEs

Each LAB consists of 16 LEs and a LAB-wide control block. An LE is the smallest unit of logic in this architecture. Each LE has four inputs, a four-input look-up table, a register, and output logic. The four-input LUT is a logic generator that can be used to configure any combinatorial logic with four variables.

4.4.2 M9K Memory Blocks

Each M9K memory block provides nine Kbits of on-chip memory that can operate at maximum 315 MHz. M9K memory blocks can be configured as RAM, fist-in first-out (FIFO) buffers, or ROM. They can also be configured as single-port or dual-port operation modes. Thus, after configuration, Cyclone III devices can be used in applications of embedded data storage, embedded processor program, and high-throughout data processing.

4.4.3 Multiplier Blocks and DSPs

The embedded multiplier blocks support two modes: one is an individual 18 X 18-bit multiplier; the other is two individual 9 X 9-bit multipliers. In addition to multipliers, by using a combinational logic with on-chip resources and external interfaces, the blocks can be configured into high-performance, low-cost, and low-power-consumption DSP systems. Altera’s facilities also provide DSP IP (Intellectual Property) cores for functions of finite impulse response (FIR), fast Fourier transform (FFT), and numerically controlled oscillator (NCO).

4.4.4 PLLs and Global Clock Networks

With PLLs and global clock networks, the Cyclone III devices can generate a maximum of ten internal clocks and two external clocks from a single external clock source. They also provide multiple input source frequencies, with which functions of multiplication, division, and phase shift are supported.

4.4.5 I/O Banks

Each Cyclone III device contains eight I/O banks. To protect signal integrity and gain high I/O performance, all the banks support programmable functions for drive strength, pull-up resistors, delay, bus hold, and slew-rate control. They support the capability to plug a board in or off a system during operation without having negative effects to the system or the board, which also called hot socketing.

The I/O banks also support single-ended and differential I/O standards as well. The high-speed differential interfaces of Cyclone III devices can transmit data at a maximum rate of 875 Mbps without the need for external resistors.

4.4.6 Embedded Processors

For ES design, Cyclone III devices provide several choices for processor cores. For example, they can be ARM Cortex M1, Freescale V1 Coldfire, and Altera Nios II. For this project, Nios II has been adopted.

They also extend the peripheral set, memory, and I/O interface in order to build up a whole ES with high performance and at low cost.

The Cyclone III 3C120F780 consists of 119,088 LEs, 432 M9K blocks, total 3,981,312 RAM bits, 288 of 18 X 18 multipliers, four PLLs, twenty global clock networks, and maximum 531 user I/Os. It has a small size. Both of its length and width are 29 mm. Its height is 2.60 mm.

4.5 FPGA-based ES Design Flow

The hardware is good at simple repeated operations whereas software does complex algorithms well. During design and development, if the features and strengths of both hardware and software can be considered and made the best use of, requirements can be met and the application goal achieved with a high-performance, low-cost target ES. Thus, a high overall performance of the target ES can be guaranteed.

Since an ES has specific applications, its system structure and components can be customised accordingly. Furthermore, the design and development of an ES should be

based on the application requirements. Figure 4.4 shows the development flow of combined hardware and software for application-specific ESs.

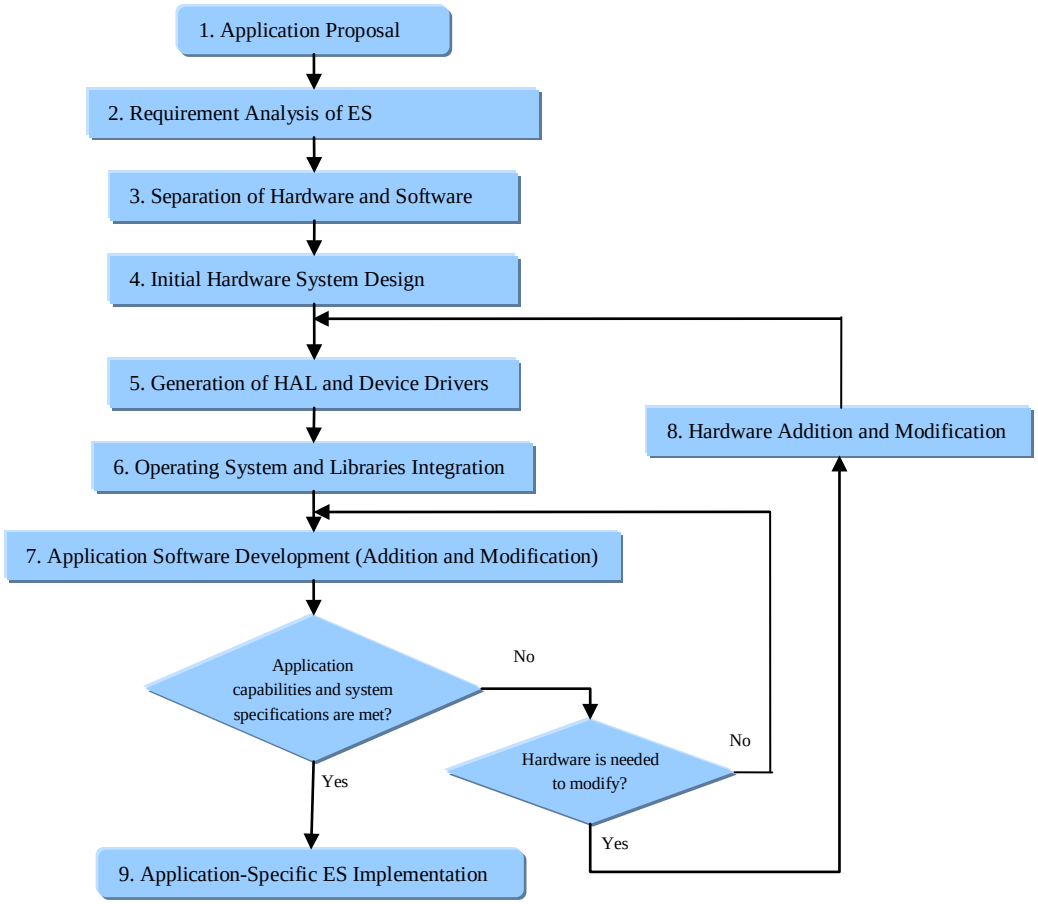


Figure 4.4 Development Flow of Combined Hardware and Software for Application-specific ESs

The FPGA-based ES design flow is as follows.

4.5.1 Application Proposal

The development of an ES starts with applications being specified – i.e. the application proposal, the first step in Figure 4.4. In this step, the applications of the target ES should be defined. Then the flow moves to the second step.

4.5.2 Requirement Analysis of ES

Requirement analysis of the target ES has to be performed. That is the second step of Figure 4.4. The requirements of the target ES are separated according to their hardware and software tendency. The requirement analysis result should provide clear instructions about how to implement the target system in terms of hardware and software. It helps to decide which part of the application should be constructed with hardware and which part of with software. Then the flow moves to the third step.

4.5.3 Separation of Hardware and Software

With a clear requirement analysis result, a boundary between the hardware and software for the target ES can be defined. This is the third step in Figure 4.4, separation of hardware and software. The separation results in construction of hardware and software parts. These parts can be constructed in parallel or in sequence, depending on the dependent relationship between them. If a dependent relationship exists, the software parts must be programmed after the hardware system (that is constructed in the fourth and fifth steps), for example, the software parts that have to communicate with the hardware. Then the flow moves to the fourth step.

4.5.4 Initial Hardware System Design

With the requirement analysis result and after the separation of hardware and software, the initial design of the hardware system can be done. With the building blocks for hardware, as shown in Figure 3.2, the hardware system can be constructed in the design environment, such as the Altera ESDK, and stored in files. After compilation, synthesis and verification, a configuration file can be generated and downloaded to the FPGA device. This is the fourth step in Figure 4.4. Then the flow moves to the fifth step.

4.5.5 Generation of HAL and Device Drivers

Depending on the components adopted in the hardware system, such as processor and peripherals, the HAL and device drivers can be generated, which is the fifth step in Figure 4.4. This is also the second layer in Figure 3.3. Most of the software in this layer is generated by the design environment, for example Altera ESDK. If a new component is used in hardware design but is not available in the SOPC libraries, the device drivers and the relevant API of this new component must be programmed and added to the HAL. Then the flow moves to the sixth step.

4.5.6 Operating System and Libraries Integration

Along with the HAL, a real-time operating system and necessary libraries (including the ANSI C standard library and application-specified library) can be added to the development. This is the sixth step in Figure 4.4 and the third layer in Figure 3.3. Then the flow moves to the seventh step.

4.5.7 Application Software Development

The application software can be programmed and downloaded to the target board to run and debug, which is the seventh step. Then the first design must be verified whether or not it meets the needs of application capabilities or system specifications. If it does, the design

of application-specific ES is accomplished, and the process moves on to the ninth step. If it does not, the design must be modified. Before the modification is done, it has to be determined whether or not the hardware system has to be modified or a new hardware module is required. If not, the modification should be done only on the software and the application software can be added or modified to enhance its performance, which means going back to the seventh step. If yes, the modification on hardware must be done, which is the eighth step Figure 4.4. The flow moves to the eighth step.

4.5.8 Hardware Addition and Modification

The hardware system has to be changed or some new modules have to be added to the hardware system. This results in the second hardware design. The second hardware design is carried out. The second configuration file is generated and downloaded to the FPGA device. The second iteration of development passes through the fifth step to the seventh step. The ES development can take several iterations until the design meets the application requirements and system specifications. Then the flow moves to the ninth step.

4.5.9 Application-Specific ES Implementation

The application-specific ES design meets all the application requirements and system specifications and the goal is accomplished.

4.6 FPGA-based ES Design with Altera Facilities

The Altera EDS (embedded design suite) and other facilities (Altera 2008a) provide an integral solution to the design and development of application-specific ESs. They take into account both hardware and software in the development flow of an ES. They provide different tools for hardware and software development, respectively, and these tools have been coherently linked together. Some tasks for system-level building and generating can be done automatically by the tools, which can save a lot of effort and time and avoid some mistakes caused by unfamiliarity and misunderstanding of designers and developers in the separated ES development, as discussed in Section 4.1. They are an effective tool suite for the ES development.

For FPGA-based hardware development, Altera provides the Quartus II and the SOPC builder. To build the software for ESs, they use Nios II EDS, including Nios II SBT (software build tools), Nios II Command Shell, and Nios II IDE (integrated design editor) with associated simulation tools. No matter they are for hardware or software, since they have internal coherent relationships these tools are linked to each other by their input and output files according to the natural sequence of design and development.

In the research of this project, Step 4 and Step 8 in Figure 4.4 are implemented with Quartus II and SOPC builder. Steps 5 to 7 are done by means of Nions II SBT, Nios II Command Shell, and Nios II IDE. To make the best use of them, the methodologies for the research adopted the design and development philosophy of these tools.

As the SOPC builder adopts a top-down design way, it conforms to the principles of modern system design principle and the development flow combining hardware and software of application-specific ES, as in Figure 4.4. The Altera IP cores and megafunctions, such as Nios II cores and standard components, support most ES development on the Cyclone III chip. They can be used to set up a target embedded hardware system. Figure 4.5 shows the development flow for embedded hardware systems with Quartus II.

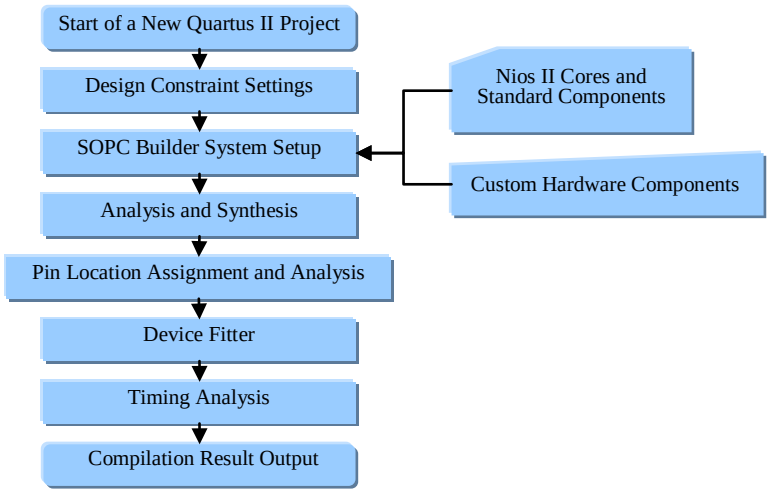


Figure 4.5 Development Flow for Embedded Hardware Systems

4.6.1 Start of a Quartus II Project and Design Constraints

The development flow for embedded hardware systems begins with a new Quartus II project, with which Quartus II manages all the files related to a new hardware design. Before the start of a real design, design constraint settings should be done manually. These settings produce explicit limitations for device usage, analysis and synthesis, time analysis and other requirements during design compilation to achieve a required design result.

The Quartus II tool includes megafunctions that are used to control the operation mode of the embedded multiplier blocks based on user parameter settings. The multipliers can also be deduced directly from the Verilog HDL or VHDL source code.

4.6.2 SOPC Builder System Setup

The SOPC Builder is a design tool integrated in the Quartus II environment. With a library of more than 50 IP blocks, the SOPC Builder can integrate IP blocks into an FPGA system-level design. The SOPC Builder can generate interconnect logic automatically and create a testbench to verify functionality.

With the SOPC Builder, Altera IP cores and megafunctions, a new hardware system design can be set up. Sometimes, custom hardware components that are not available in the Atera standard components have to be created with one of the HDL languages, such as Verilog HDL and VHDL, and integrated into the user custom components.

The SOPC Builder is used to create a SOPC system that is a representation of a real hardware design. Differently from the conventional bottom-up FPGA design, which is programmed with the HDL, the SOPC builder adopts a straightforward GUI (graphical user interface) to aid the design and generate an HDL file for it, as shown in Figure 4.6. The compilation result of the HDL file is an SRAM object file, which is the usable design and can be downloaded into an FPGA device.

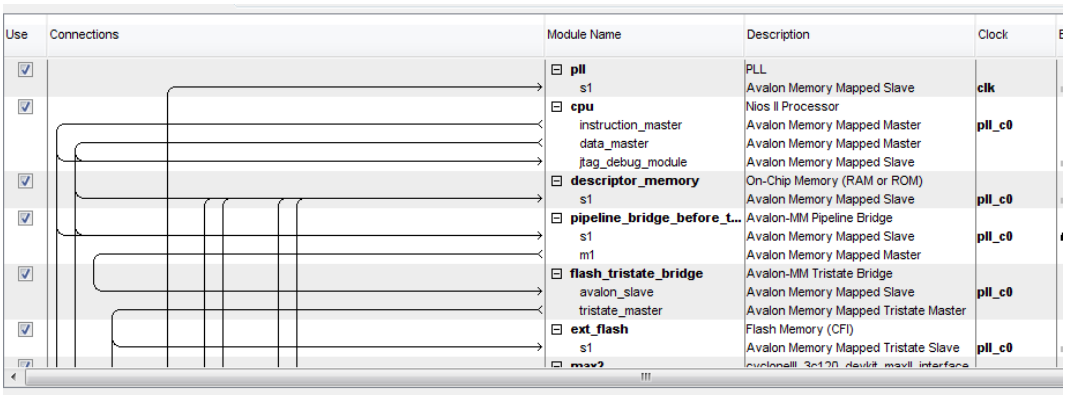


Figure 4.6 Part of the SOPC Builder GUI

Along with a library of 50 other IP blocks in SOPC Builder tool, the embedded processor can be selected among Freescale V1 Coldfire, ARM Cortex M1, and Altera Nios II.

4.6.3 Analysis and Synthesis

The analysis and synthesis engine automatically performs the following tasks: it verifies the design, removes the redundant logic by using don't-care conditions, detects the feedback loops in combinational logic, finds the unused states, removes equivalent states, does state assignments, synthesises the logic to meet the constraints of area or speed in the FPGA device, and optimises and maps the result into a hardware device. The analysis and synthesis are processed automatically by the Quartus II facilities according to the above design constrain settings.

Conventionally, whether or not the synthesis of a design is successful has mostly depended on how well the models written with the HDL accord with synthesis rules. The written models must conform to the constraints of the synthesis tool. With the SOPC builder, since all of the standard components have been verified, the analysis and synthesis engine can optimise logic quickly without errors.

4.6.4 Pin Location Assignment and Analysis

Since they are implemented on an FPGA chip, the placement of each pin must conform to the microelectronics restrictions of the chip and the physical limitations of the board layout. For this reason, the pins on an FPGA chip are sometimes scare and valuable resources and cannot be allocated without limitations. Physically, an FPGA device has I/O banks, I/O standards, and VREF groups. Understanding the physical features of the FPGA chip is helpful in terms of pin placement. Some features that can influence the pin placement are listed as follows.

- The I/O pins are grouped into different I/O banks in order to support different I/O standards. Each bank has its own voltage source pins, VCCIO pins. The pins in an I/O bank must share the same VCCIO voltage source.
- A VREF group is a group of pins that share a same reference voltage pin, a VREF pin. An I/O bank usually consists of one or more VREF groups. Thus, the pins in a VREF group have the same VCCIO and VREF voltages.
- In the silicon die of a FPGA device, the I/O pins connect to the bond pads that are on the perimeter ring of the die top. To guarantee the signal integrity, there are restrictions on the minimum number of pads separating single-ended input or output pins with a differential pin. There is also a restriction on the maximum number of I/O pins supported by a VREF pad.
- Except VREF and VCCIO pins, there are some other pins specified for fixed uses that cannot be assigned by users any longer. The pins available for user I/O are far fewer than the set of pins that an FPGA device has. For example, some pins have their original usages that were retained by the manufacturers when they were produced.

These features have been merged into the pin placement rules of the pin location analysis of Quartus II. It is useful to know them when errors are reported and needed to correct during the pins are placed manually.

For a hardware design, the I/O planning is a detailed and comprehensive task. It determines whether or not a design can be implemented with an FPGA device. Therefore, the I/O planning influences hardware design throughout. There are several issues that

should receive more attention. They are listed as follows.

- The first step of I/O planning for a hardware design is to select a proper FPGA device that has adequate pins according to the logic and I/O requirements of the design and the above restrictions of the FPGA device.
- During the pin location assignment, the pin assignments are done along with the definitions of I/O attributes, including I/O standard, slew rate, drive current strength, and output load for output and bidirectional pins. A higher slew rate means faster transitions and more noise transients for high-performance systems, and influences both the rising and falling edges of signals. The drive current strength settings can reduce the influences of simultaneously switching outputs and the system noise. The pin location assignments have to be verified with I/O assignment analysis.
- For some input or output ports, timing constraints can restrict some pin location. But this type of effect is not straightforward and can be verified at the later step, device fitter, as shown in Figure 4.5.
- After complete compilation, a validated pin-out file for PCB (printed circuit board) tools is created and is ready for the board layout. If the compilation is failed, the pin location assignments have to be modified according to the reported errors.

The Pin Planner in Quartus II is a tool for pin location assignment. It can assist designers to create, modify, complete and validate pin-related assignments. With the Pin Planner, pin location assignments can be done. But the verified pin location assignments do not rely on the Pin Planner tool. Some detailed I/O resources on a typical FPGA device are quite different from others. Some pins have their original usages and cannot be assigned for other purposes. To prevent signal integrity issues, the DDR (double data rate) interface has special constraints on the number of output pins in a VREF group. Noticing these restrictions is not the responsibility of the Pin Planner. They should be known by the designers before using the Pin Planner.

I/O elements of Cyclone III devices contain five registers: one input register, two output registers, and two output-enable registers. The input register can be used for fast setup times, and the output registers and output-enable registers are used for DDR applications.

4.6.5 Device Fitter

It is not evident to designers what the device fitter does. But it is very important for a successful design. Typically, it maps a design to the physical LEs, I/O element and interconnect network of a specified FPGA chip. Since there are so many LEs (119,088 for 3C120F780) in a typical FPGA chip, the number of different solutions for a design can be

huge. It means that the fitter has to work through an immense solution space to look for a good mapping area as quickly as possible. This is why a small change in one part of a hardware design can lead to different results in other parts and partly why the hardware design compilation is time-consuming.

The fitter needs restrictions to determine how to map. The two primary restrictions are timing and routing requirements. The timing requirement makes all the timing-critical paths meet their timing requirements and reduce the signal delay. The routing requirement is intended to make sure that the mapping meets the physical limitations of the target FPGA chip and puts connected elements closer together.

To reduce the compilation time, Quartus II provides the Fast Fit option, besides Standard Fit and Auto Fit. Fast Fit can decrease compilation time by a maximum 50% for a design by reducing fitter effort. Standard Fit does not reduce fitter effort, but Auto Fit can decrease fitter effort after meeting the timing and routing requirements of a design. Fast Fit may take half an hour to several hours to accomplish, which depends on the memory space and CPU speed of the computer where the design of the embedded hardware system is processed. More free memory space and higher CPU speed can accelerate the device fitting.

4.6.6 Timing Analysis

Timing analysis tests whether or not a hardware system design meets the timing requirements. As regards a high-speed design, the propagation delay is vital for proper system operation.

With Altera TimeQuest Timing Analyser, a static timing analysis can be done on register-to-register, I/O, and asynchronous reset paths. The TimeQuest Timing Analyser can detect possible timing violations by using clock arrival times, times required by data processing, and data arrival times.

4.6.7 Compilation Result Output

Quartus II provides many compilation and simulation reports for different design stages: for example, compilation flow reports, analysis and synthesis reports, partition merge reports, fitter reports, and TimeQuest Timing Analyser reports. At any stage, the tools can give a report for that stage. With these reports, designers can make modification on their designs. The most important ones are the configuration files that can be downloaded to the target FPGA device and make the FPGA function as the target embedded hardware system.

4.6.8 About Go-Back

In Figure 4.5, the development flow does not give any explicit return to the previous steps like those in the software, but almost any step in this flow can go back to its previous steps. The compilation of a hardware design can be stopped in half way unsuccessfully by some constraints, such as constraints of the design logic, the pins and internal resources of a specified FPGA device, time analysis, and others. Therefore, if the compilation of a design stops at any point, it is the only way to modify the design until it passes all the compiling requirements. The analysis and synthesis have to be redone if any change is made to the design. Fortunately, most of the tasks are performed automatically by the facilities of Quartus II. They can help reduce the time-consuming work.

4.7 Setup of FPGA-based Embedded Hardware System for Graphics Applications

As an algorithm for the surface modelling and editing, the PAMA is one of 3D graphics applications. An FPGA-based embedded hardware system has to be constructed for the graphics applications. Besides the microprocessor, memory and general peripherals, an LCD, frame buffers and hardware units for graphics pipelines have to be specified for this project. To build up an FPGA-based embedded hardware system, as shown in Figure 3.6, two tools provided by Altera Corporation must be used, Quartus II and SOPC builder.

4.7.1 Nios II Processer Settings

As a system component, a Nios II processor is a soft core that is volatile and present only after the FPGA is configured, as shown in Figure 3.6. Therefore, it must be added to the SOPC system with the SOPC builder when the embedded hardware system is set up.

As shown in Figure 4.7, there are three configurations of the Nios II processor, the lowest configuration is Nios II/e (size-optimised economy), the middle one is Nios II/s (standard), and the highest one is Nios II/f (performance-optimum fast). Since the graphics speed-up with hardware can support the graphics acceleration with the hybrid way that is the goal of this project, the Nios II/f has been chosen. Besides the RISC and 32-bit structure of the Nios II/e, the Nios II/f consists of hardware supports for the instruction cache, data cache, dynamic branch prediction, hardware multiply, hardware divide, and barrel shifter. All the elements can accelerate the computation systemically. The accelerated computation is critical for the algorithms of 3D graphics speed-up.

The Nios II/f processor is configured to run at the 100-MHz frequency. Its performance can be up to 113 DMIPS. Its logic usage in FPGA is 1400-1800 LEs. The reset vector is located at the physical address 0x10000000 and offset 0x0 in the ext_flash, which is the external 64-MByte CFI (common flash interface) flash. The exception vector is located at

the physical address 0x1c000040 and offset 0x40 in the ddr_sdram_1, which is one of two external DDR2 SRAM memories. Since the operating system does not support the memory management, both of MMU (memory management unit) and MPU (memory protection unit) options are disabled. In Figure 4.8, both the instruction and data caches are set to 32 Kbytes. The data cache line size is 32 Bytes.

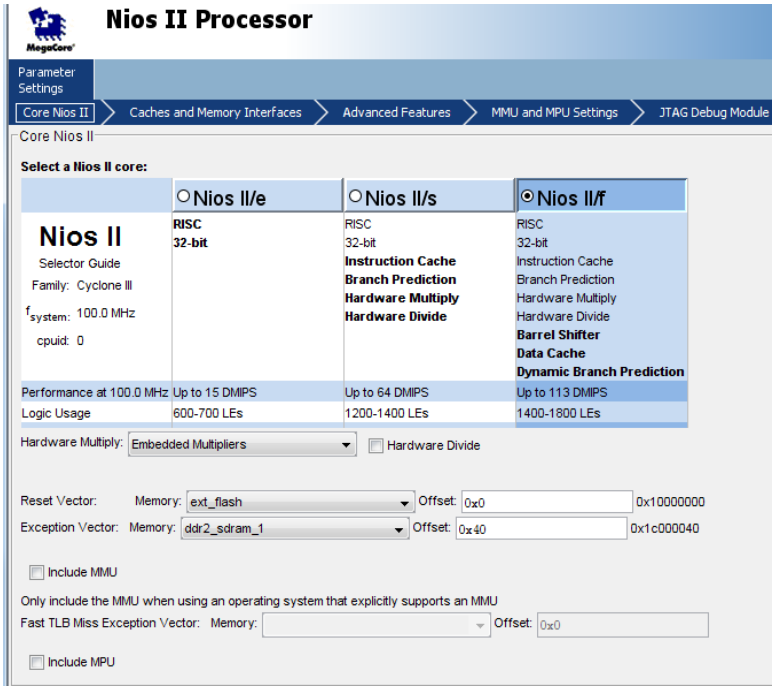


Figure 4.7 Nios II Settings (1)

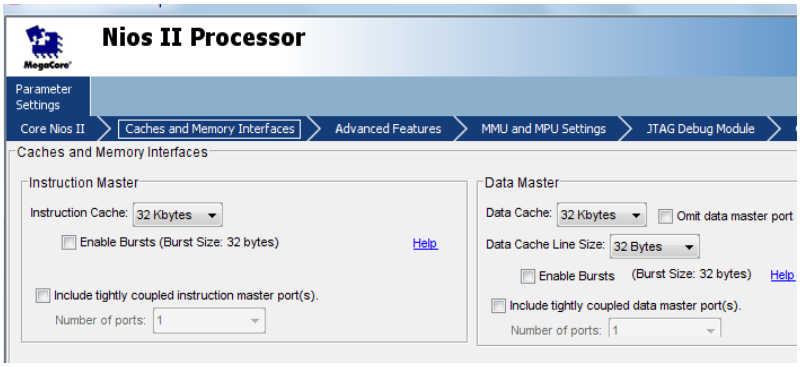


Figure 4.8 Nios II Settings (2)

4.7.2 System Clock Settings

To enhance the whole performance of the system, different peripherals run in the different clock domains from one of Nios II processor.

Since the Nios II and peripherals function under different system clocks, the system has

two external clock sources. One is 50 MHz; the other is 125 MHz. From the 50 MHz clock, a PLL is used to produce two clocks. One is 100 MHz for the CPU and the LCD; the other is 60 MHz for the slow peripherals. The 50-MHz clock is for one of the two external DDR SDRAM memory controllers and the 125-MHz clock is for the other one.

Because the Nios II processor is connected to the peripherals that run in different clock domains from its clock, three clock crossing bridges are needed. Two bridges are used to connect the CPU with two external DDR SDRAM memories, respectively. One bridge connects the CPU to the slow peripherals, such as system timers, and others.

4.7.3 DDR2 SDRAM Memory Controller Settings

For graphics applications, frame buffers are necessary. As an ES, the system code and data have to be stored during the system runs. These requirements result in the system memory. Two external 64-MByte DDR2 SDRAM memory banks of Micron MT4732M16CC-3 are used in the system. They can run at a full rate of 153.85 MHz or half rate of 76.9 MHz.

One memory that stores video frames is controlled by one of two controllers and connected to the entry of the video pipeline. It plays the role of video frame buffer. The video frame data stored in the buffer adopt a 64-bit format. The entry of the video pipeline is an SG-DMA, which is set to a 64-bit width at the half rate of 76.9 MHz and delivers the video stream data from the memory into the FIFO of the video pipeline. Thus, the memory is controlled to run at the half rate of 76.9 MHz with a 64-bit data width.

The other memory is controlled by another controller. It is the system instruction and data memory, stores instruction code and data, and is connected to the Nios II data bus. It runs at the full rate of 153.85 MHz with the 32-bit data width.

4.7.4 CFI Flash Memory Controller Settings

FPGA configuration files of FPGA-base ESs have to be stored in a permanent memory. For graphics applications, the application software also has to be also stored in a permanent memory. A Spansion Flash memory with 64-MByte capacity has been chosen as the permanent memory in the present research. It stores the FPGA configuration data and application programs for this project.

The CFI (common flash interface) -compliant flash memory controller is set to control this external flash device. Its address width is 25 bits, and its data width is 16 bits. The setup time, wait-state time, and hold time for read and write transfers are set to 80.0 ns, 40.0 ns, and 20.0 ns, respectively.

4.7.5 JTAG UART Settings

During ES development, it is necessary to communicate between a PC host where the ES development is practised and the FPGA development board where the target FPGA device is located. The JTAG UART (universal asynchronous receiver/transmitter) is used to build up the serial communication between the PC host and the FPGA development board. Through the JTAG port and cable, the FPGA configuration file and application software are downloaded to the devices on the FPGA board. The write FIFO (from Avalon interface of FPGA to JTAG) of the JTAG UART is set to eight bytes of buffer depth and four of IRQ level. Its read FIFO (from JTAG to Avalon interface) is also set to eight bytes of buffer depth and four of IRQ level. Both of them are constructed by using the on-chip registers.

4.7.6 Settings for LCD Controller Interface and Video Pipeline

For graphics applications, an LCD and video pipeline are the final part of the graphics pipeline. The LCD is the screen device used to display the pixels of a graphics image. Since the data format stored in frame buffers is different from one streaming to the LCD device, the video pipeline is used to do the data matching and synchronising. In this project, the LCD device on the board is a 4.3” Toppoly TD043MTEA1 active matrix colour display with 800 X 480 pixel resolution. The LCD controller interface and video pipeline are integrated into the system.

4.7.6.1 LCD Controller Interface

The LCD controller interface built with three Altera PIO (parallel I/O) cores consists of three one-bit ports, including <lcd_i2c_scl>, <lcd_i2c_sdat>, and <lcd_i2c_en>. The <lcd_i2c_scl> is an output port for the LCD controller clock output. The <lcd_i2c_sdat> is a bidirectional port for the LCD controller data. The <lcd_i2c_en> is an output port for the LCD controller enable.

4.7.6.2 Video Pipeline

The video pipeline is composed of IP cores that can be customised to suit the resolution and aspect ratio of the LCD device. Besides the video frame buffer and SGDMA that were introduced in Section 4.7.3, the video pipeline is composed of an FIFO, two data format adapters, a pixel format converter, and a video sync-generator.

The FIFO is set to a dual clock FIFO with 128-unit depth and constructed with the on-chip memory blocks. It can buffer video stream data when the rate of fetching video stream data from the video frame buffer is faster than the rate of displaying the pixels on the LCD device.

One of two data format adapters turns the 64-bit frame data into 32-bit data. It is set to

eight data symbols per system clock for input and four data symbols per system clock for output. Each symbol has eight bits.

The pixel format converter is designed to take the 32 bits from the upstream and send 24 bits to the downstream by discarding eight bits. The 24 bits consist of eight bits for each of three channels of red, green and blue.

When sent out of the FPGA chip, the pixel stream data are a stream of three eight-bit data. It needs the other data format adapter to turn the 24-bit stream into the eight-bit stream. This adapter is set to three data symbols per system clock for input and a data symbol per system clock for output.

The video sync-generator is used to synchronise the RGB pixels in rows and columns in an image by means of horizontal and vertical synchronisation signals. As the display scan is done line by line, the horizontal synchronisation for a whole line of pixels should be done before the vertical synchronisation. The horizontal synchronisation of 800 RGB pixels in a line is done at the rate of one pixel per system clock. The vertical synchronisation of 480 lines of pixels in an image is done line by line. Since the horizontal blank pixels are set to 216, horizontal front porch pixels are 40, and horizontal sync pulse pixels are one, the total number of horizontal scan pixels is 1056. The vertical blank lines are set to 35, vertical front porch lines are ten, and vertical sync pulse lines are one. So the total number of vertical scan lines is 525. Figure 4.9 illustrates the video pipeline.

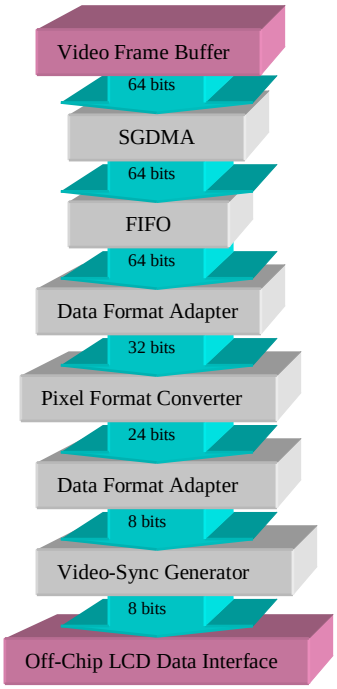


Figure 4.9 Block Diagram of Video Pipeline (Purple Blocks are Off-Video-Pipeline Blocks)

The video pipeline provided by the Altera is used simply for the fundamental functions of the control and transfer of pixel data to the off-chip LCD display device. It is not sufficient for the graphics pipeline. The rest of the graphics pipeline is implemented by an algorithm-specified module and Mesa-OpenGL implementation. The detail of graphics pipeline will be discussed in Chapter 5.

4.8 Challenges and Features of the FPGA-based ES

Several challenges are tackled during the system setup and when the solutions are tested to achieve the integrated system implementation.

A Samsung R480 laptop computer with an Intel Core i5 CPU M 460 of 2.53 GHz and 4.00 GB was used in software programming for this project. It does not have sufficient computing power for compiling the design of the FPGA ES in this research, however. An HP graphics workstation is used, which adopts the architecture of ACPI multiprocessor PC and consists of four 2.27-GHz Intel Xeon E5607 microprocessors. This proves the effectiveness of parallelism.

This project adopts a forward method suggested by the study of Underwood 2004. It applies a number of design interactions. It starts with the simplest, most straightforward implementation. Then it gradually adds the advanced modules to the simple implementation and adjusts each new module to verify its function until it achieves the best solution or runs out of FPGA resources.

Conventionally, it is taken for granted that hardware design is fixed with less flexibility software. It is now possible to change hardware design with current FPGA design tools even though substantial time and experience are needed. The FPGA-based platform provides a design-friendly environment for updating a design to realise new academic ideas and new market requirements.

4.9 Chapter Summary

In this chapter, a traditional ES development method is discussed and compared with the novel development of an FPGA-based ES. The evolution and applications of FPGA devices are introduced and FPGA-based ES development described. FPGA-based ES design and Altera facilities for FPGA development are presented for this project. The FPGA-based embedded hardware system is developed for graphics applications. Finally, challenges and features of the FPGA-based ES are discussed.

Constructing a system from hardware can help designers to customise and reuse the resources. The FPGA-based platform can provide a design-friendly environment for

updating the design to realise new academic ideas and market requirements. This research presents a novel alternative to GPUs and designated video cards by applying an FPGA-based embedded hardware system to computer graphics. Further development of the system incorporating OpenGL is presented in the next chapters.

Chapter 5 Integrating Mesa-OpenGL into FPGA-based ES

Discussed in Section 2.3, the OpenGL is a standard for applications of drawing high-quality images of 2D and 3D objects in real time with a user interaction interface. The user interaction interface allows the user to input operation instructions to edit any object or modify any image frame in real time. With different hardware platforms, OpenGL implementations are varied, especially for the hardware-dependent part.

Derived from the OpenGL, the OpenGL ES (OpenGL for Embedded Systems) is one of OpenGL standards specified in 2D and 3D graphics on ESs, including mobile phones, hand-held gadgets, and automobiles. As the range of ESs is wide, OpenGL ES implementations are varied with hardware devices and applications.

For this project, with a goal different from the general OpenGL ES, it is expected to realise the surface editing with FPGA-based implementation and must make a modification and addition to the general OpenGL ES implementation. The PAMA presented in this research must be supported with the algorithms of Bézier curves and surfaces. The evaluation of Bézier curves and surfaces does not belong to the general OpenGL ES and must be added to the OpenGL implementation of this project.

As the introduction in the previous chapters, the Altera ESDK, Cyclone III Edition (Altera 2008a), is used in this project. This kit is comprised of three components, a Cyclone III 3C120 FPGA base board, a LCD Multimedia High Speed Mezzanine Card (HSMC), and a multi-purpose card for debugging software and developing USB and SD card interfaces. The software processer core of Nios II plays the role of the general purpose microprocessor of the ES, which carries out the execution of the graphics pipeline and applications. The Nios II 3C120 general purpose processor system has 100MHz CPU clock and 60 MHz peripheral clock. This kit will be called the FPGA-based ES board (or platform) in the following discussion.

In this chapter, after the introduction of the OpenGL and OpenGL ES standards, the differences between the OpenGL ES standard and the implementation of this project will be expounded in detail.

5.1 OpenGL

According to Kilgard 1997, and Kilgard and Akeley 2008, the OpenGL can be treated as an architecture, which provides a well-specified, widely-accepted pipeline for 3D graphics, and an OpenGL-capable computer is a hardware implementation or exemplification of the architecture. In the OpenGL, the graphics pipeline is also called the state machine with a fixed topology. The OpenGL's state variables are orthogonal. Rendering steps can be broken down and embodied in special-purpose hardware in order to accelerate an object's drawing.

Described in the studies of Hearn et al 2011, and True et al 2004, the OpenGL pipeline consists of two joined-together sub-pipelines, geometric pipeline (or vertex pipeline) and fragment pipeline (pixel pipeline), as shown in Figure 5.1.

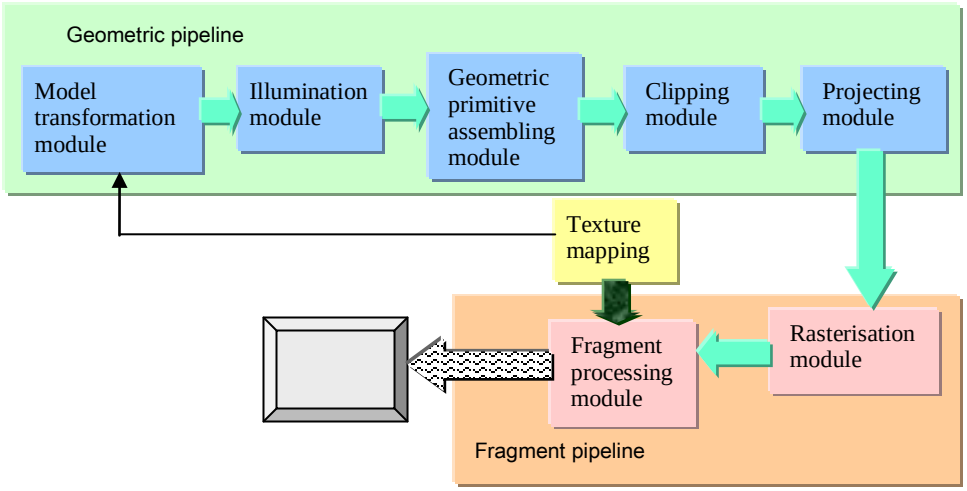


Figure 5.1 Graphics Pipeline

5.1.1 Geometric Pipeline

The geometric pipeline of OpenGL follows a natural process to draw graphics objects on a device screen in the analogous way of photography (Cunningham 2008, and Hoschek and Lasser 1993). This process carries out a transformation from a drawn 3D object that is convenient to be drawn by the application programmers to a 2D image that is easy to be displayed on the screen by a hardware device. This process can be simplified as a series of stages including

- Drawing objects individually one object after another;
- Placing each of them on a proper location in a common scene;
- Seting a viewing direction for the audience;
- Clipping the visible part according to the viewing volume;

- Projecting the 3D view onto a 2D viewing plane that is vertical to the viewing direction of the audience;
- Mapping the 2D viewing plane onto a 2D screen window of a hardware screen device.

As regards this process, it is conventional to define a scene where a collection of graphics objects exist. The scene defines what exists in the scene world, so its coordinate system is called the world coordinate system and the coordinates referring to this system are the world coordinates. The world coordinate system is shared by each object in this scene. Since each object is drawn individually first and then moved into the scene, it is convenient to assign a local coordinate system to each object in such a way that the object is easy to be drawn geometrically. For example, the centre of a sphere is a good option as the origin of its local coordinate system. Because of this process, there are several coordinate transformations involved in the geometric pipeline of OpenGL, which will be detailed in the following sub-sections.

5.1.1.1 Model Transformation Module

In the geometric pipeline, to create an image of a 3D object, the geometric descriptions of the object must be input. These descriptions are composed of the modelling coordinates or local coordinates. To put all the objects in a common scene, the coordinate transformation from local coordinates to world coordinates must be done. This is the work of the model transformation module.

5.1.1.2 Illumination Module

If an object in the scene is visible, the colour of the object will be drawn on the device screen. The light that emits on the object can influence how colour shades to look like on the device screen. In the OpenGL, the attribute of a light source can be set, which will make contribution to the evaluation of the final colour of an object lit by the light. This is the work of illumination module.

5.1.1.3 Geometric Primitive Assembling Module

To draw in detail an object with a complex shape, some technologies should be taken. One of them is to divide the object into simple primitives, such as triangles, cubes, spheres, cylinders, and others, in advance. These primitives are modelled and transformed individually. Finally they must be assembled together in order to represent the object. This is the work of geometric primitive assembling module.

5.1.1.4 Clipping Module

In general, there is a viewing point, from which the audience look at the scene. Like a camera viewfinder, there is a viewing volume in the OpenGL, called viewing frustum,

which limits the sizes of three dimensions. Objects in the scene must be clipped by the size of the viewing frustum. The part outside the viewing frustum will be invisible and have to be removed. The part inside the viewing frustum will be retained for the next step in the pipeline. These are the work of the clipping module.

5.1.1.5 Projection Module

The objects in the scene are then projected to a 2D plane that corresponds to the screen of the display device. The third coordinates of the scene will be used in depth test later. In general, coordinate values are normalised in the range from -1 to 1. In this way, the graphics software is independent of the coordinate range for any specific display device, no matter which is 800 X 480 pixels or something else. The above is the work of projection module.

5.1.1.6 Texture Mapping

As shown in Figure 5.1, except for the illumination module, texture mapping does also affect the rendering of an object. For example, a 2D texture can be used to replace the colour of or paste on the surface of a 3D object and make the surface of the object coated with the texture's pattern. Because of the texture coordinates' mapping onto the surface of the object, if the user interaction makes the object shape deformed, the coating texture must be changed along with the deformation. If not, an artificial effect may be caused.

5.1.2 Fragment Pipeline

Before rasterisation, everything related to an object is represented with the vertex coordinates. Therefore, it is a geometric description. The geometric description is convenient for geometric evaluation and application programming. But for the display device, everything drawn on the screen is painted as pixels, the values of red, blue and green at different positions with two-dimension coordinates on the screen.

In OpenGL, a fragment is a pixel with attributes of position and colour (Kuehne et al 2005). Fragment pipeline is responsible for shading or colouring the appropriate pixels in the frame buffer.

5.1.2.1 Rasterisation Module

Rasterisation is to make the normalised coordinates of vertices of objects scaled with the size of the display screen. After rasterisation, representations of objects are expressed in pixels or fragments.

Except for rasterisation module, typically, the fragment pipeline has several processing stages, as shown in Figure 5.2. In the fragment pipeline, many stages must be traversed before a fragment being written into a frame buffer. But it is costly and makes the rendering slow if all of them are enabled and processed. Therefore, it is wise to make a

control to optimise which combination of fragment processing stages is used in an application.

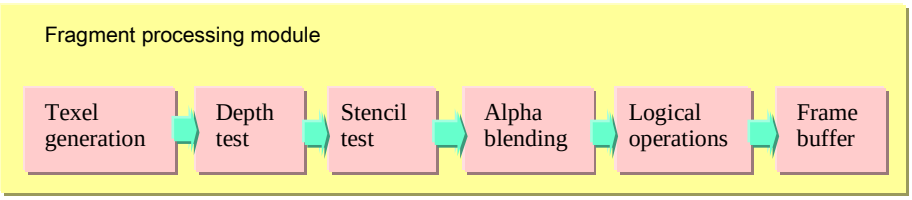


Figure 5.2 Processing Stages in Fragment Pipeline

5.1.2.2 Texel Generation

A texture element, called texel, is a unit for texture space. Take a 2D texture as an example. When texturing a 3D surface, the OpenGL maps texels to appropriate pixels in the frame buffer. The texturing process starts from mapping the texture into the model space of the object surface where the texture is put on. Then along with projection of the object model, the texture is projected on the 2D screen space. As two coordinates of a pixel position on the screen are integers, the projected texture coordinates should be filtered. Then texels are generated. Sometime a texel can be outside of a texture, such as outside the border of the texture. Clamping or wrapping can be used. The former makes the texel outside the texture have the colour value of the nearest edge; the latter makes the colour value of texel outside the texture repeat the texture colour. Sometimes in the texture space, a value exactly corresponding to a texel cannot be found. In this case, '*GL-NEAREST*' or '*GL_LINEAR*' can be used. The former assigns the nearest value to the texel; the latter sets the texel to a linear-interpolated value.

5.1.2.3 Depth Test

Since all the images have to finally project on the 2D screen, for 3D objects, the vertices at the front can be visible whereas those at the back can be blocked and invisible. As regards this situation, it is necessary to do the depth test. When coordinates of a vertex are projected on the screen, two transformed coordinates, *x* and *y*, are corresponding to two dimensions of the screen. The third one, *z*, is taken as the depth coordinate directing into the screen, and can be stored in the depth buffer, or z-buffer, when the depth test is being done. The depth buffer is a two-dimensional array with one element for each screen pixel. If an object in the scene is rendered, the depth test of OpenGL compares its depth value with the value of the same pixel in the depth buffer. If it is less, which means the object is closer to the audience, it overrides the current value of the pixel. The new depth replaces the old one and is stored in the depth buffer. After the depth test, the hidden object will be culled and will not be rendered any longer.

5.1.2.4 Stencil Test

The stencil test is mainly used to limit the area of rendering with boundary patterns for a scene. The boundary patterns are stored in a stencil buffer, which is usually an array of bytes, one byte for each screen pixel. Combined with the depth buffer, the stencil buffer can be used to make many effects, such as outline drawing, shadows, and highlighting of intersections between complex primitives.

5.1.2.5 Alpha Blending

To draw a translucent object in a colourful surrounding, such as a glass bottle, alpha blending should be used. This function is to mix a translucent foreground colour with a background colour to produce a new blended colour. The alpha value can be ranged from 0 to 1.0, which means the range from completely opaque to completely transparent. At two extremes, zero is completely opaque and the blended colour is the foreground colour; one is completely transparent and the blended colour is the background colour. At other values, the blended colour is assessed by weighting the foreground and background colours with the alpha value.

5.1.2.6 Logical Operation

In OpenGL, a bitwise, logical operation of combining source and destination pixel colour values, such as and, or and exclusive or, is provided. The source pixels can be a copy of a block of pixels in the frame buffer, which can be read from the frame buffer by using OpenGL commands in advance. A logical operation can result in an effect on the destination pixel block in the frame buffer and make the pixel colours different from the original ones.

5.1.2.7 Frame Buffer

The frame buffer is the video output memory space that contains a complete frame of data, colour values of pixels of an entire screen. Each pixel can have one bit for monochrome, four bits for palettised, sixteen bits for highcolor, and twenty-four bits for truecolor formats. An additional alpha channel is sometimes used to retain information of pixel transparency. After the frame buffer is written, a graphics device will paint colour values on the device screen by some instruction, such as *glFlush*.

To attain a good performance, two frame buffers are usually provided in the graphics device. In this case, when the front frame buffer is displayed by the device, the back one can be written by the graphics pipeline. Then two frame buffers are swapped. So the video images can be played smoothly.

As the OpenGL implementation involves the image display and user interaction interface, there are always hardware-dependent and hardware-independent parts involved in the OpenGL pipeline. The upstream of graphics pipeline tends to be

implemented with software solution while the downstream is prone to require hardware support. Software solution is flexible, varied and slow, and hardware solution is fixed, designated and fast.

5.1.3 Rendering Flow of an OpenGL Interactive Application

For an object rendering, the processing starts from the input of object vertex coordinates at the upstream end of OpenGL API, performs the intermediate steps that the data are transformed and transmitted with a set of state variables, and ends by writing the pixel colours into a frame buffer to be displayed. The OpenGL state machine can practise the whole processing in a fixed order that follows the geometric and fragment pipelines and depends on whether an intermediate step is enabled or disabled, and finally make the object image display on the device screen.

From the application perspective, the OpenGL is a structure with several levels, as shown in Figure 5.3. From the top (interactive applications) down to the bottom (hardware platform), an application program is executed with an application, commands, geometric pipeline, fragment pipeline, hardware abstract layer, and hardware platform.

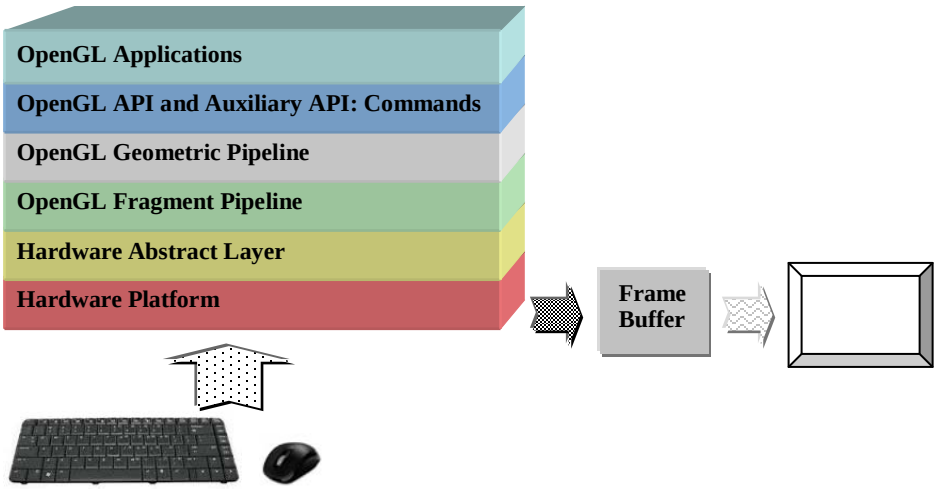


Figure 5.3 OpenGL Structure from the Application Perspective

In this project, the FPGA-based ES board is the hardware platform. The Altera HAL is the hardware abstract layer. Part of OpenGL fragment pipeline is merged into the hardware system, for example, the frame buffers, LCD, video pipeline, and buttons for user interactions. Part of OpenGL geometric pipeline is also constructed with hardware units, which is the algorithm-specified module as shown in Figure 3.6. These hardware parts are used to speed up the graphics pipeline. The rest of OpenGL graphics pipeline and OpenGL API are constructed with software to form the main part of Mesa-OpenGL on FPGA-based ES, which will be detailed in Section 5.5. The auxiliary API is composed of three supports that are of the Mesa-OpenGL API, ANSI C library and HAL API. The first

support will be expounded in Section 5.5.2. The latter two supports have been discussed in Section 3.3.4. The OpenGL applications are the surface modelling and editing, the core algorithm of which is the PAMA, which will be detailed in Chapter 7.

The general rendering flow of an OpenGL interactive application is:

a Open and initialise a window to draw OpenGL objects.

b Set any OpenGL state to a target value used on all the objects in the application.

c Register any event that the user may enter for the user interaction. The event can be pressing a key or button, moving or clicking the mouse, or moving or resizing the application's window.

d Draw the image of 3D objects by using OpenGL with values set by the user or defaulted of the OpenGL. The OpenGL state machine keeps a main loop running to catch any event and re-draw the variation of objects for the event.

5.2 OpenGL ES

The OpenGL ES is the standard for embedded accelerated 2D and 3D graphics (Angle and Shreiner 2008, and KHRONOS 2013). It provides a graphics API – a low-level interface between software applications and hardware or software graphics engines.

As the hardware conditions of ESs are more limited by their small sizes and designated applications than desktop computers, an implementation of OpenGL ES can be turned into one of subsets of the desktop OpenGL. The family of OpenGL ES implementations may have various versions for different pipelines and multiple hardware platforms, which may or may not be integrated with EGL (Native Platform Graphics Interface Layer, detailed in Section 5.3.1). Each implementation can be a much smaller engine with few function calls.

Compared with the desktop OpenGL, the archive of OpenGL ES includes profiles not only for floating-point systems but also for fixed-point systems. It also provides the EGL specification for portability between different native windowing systems. As an implementation of the OpenGL ES has a closer bind with hardware devices than the desktop OpenGL, the porting of OpenGL ES from one platform to another is more difficult and brings about more detailed issues for transplanting.

5.2.1 Versions and Profiles of OpenGL ES

The OpenGL ES has several subsets of OpenGL standard for ESs. It is divided in two ways, versions and profiles. The former is for different ways in the programming; the latter for different dimensions on the footprint.

5.2.1.1 Versions

Since the graphics requirements have a broad range of 3D devices and platforms in embedded markets, the OpenGL ES has been developed two versions with two different roadmaps, which are for two different development requirements and platforms and have evolved in two different directions, respectively. They are OpenGL ES 1.X and 2.X, both of which have their counterparts in OpenGL.

The OpenGL ES 1.X is for fixed function hardware. Its goals are to enhance hardware acceleration, to improve image quality and performance, and to reduce the memory bandwidth. It is also a software interface that consists of a set of procedures and functions. The function calls can be used not only to implement rendering but also to support transformations, matrix stacks, Phong lighting, fog, and others. The programmer can specify a state by calling a specific function to control the state machine in producing graphical images of 3D objects. It can be applied to the new-generation fixed function 3D accelerators.

The OpenGL ES 2.X is for programmable hardware. It defines a programmable 3D graphics pipeline to create shader objects and to write vertex and fragment shaders in the OpenGL ES Shading Language. Since almost all graphics operations are done in shaders, its interface is very small, and focuses on specifying geometry and textures, and dealing with shaders and their variables. It can be applied to the emerging programmable 3D pipelines.

Even though the OpenGL ES 1.x is more fixed in functions compared to the OpenGL ES 2.X, it does not mean that the use of functions of the OpenGL ES 1.x is fixed during programming applications. There is still a lot of room to adjust and optimise the order of function calls and disable some unnecessary modules in the pipeline during programming to speed up the object rendering and save the memory space.

5.2.1.2 Profiles

The definition of profiles of the OpenGL ES specification is dependent on their application fields. Each profile has its own emphasis, adopts part of the desktop OpenGL specification, and extends its own OpenGL ES-specific part as well. For the part derived from the OpenGL, each OpenGL ES profile retains the similar API, processing pipeline, command structure, and the same name space as the OpenGL. For the extension part, it adds OpenGL ES-specific functions to the OpenGL specification in order to meet the needs of embedded platforms. Three profiles have been issued. They are Common, Common-Lit, and Safety Critical Platforms. As the family of OpenGL ES profiles still grows, along with new technologies emerging in the ESs, more profiles may be issued in the future.

Since it is designated for consumer handhold devices, the Common Profile provides the

market support for platforms that have different computation and memory abilities, such as full-functioned and texture-mapping 3D graphics with minimum footprint, gaming platform, and cell phone platform. The Common Profile supports the floating-point computations.

Because it focuses on a simple class of graphics system with a small footprint and only supports the fixed-point calculations, the Common-Lite profile does not support high-performance OpenGL functions that need the floating-point computations. The Common and Common-Lit profiles not only share many commands, but also have some commands designated to only one of them.

The Safety Critical Profile has high requirements on being reliable and certifiable. It can meet the needs of 3D graphics applications in avionics and automotive displays, and safety certifications.

5.2.2 OpenGL ES Implementations

Since the OpenGL ES is derived from the OpenGL, many resources and development experiences of OpenGL can be applied to the OpenGL ES. These make it feasible a crossing-platform migration from desktops to different embedded platforms. The OpenGL ES also makes it affordable to provide the diversity of 3D graphics and games in the mainstream of embedded and mobile platforms. According to KHRONOS 2013, the OpenGL ES has had many implementations that are of some companies, such as Intel, Imagination Technologies, QUALCOMM, ARM, Marvell, Apple, NVIDIA, Creative Technology Ltd, Media Tek Inc, and NOKIA OYL.

5.3 Different Roles in OpenGL ES

Since part of the work in this project is to implement the OpenGL ES on the FPGA-based ES platform, it is very important to have a proper perspective to view the OpenGL ES. Especially, the role of the author has to be turned between an OpenGL implementer and application developer when programming the applications to test the effect of integrating the Mesa-OpenGL in the FPGA-based ES. Therefore, there are three roles that should be considered in the OpenGL ES. They are graphics application developers, implementers, and standard setters.

For the OpenGL standard setters, after the needs of general graphics application developers, the conditions of general graphics platforms, and how to implement the model on a general graphics platform were known, the OpenGL ES standard structure should be built up to meet general graphics application developers' needs and satisfy the conditions of general graphics platforms.

Before starting implementing OpenGL ES on the graphics devices, it is necessary for

OpenGL implementers to know the specification of standard OpenGL ES. If so, the implementation of OpenGL ES can be used by application developers of OpenGL ES in a right way.

The graphics application developers should understand the OpenGL ES specification and how its implementation at the local platform to work. Although all relative researchers try very hard to make OpenGL hardware-independent, it is inevitable to involve the hardware part because of the wide species of hardware platforms in ESs. The benefit of this is to make the best use of OpenGL ES to realise their applications and attain the best performance of the OpenGL ES and graphics platform.

5.3.1 OpenGL ES Standard Setter Role

In the view of OpenGL ES standard setter role, the OpenGL ES is treated as a structure model and state machine that controls a set of specific operations on drawing (KHRONOS 2013). The structure model should yield a specification that satisfies the needs of both developers and implementers. It does not provide a model for the implementation which must bring about the specified results. The ways adopted by different implementations can be different and have different efficiency for a particular computation. One of the main goals of this specification is to define the OpenGL ES state information, and to explain how it changes and what its effects are without ambiguity.

5.3.1.1 Command Descriptions

For a graphics command, the specification gives an explicit description of its function and interface so that an application developer can correctly use the command and an implementer can implement completely the corresponding operation or computation in its function.

5.3.1.2 Profile Definition

Each profile has its own designated definition of header files, tokens, and command library. To simplify the maintenance of a single profile, some conditional pre-processing directives can be defined in the header files, which control and show which profile is used and indicate which profile runs when an application inquires it by using the OpenGL version query.

5.3.1.3 Minimised Footprint

Compared to the desktop computers, ESs have smaller memory space and slower microprocessor speed. In ESs, the memory space ranges from 1 MB to 64 MB and the speed of microprocessor varies from 50MHz to 400MHz. These data are still changing. To meet these requirements for different conditions of embedded hardware, the OpenGL ES must be implemented with a minimum footprint. It means minimising the instruction and

data storage and traffic requirements. It also means that the floating point computation may be limited dependently and the integer computation can be encouraged mostly. The power consumption can be lowered as well. Furthermore, the size of binaries of an OpenGL ES implementation is so small that it is convenient for users to download it to devices.

In this project, since the FPGA-base ES has a limited memory space, which is a memory of 2 X 64 MBytes, it is very important to minimise its footprint. The PAMA takes this issue into account during the algorithm research, which will be detailed in Chapter 7. The implementation of OpenGL for the FPGA-based ES must balance between the freedom for shape modelling and editing and the storage space requirement. On the one hand, it has to add the complex algorithms to the OpenGL implementation in order to support the graphics applications for surface modelling and editing, which results in enlarging the footprint; but on the other hand, it must decrease the total requirement on the storage space during the software programming in order to gain a proper system performance.

5.3.1.4. Invariance in Images

As the hardware platforms and display devices are different, the OpenGL ES specification does not require matching exactly among implementations on different hardware platforms. However, in some cases, images produced by the same implementation have to match exactly in order to make a judgement in some situations, such as making a comparison between two images.

In another case, invariance is necessary. A complicated object is drawn, which requires a whole sequence of operations that may involve many algorithms to support. These algorithms may be executed many times for the object rendering. It is expected that whenever the rendering is executed, the image displayed in a window should be invariant.

Invariance, however, can be costly. It can significantly increase the complexity of implementation of the OpenGL ES and greatly limit the parallel processing ability of the OpenGL ES. In fact, speeding up and a high image quality can be contradictory. An implementation of OpenGL ES should balance all the requirements to implement a relatively complete OpenGL ES profile. It is not acceptable to make one aspect very strong at a price of sacrificing other abilities. On the other hand, among so many choices in the ES platforms, a strong invariance requirement on the identical behaviour of the hardware and software modules may be too restrictive to meet. A low requirement on the invariance behaviour can make a lot of different implementations of OpenGL ES flourish. In addition, a great deal of software tools and hardware platforms are disposed to accepting the OpenGL ES. It may be one of reasons why there are more profiles in the OpenGL ES than in the desktop OpenGL.

The OpenGL ES should be able to execute on a wide range of graphics platforms that may have different graphics capabilities. To support this diversity, for an OpenGL ES operation, the ideal behaviour, rather than actual behaviour, is designated. Therefore, it is possible that the behaviour of an implementation is an approximation to or deviation from the ideal one. Two different OpenGL ES implementations are allowed not to agree pixel for pixel even though they have the identical input and frame buffer configurations. In this way, different implementations can be developed in order to fit into different application situations and to meet different system requirements on different platforms. For example, the requirements on invariance of two images are quite different between the image identifying and shape changing of a moving character in a game. The former should meet the higher requirement on invariance; the latter can get a benefit of speeding up the graphics computation from a lower requirement on invariance. Because of the computation of 3D graphics in surface modelling and editing in real time, speeding up is crucial for the PAMA. There is not a high requirement on invariance for PAMA since it is not necessary to compare two images. This has been taken into account during this research. For example, a fixed point arithmetic system is adopted for this reason in this project. It will be discussed later in this chapter.

5.3.1.5 Repeatability of Command Results

Given an OpenGL ES implementation, frame buffer, and hardware platform, the result of any command must be identical whenever the command is executed. The reason is that when double buffers are used, subtle difference between results of a command used in twice executions can lead to visual difference between two frame buffers when they are swapped for rendering with the same command sequence. This difference makes the image look artificial or the viewer misunderstand it as the image trembling. In addition, the difference of the same command sequence at different moments means that the implementation is not stable during testing.

It is obvious that repeatability is very important in many cases, but not always. For example, in changed view angles, a small primitive can be viewed differently. If the small primitive is just a small portion of a drawn object or a whole scene, an acceptable variance can be allowed when considering the cost and importance of trying to eliminate the variance. In fact, the requirement of accuracy on an image drawing is not as important as that on the pure computation. The reason for this is that the accuracy of a result of computation is usually examined by a tolerance range that is defined for the computation. It mostly depends on the audience's visual impression whether or not a displayed image is satisfactory, however. The computation accuracy can be stored to be measured and compared later. The visual accuracy of image-drawing is instantaneous and varied for different viewers. In addition, some technologies are sometimes used to yield a special visual impression in order to satisfy the viewer's eyes. For example, the antialiasing

technology is usually used in computer graphics and others to give the audience the visual impression of the smoothness at a corner or boundary, but strictly speaking, it produces only a beneficial illusion for the audience.

Thus, it is often sufficient for an image to convey the meaning that it must deliver. As mentioned in the previous section, the comparison between two images sampled in real world should be more accurate. For surface modelling and editing, the PAMA is realised for the similar goal as the meaning's conveying, and it has equally an important requirement on speeding up for the user interaction in real time.

5.3.1.6 Extensions

As the OpenGL family and functions keep increasing, it is necessary and important for the OpenGL ES to keep open and make any extension to the OpenGL ES profiles unimpeded.

Extensions to the OpenGL ES are treated in two ways. In one way, the extensions have a strong functionality and will be included into the core profile revisions in the future. In the other way, the extensions are still valuable but may not belong to the mainstream of OpenGL ES.

The OpenGL ES allows the implementations to add new features to the OpenGL ES, which may not belong to any existent profile. As mentioned in Section 5.2.1, an existent OpenGL ES profile is composed of two sections. One is a subset of the full OpenGL graphics pipeline. The other is the extension of OpenGL ES-specific functions, which may turn to a core addition to OpenGL in the future, or be collected in another set, the set of OpenGL ES-specific functions that is not suitable for the general OpenGL. The extension set of OpenGL ES-specific functions are further divided into two groups, the required and optional.

In any new extension, the commands and tokens should match the corresponding command subsets. Commands and tokens in the core addition subset do not have extension suffixes in their names whereas those in the OpenGL ES-specific set should contain the profile extension suffixes. The required part must conform to the existent profile implementation. The optional part can be defined by the implementer.

Following these rules and mechanisms to extend programming, new emerging hardware can be accessible through the OpenGL ES API and the OpenGL ES API can be upgraded as well. When they are accepted broadly, the new extensions can be merged into the OpenGL ES core standard. Furthermore, OpenGL ES, itself, can evolve innovatively and controllably. Practically, in the process of implementation and development, the OpenGL or OpenGL ES structure model can be further developed and extended without separation from the original OpenGL core.

All the above allows this project to add functions of surface editing and API for the

FPGA-based ES to a new implementation even though there is not an existent profile of OpenGL ES for FPGA-based ES. For example, the algorithms of Bézier curves and surfaces are not included in the specifications of the general OpenGL ES. But they are necessary for the surface modelling and editing with the PAMA. Thus, they are added to the new implementation of OpenGL for the FPGA-based ES in this project.

5.3.1.7 Native Platform Graphics Interface Layer

As shown in Figure 5.3, there is a layer above the hardware platform, which can provide the device-dependent functions, including device drivers, memory allocation and deallocation, memory access, frame buffer access, and others.

Some functions of OpenGL ES are device-dependent, for example, drawing objects like points, lines and polygons. These functions require writing a frame buffer of the graphics hardware. For this reason, they are concerned with frame buffer manipulation. They need some functions to bridge between writing the frame buffer and drawing the objects. These function implementations are varied with graphics device specifications. Other functions, like antialiasing or texturing, if they are enabled, can also influence how to draw objects. They must be processed before writing the frame buffer.

To minimise and isolate the device-dependent part from the application developers and make it portable crossing the hardware platforms, the OpenGL ES establishes a specification of a common platform interface layer, called EGL (Native Platform Graphics Interface Layer). It is an independent platform interface. The OpenGL ES implementers can make a choice between constructing a native hardware platform interface based on the EGL and defining their own platform-specific embedding layer in their implementations. The former's benefit is that the implementers do not necessarily consider the consistency of the platform interface with other part of OpenGL ES because an associated conformance test is provided along with the EGL.

For the application developers, the hardware-independence brings them a seamless transition from software to hardware and from one piece of hardware to another. Although the OpenGL ES specification is defined as a hardware-independent graphics processing pipeline, the implementations of OpenGL ES must be caught out by a combination of software and hardware. The commands call the software routines that finally run on the system microprocessor with the system memory access. Following the OpenGL ES specification during programming can make applications and implementations seamlessly connect to the hardware devices. It means that developers can get a normative software engine no matter which implementer completes it, and users can download applications to their device gadgets and play them no matter which developer develops them.

5.3.2 OpenGL ES Implementer Role

For an implementer, the OpenGL ES is a set of procedures, or functions that carry out all the computations in the graphics pipeline and all the operations of graphics hardware.

Imagine a graphics hardware device on which the OpenGL ES is implemented. If the graphics hardware only had an addressable frame buffer, all the routines of OpenGL ES should be implemented and executed on the host CPU. In reality, the graphics hardware can be composed of varied graphics acceleration units, ranging from a simple raster subsystem that can render 2D lines and polygons to high-end floating-point processors that can transform and compute the geometric vertex data.

The OpenGL ES implementer's task is to implement the programming of each OpenGL ES command procedure. At the same time, they must provide the CPU software interface for the application developers. They have to know how to divide the work of each procedure between the CPU and the graphics hardware. The former is software-relative and logical; the latter needs the device drivers' support to implement OpenGL ES calls. The latter also must be customised to fit into the available graphics hardware units and obtain the optimum performance.

Thanks to the open and widely-accepted features of OpenGL and OpenGL ES, there are numerous resources available and free on the Internet, which can speed up an implementation on a new hardware platform and mitigate the programming work without building it from scratch. Even better, without too much explanation about the new implementation, the application developers of OpenGL ES can understand how to develop their applications with it.

As an implementer needs to test whether or not the implementation is successful, it is necessary to stand in the shoes of application developers during the programming and verify the implementation in applications.

In this project, one role of the author is the implementer of OpenGL ES because a new implementation of OpenGL ES must be constructed on the platform of FPGA-based ES. In the discussion of next section, we will see that the other role of the author is the application developer of OpenGL ES.

About command implementation details will be introduced in Sections 5.4 and 5.5.

5.3.3 OpenGL ES Application Developer Role

As the OpenGL ES is an open embedded graphics standard, anyone can download the OpenGL ES specification from the website of KHRONOS Group (KHRONOS 2013) and learn it. With its broad and crossing-platform support in the industry, many individuals and companies implement products based on OpenGL ES standard. The standardised

abstraction and device-independence at the high level make any developer just focus on the function commands without considering the details of hardware platforms and code implementations.

Because the OpenGL ES is derived from the OpenGL, a great deal of relevant information, documents, books, and sample code of OpenGL ES can be to find on the Internet. For developers having the experience of programming with the OpenGL, it is readily to learn the OpenGL ES and write the OpenGL ES applications with the similar structure of the design and logical commands as those in the OpenGL.

For the application programmer, the OpenGL ES is a set of commands. Some of them define the specification of geometric objects in two or three dimensions. Some of them control how these objects are rendered into a frame buffer. Others provide an immediate-mode interface to specify an object, which causes the object to be drawn.

5.3.3.1 State Information

The OpenGL ES maintains a considerable amount of state information. The state information controls how objects are drawn into a frame buffer. Some state information is directly available to the application developer. The developer can use some commands to make calls to obtain its value. Other state information is not, but its effect on what is drawn can be visible. For example, the *glEnable(GL_DEPTH_TEST)* command can enable the state of depth test and the OpenGL state machine will load and execute depth test in its execution loop and create the scene with 3D effect.

5.3.3.2 OpenGL ES Application Flow

The programmers should understand the flow of an OpenGL ES application and know which part is computation-intense and which part is input/output-intense. During programming, they can find strategies for these parts so as to draw objects as fast as possible.

In the flow of an OpenGL interactive application, a typical program starts from creating a window to draw objects by using the window API commands. A frame buffer (or two frame buffers if double buffers are available) and other context are created and connected to the window. Once a context is allocated, the initialisation for lighting and background colour cleaning should be done by using the OpenGL ES commands. With the window API commands, some user procedures, such as drawing objects, reshaping the window and issuing user interaction events, can be registered to enable the OpenGL ES to execute them in the main loop.

In the user procedure of drawing objects, OpenGL ES commands can be used to draw specific geometric objects. Some commands are used to draw geometric objects, including points, lines, and polygons. Some commands have influences on the rendering

of these primitives. For example, they may decide how an object moves away from the view point and how the object image of the user's 2D or 3D model space is mapped to the 2D screen space. Others perform directly operations on the frame buffer, such as reading pixel colours from the buffer.

In the user procedure of reshaping the window, OpenGL ES commands can set the model view, projection view and mode, and resize the window. The operations acting on the scene world should be performed in this procedure because they have influences on all the objects in the scene world.

In the user procedure of issuing user interaction events, some events that make responses to some actions done by user interactions through the standard input, like pressing a key or button, moving and clicking the mouse, and touching the screen, should be programmed here.

5.3.3.3 Global Effect vs. Local Effect

For the application developer, it is necessary to know how to distinguish commands that act on the global scene world from those that have an influence only on individual objects. The commands that act on the global scene world where all the objects exist should be performed outside or before the display loop specified for individual objects. In this way, the processing time can be decreased to a low level.

In this project, since the PAMA is the graphics application that is the second development on the new implementation of OpenGL ES, the author is also the application developer of OpenGL ES.

5.4 Mesa-OpenGL

For this project, there are three reasons that promote to create a new OpenGL implementation that can be played on the FPGA-based ES.

The first reason is that the OpenGL ES takes into considerations only the part of graphics pipeline for writing to or reading from a frame buffer. There is no specification for how to paint on the display screen and how to interact with other peripherals associated with graphics hardware, such as mice and keyboards. Implementers must rely on other technologies, such as the Khronos OpenKODE API, to drive the screen painting or obtain the user input. Different hardware platforms can have different drivers for their own input and output devices, however.

The second reason is that Altera development and compiler tools, Nios II IDE (Integrated Development Environment) and Nios II Software Building Tools for Eclipse, are more skilled in software connection to an FPGA design but more restricted in C

programming and compiling than a general C-specific development environment. The surface modelling and editing need a great deal of flexibly and complicatedly programming and a lot of computation and storage space for processing data.

The last reason is that the implementations of existent OpenGL ES profiles cannot fit into this project. In the introduction of Chapter 2 and Section 5.2.1.1, we have observed that since GPUs have attracted most attentions of computer graphics researchers, GPUs have attained a lot of support from the computer graphics society. One of their benefits is that the OpenGL ES 2.X has adapted to the trend of GPU development. On the other hand, the algorithms of Bézier curves and surfaces are excluded from specifications of three profiles mentioned in Section 5.2.1.2, which are Common, Common-Lit, and Safety Critical Platforms. In addition, FPGA-based ESs are an emerging branch in ESs, and they still need some time to develop in various application fields, especially in graphics applications.

Because of the above reasons, a new Mesa-OpenGL implementation must be made for FPGA-based ESs. More functions should be added to the new implementation for this project than general OpenGL ES profiles. The results of Chapter 8 will show that the new implementation has been ported to the FPGA-based ES platform successfully.

5.4.1 Introduction of Mesa OpenGL

Mesa (Paul 2013) has a sequence of open-source and evolutionary implementation models of the OpenGL specification. Originated from 1993, the Mesa project was started by Brian Paul. It has been evolving in twenty years.

There are a variety of hardware platforms that use the Mesa in various environments ranging from software simulation to hardware acceleration for GPUs. For example, the hardware drivers that the Mesa supports include Intel i965, i915, AMD Radeon, and NVIDIA GPUs. The operating systems that the Mesa supports consist of Linux systems, Microsoft Windows, UNIX, and Haiku. The Mesa did not have an implementation for a FPGA-based ES before this project, however.

5.4.2 General OpenGL Implementation

The following are important elements that must be implemented in the OpenGL graphics pipeline, called I-GL in this thesis, which represents Implementation of OpenGL. According to their inner relationships, the introduction is from the top of commands to the bottom of the connection between software and hardware.

5.4.2.1 Commands and Orders

The I-GL provides the API of graphics pipeline for application developers with a set of commands. These commands can be used to specify a primitive, set the mode, or perform

an operation. Commands can be called with arguments. These arguments are used to transfer data to the settings of variants, such as primitives and modes. The data are bound to the current context and are transferred to the process that the command executes at the moment when the command is called. The subsequent changes on the data will not have an influence on the current pipeline.

As the OpenGL graphics pipeline is a state machine, commands in the I-GL are always executed in an order that they are received although it can be delayed for a command to really take effect. For example, an enabled texture can be delayed to be put on the surface of an object until the object is drawn, but the enable operation has been performed at the point where the *glEnable* command is used. This delay relies on the programming inside the I-GL and is a meaningful strategy for programming that allows all the similar operations to be done collectively and enhances the performance of the I-GL.

In general, the relationship between commands used in an application program and the I-GL is client-server. For a simple platform with a small amount of memory and CPU resources and single-threaded programming, such as small-footprint ESs, the relation between an application program and I-GL can be treaded as a caller and a set of called procedures. The application program (the client or caller) calls commands, and these commands are interpreted and processed by the I-GL (the server or procedure service). That is, the application program should issue commands to require the I-GL to create a context for one drawing task and connect this context to the task at the initialisation stage. If the program is not connected to a context with a complete initialisation in advance, calling commands can cause undetermined behaviour. A server can manage several contexts concurrently, each of which is corresponding to the current pipeline state for a drawing task. A client can choose one of the contexts to be connected.

5.4.2.2 Primitives and Modes

In the I-GL, a primitive can be a point, line segment, or triangle. A primitive can be drawn in several selectable modes. Each mode is independent. The setting of one mode does not influence others, but their effects may have a synergy in the frame buffer. Primitives and modes are set with the arguments of commands.

In the I-GL, every primitive is an entirety and must be drawn completely before any subsequent one can influence the frame buffer. Also, the effect of one mode on the frame buffer must be complete before any subsequent mode setting can have such effect.

The I-GL does not provide a complete command for constructing complex geometric objects, however. To describe a complicated geometric object, the surface of the object must be broken down into patches that can be represented by the basic primitives. With a whole sequence of commands of describing primitives and some advanced I-GL mechanisms, the object can be modelled. That is, the I-GL only provides commands and

mechanisms how complex geometric objects are rendered. It is the responsibility of application developers to realise and figure out how the complex objects are described.

Application developments should think about how to divide a complicated-shaped object into primitives and divide a complicated process into simple operations, and how to link them together in order to render an entire object in a feasible and effective way. They also must realise what the effect of a sequence of operations on the frame buffer is. They have to re-test it several times and make sure of it.

5.4.2.3 States

For the I-GL, each state should put its data into the specified location of the memory. The evaluation function should be launched after all the required data are put into their specific locations. The evaluation function does just its tasks including fetching the data from the assigned location, doing operations, and putting its result into the result's specified location. All the locations in memory have been initialised to allocate to the context of a drawing task when an application program makes a requisition for a window initialisation. This initialisation is the starting of the drawing task.

In the I-GL, there are numerous states that can be divided into two groups, server states and client states. Most states are server ones. A connection from a client to a server needs both the client state and server state, and the transfer between them. An operation in the pipeline, which may be primitive specifying, mode setting, or data evaluating, is a server state. A command starts from a client state and then the processing is transferred to server states when the I-GL processes the command.

5.4.2.4 Controls

From the view of application developers, the commands and mechanisms of the I-GL provide the direct control over the fundamental operations of 2D and 3D graphics drawing. Inside the I-GL, this control can be treated as the states of the OpenGL state machines that drive a drawing task to be executed and completed step by step. This control consists of commands with parameters of primitive setting, view point specifying, transformation matrices, lighting and material equation coefficients, texture mapping, antialiasing methods, pixel update operators, and others. Therefore, the entire efficacy of the I-GL relies on two aspects, how well the I-GL is accomplished and how well the I-GL is used in the application programming.

5.4.2.5 Data and Orders

Geometric objects with different shapes and styles may be drawn in many primitives, such as points (including point sprites), separated line segments, line strips (connected line segments), line segment loops (connected line segments with met endpoints), separated triangles, triangle strips (triangle patches arranged in a long strip and stitched seamlessly

together), and triangle fans (triangle patches arranged in fan and stitched seamlessly together). Primitives are defined as an array of one or more vertices with coordinates. A vertex can be the coordinates of a point. A point can be a single point, an endpoint of an edge, or a corner where two edges of a triangle or other polygon meet.

Besides the positional coordinates, data of a vertex consist of colours, normals, and texture coordinates. Each group of data with the same attribute is processed independently in the same way and order. There is an exception, however. If an object cannot fit into a viewing volume, some of its primitives, such as points, lines or triangles, must be clipped. Vertex data can be modified. In the line or triangle primitives, some new vertices can be inserted into primitives. The new associated data, such as colours, normals, and texture coordinates, must be computed or interpolated.

Typically, data of colours consist of four values, representing red, green, blue and alpha values, respectively. Each of them is initiated as one (with values within *[0, 1.0]*; *(1.0, 1.0, 1.0, 1.0)* means white and completely transparent). Data for normals have three values, representing three coordinates of normal vectors. They are initialled as *(0, 0, 1.0)*, represented a vector directing in *z* axis. Texture units have four values that represent texture coordinates of *s, t, r,* and *q*. For a 2D texture, the *s* and *t* coordinates are used, and the two others are not. *[s, t, r, q]* is initialised as *[0, 0, 0, 1.0]*, defined as homogeneous coordinates.

Vertices of an object are organised in a vertex array. All the associated data are copied in the vertex array and the transformed data of the object are also storied in this vertex array. The transformed data consist of eye, clip, normalised device, and window coordinates, which will be detailed in Section 5.4.2.7. With the window coordinates, vertices of a primitive can be written in the frame buffer with the colours or the texture colours according to the texture coordinates.

5.4.2.6 Vertices

In the I-GL, a data structure is defined specifically for a vertex array. At the stage of the user's inputting data of an object, the vertex data are placed in the user or client address space. Once a command to transmit the data is used, the vertex data are transferred to the I-GL or server address space.

Blocks of data in these arrays can be specified for vertex coordinates, normals, colours, point sizes, and one or more texture coordinate sets of different geometric primitives in the execution of an I-GL command. For vertex coordinates, the command of *glVertexPointer(size, type, offset, pointer*) is used. For normals, it is *glNormalPointer(type, offset, pointer*). For colours and texture coordinates, they are *glColorPointer(size, type, offset, pointer*) and *glTexCoordPointer(size, type, offset, pointer)*. All of them have the similar argument structure that gives the information of locations and organisations of

these arrays: *size* represents the number of components of the corresponding data stored in the array, *type* represents the data type of the values stored in the corresponding array, *offset* is the number of bytes for a set of components of the corresponding data, and *pointer* means the pointer pointing to the location of the array of corresponding data in the user or client address space. The *pointer* is the location in memory of the first component of the first element in the corresponding array. Some arguments are not present because they are constant for their data array. *Size* and *type* have their allowable values in OpenGL. For example, the acceptable values for type can be *BYTE*, *UNSIGNED BYTE*, *SHORT*, *FIXED*, and *FLOAT*.

All of them can transfer the corresponding array from the user or client address space to the I-GL space. The data in arrays are stored sequentially. The array elements are stored sequentially even though the offset is set to zero. No matter how many values an array element consists of, from one to four values, a single vertex covers an array element. The values in each array element are stored in memory in a consecutive manner as well. The number of values of each array element defines the offset between two adjoining elements. In an array, the pointer to its (i + 1)st element is greater by one offset number of machine memory units (say unsigned bytes) than the ith element.

5.4.2.7 Coordinate Transformations

In the graphics pipeline, there are several coordinate transformations. Data of vertices, such as, vertex coordinates, normals, and texture coordinates have to be transformed before the coordinates are written in the frame buffer to produce an image.

In Figure 5.4, the general sequence of transformations applied to vertices is shown. All the matrixes are 4 X 4. All the coordinates are four-dimensional, including *x, y, z,* and *w* coordinates.

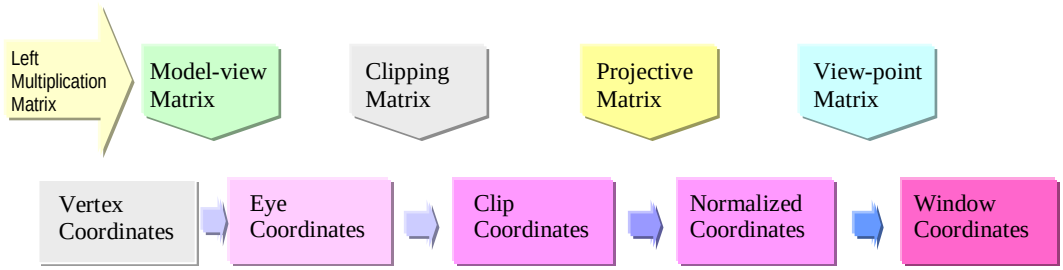


Figure 5.4 Coordinate Transformation Sequence

The transformation sequence in Figure 5.4 has an inherent relationship with the geometric pipeline in Figure 5.1 and Section 5.1.1. In the I-GL, the coordinate transformation sequence starts from the transformation from the vertex coordinates of the drawn object to eye coordinates by left multiplying with the model-view matrix. This corresponds to the transformation from the local coordinates to the world coordinates in

Section 5.1.1. The transformation from eye coordinates to clip coordinates is carried out by left multiplying eye coordinates with the clipping matrix. It is the function of the clipping module of Section 5.1.1.4. The transformation from clip coordinates to normalised coordinates is done by left multiplying clip coordinates with the projective matrix. The last transformation from normalised coordinates to window coordinates is done by left multiplying normalised coordinates with the view-point matrix. The last two transformations are the function of the projection module in Section 5.1.1.5. In window coordinates, *x* and *y* are for the location coordinates of two dimensions on a display screen, and *z* represents the depth of the object looked into the screen by the viewer.

As mentioned above, the data are put into the specific location of memory when a command is called, and the evaluation should start after all the required data are prepared. The matrix settings with commands of *glMatrixMode* and *glOrtho* can be scattered in different parts of the application program, which depends on the application. The order of the matrix settings can influence the effect of the object rendering. The evaluation of all the transformations in a complete sequence is really done at the final stage of the geometric pipeline just before the rasterisation in Figure 5.1, however.

5.4.2.8 Colours

The colour processing is also done before rasterisation. There are two sources of colours: one is the colours of a primitive; the other is the colours of a texture. Typically, colours have four components of red, green, blue and alpha. Following the OpenGL specification, the I-GL accepts each colour component with a value ranged in *[0, 1.0]*. The colour can be set by *glColor* command at the beginning of a drawing task.

On the other hand, there are many colour palettes in the LCD and screen space. They include three bytes in whole and eight bits for each red, green, and blue, two bytes in whole and five bits for red and blue but six bits for green, and others. In this project, the LCD device adopts three bytes in whole and eight bits for each red, green, and blue. As the input colours can be in a different format, the I-GL must do the conversions. As a result of limited precision, especially for the hardware platforms of simple ESs, some converted values cannot be represented exactly. Different implementations on different platforms can yield discrepancies in colour tones and shades.

5.4.2.9 Lighting

If lighting is disabled, the current colour is used in the subsequent drawing. If enabled, lighting can influence the colours of the drawn object. A light can be set ambient or position. The former produces a far ambient lighting effect; the latter yields a near spotlight effect. The material reflection can be set as diffuse, specular or shininess. With the material setting, the lighting effect can be evaluated with lighting equations and yield new colour values. The produced values of colour components must be clamped to the range

of *[0, 1.0]*.

The shade mode can also influence the colours of a primitive. If the shade mode is set to flat shade by the *glShadeMode(GL_FLAT)* command, all vertices of the primitive are to have the same colour. If the shade mode is set to smooth shade by the *glShadeMode(GL_SMOOTH)* command, the colours of the primitive are produced by smoothly interpolation.

5.4.2.10 Texture

In the I-GL, texture functions can insert a texture pattern in a spatial region that may be a section of line or a patch of surface. A texture source can be an image described as an array of colours in one, two, or three dimensional coordinates.

Take a 2D texture image as an example. The texture image consists of a sequence of groups of values. Each group represents the set of values of red, green, blue and alpha components. The first group is the colour value at the lower left corner of the texture image. Subsequent groups are arranged firstly in width and secondly in height. In a width row, the next colour with the coordinates of *(i, j+1)* is located just after its previous one with the texture coordinates of *(i, j)*. In a height column, the offset between the next colour with the coordinates of *(i+1, j)* and its previous one with the texture coordinates of *(i, j)* is the value of width of the texture image. The texture coordinates are ranged in *[0, 1.0]* in default. Therefore, the coordinates of a texture image have to be normalised in *[0, 1.0]* before texturing an object.

The I-GL provides a set of functions to perform the operations of texturing. The *glTexImage2D(tTarget, rv, pMode, tWidth, tHeight, bd, tFormat, tType, tArray)* can be used to load a user 2D image to the texture space that the I-GL allocates to the context of a drawing task while the *glTexImage1D()* is used for 1D texturing. In the command, *tTarget* shows the dimensions of a texture, for instance, *GL_TEXTURE_2D* indicating a 2D texture and *GL_TEXUTRE_1D* representing a 1D texture; *rv* is an indicator for a reduction version of a larger texture array; *pMode* is the style of colour pattern of the texture image and usually adopts the GL_RGBA style with four values for red, green, blue and alpha; *tWidth* and *tHeight* are the numbers of columns and rows of the source image array; *bd* is an indicator for having a one-pixel border around the texture image; *tFormat* can specify a monotone of single red (*GL_RED*), green (*GL_GREEN*), or blue (*GL_BLUE*), or a colour in the order of blue, green and red (*GL_BGR*); *tType* is the data type for the colour values in the texture image array; *tArray* is the pointer pointed to the location of the source texture image array. The texture image is stored in a *4 X tWidth X tHeight* array.

Since the *tWidth* and *tHeight* are the numbers of columns and rows of the source image array, they must be set to numbers that are powers of two. With a one-pixel border around the texture image, these numbers have to be the sum of two plus powers of two.

When a texture has to be contiguously mapped several regions on a surface or a line, the boundaries of different texture copies may not align with the positions of pixel boundaries. The *glTexParameter(tTarget, tAction, tManner)* can be used to set operations that can be done on the texture. In this command, *tTarget* has the same meaning as that in the *glTexImage*; *tAction* has selectable values to set different operation actions that can be done on the texture; *tManner* defines which way is chosen to do the operation action specified by *tAction*. For example, given *GL_TEXTURE_MAG_FILTER* set to *tAction* and *GL_NEAREST* to *tManner*, the operation action is to enlarge a section of the texture image to fit a specified coordinate range in the region of a primitive with the nearest colours. Oppositely, the *GL_TEXTURE_MIN_FILTER* means to reduce the texture image. The tManner can get values of *GL_NEAREST* (for nearest) or *GL_LINEAR* (for linear), as mentioned in Section 5.1.2.2.

Sometimes coordinate values in texture space are outside the range from *0* to *1.0*. The *glTexParameter* can be used to make up for the patterns. In this way, *tAction* indicates a coordinate on which the compensatory pattern will be done, such as *s* coordinate with *GL_TEXTURE_WRAP_S* and *t* with *GL_TEXTURE_WRAP_T*; *tManner* has selectable values, such as *GL_REPEAT* (replicating with the fractional part of a texture coordinate) and *GL_CLAMP* (clamping a texture coordinate to the unit interval).

The *glTexEnvi* command can control how texture elements are applied to an object. According to the argument values of *glTexEnvi*, the I-GL can replace the object colours with texture values directly (*GL_REPLACE*), or combine the current object colour components with texture values (*GL_BLEND*).

Take a 2D texture as an example. If the texturing is enabled, the texture colour at the location indicated by its coordinates will replace or blend with the colour at the corresponding location in the region of the object surface. Texture mapping can apply more than one specified image to a primitive at a time. In this case the fragment has multiple sets of texture coordinates *(s, t),* which are used to index separate images and to collectively modify the fragment's *RGBA* colour. Multiple texture mapping requires more storage space for different texture images and more computation for processing them than one texture mapping. For this reason, this project applies just one texture mapping.

When the red, green, blue, and alpha components are computed for a group, they are assigned to components of a texel as described above. As shown in Figure 5.4, if a primitive is clipped, colours (or texture coordinates) must be computed at the vertices introduced or modified by clipping. In this project, texturing is applied to patches of a surface along with Bézier-spline surface fitting. The texture mapping is done on each patch of the Bézier-spline surface. The Bézier-spline surface will be discussed later in Section 5.5.

5.4.2.11 Rasterisation

Rasterisation tackles the conversion of primitive coordinates to a 2D image that may be displayed on the hardware screen. Each point of the image represents a square (a pixel) located with window coordinates and has its values of colour and depth. A window on the screen is a grid with integer 2D coordinates while primitive coordinates may be not integer. Rasterising a primitive must determine which squares of the window grid are covered by the primitive and assign values of colour and depth of the primitive to these squares.

As mentioned in Section 5.1.2, in OpenGL, a grid square along with its parameters of colours, depth, and texture coordinates is called a fragment. In the I-GL, a fragment is positioned by its lower left corner, which is on integer grid coordinates. Since a fragment's centre is offset by half of side length of a square in the right and up directions from its lower left corner, the centre is on half-integer coordinates. In antialiasing and texturing, this subtle difference should be considered.

In the I-GL, grid squares may not be square. The non-square grid may make a rendered image of an object deformed, more flatted or slimmed than its actual situation because of the accumulation of differences of all the squares. It can also make the antialiasing and texturing even more difficult to be implemented. Thus, a square fragment is accepted widely. This project adopts the square fragment.

A point's size and line's width can also influence rasterisation. For large round points and wide line segments, antialiasing must be done by using pixel coverage values to prevent their images from looking artificial. Computing pixel coverage values is not effective for polygon antialiasing. In polygon antialiasing, multisampling is used.

Polygon rasterisation is more complicated than those of points and line segments. A polygon can be a triangle strip, triangle fan, or series of separate triangles. Firstly, for a polygon, it must be determined whether it is front facing or back facing. The front facing may be produced by rasterisation, but the back facing may not. Secondly, it is determined if a fragment centre is inside the polygon or not. The method used to test a fragment centre is called point sampling. Inside the polygon, the fragment may be produced by rasterisation. Outside the polygon, it is not yielded. On a polygon boundary edge, it should be processed carefully to balance two sides of the polygon from left to right in order to avoid enlarging the polygon image area by counting both sides and making the image artificial. A fragment on a common side of two polygons must be yielded by only one polygon rasterisation rather than by both of them.

To produce fragments of a triangle, defining barycentre coordinates for a triangle is recommended by the OpenGL ES because barycentre coordinates can specify any point inside or on the boundary of a triangle uniquely. Barycentre coordinates are defined by the centre of mass with a set of three numbers, *a*, *b*, and *c*, each ranged in *[0, 1.0]* and with *a*

$+ b + c = 1.0$. The rasterisation results are transferred to the next stage of the fragment pipeline, as shown in Figure 5.1 and 5.2. The information may be applied to the frame buffer update.

5.4.2.12 Depth Offset

As introduced in Section 5.1.2, each fragment (or pixel) on a display screen has a unique element in the depth buffer. In the context initialisation, all the values of elements in depth buffer are set to the depth value of the scene background. When the depth test is enabled, any element value in the depth buffer can be changed according to the depth offset that any fragment of a foreground object is away from the background. Since several objects may exist in the scene and be scattered near or far, the depth test must be done for each of the polygons that compose any existing object. Thus, the depth offset in each element of the depth buffer may be changed by any polygon that is at the most front of the scene. The more individual objects exist in the scene, the more time the depth test processing costs. For this reason, to simplify the depth test processing, the depth values for all fragments produced by a polygon rasterisation can be set to a single depth offset that is computed for the polygon. In addition, the depth test is done along with updating of colours in the frame buffer, which will be discussed in the next section.

5.4.2.13 Fragment Operations

After rasterisation, each fragment is sent to the next stage of the fragment pipeline, as shown in Figure 5.2. The fragment pipeline performs operations on individual fragments before they finally alter the frame buffer.

These operations include updating the frame buffer based on the comparison between incoming depth offsets and previously stored ones, masking some fragments based on the depth test and stencil test, blending of incoming fragment colours with previously stored ones according to the settings of lighting and texturing, and doing logical operations on some fragment colours according to logical operation setting. All these operations are hidden from the application development. What the applications must do is to specify the operation modes and enable them.

5.4.2.14 Frame Buffer Access and Hardware Controls

Although the operations of the OpenGL ES are done finally on a frame buffer, the frame buffer is not part of the OpenGL ES because of the device-dependence of the frame buffer in an implementation.

A frame buffer is composed of a two-dimensional array of pixels, but the width and height of the array rely on an implementation. For this reason, the width and height of a window and the colour format can be set via the I-GL auxiliary API. The conversion from the OpenGL ES selectable colour formats to the colour format accepted by the device

screen should be implemented by the I-GL. Thus, each pixel in the frame buffer can be simply treated as a set of OpenGL ES selectable number of bits by the graphics applications. The applications do not necessarily realise how different the specification for a physical frame buffer in one implementation context from that in another implementation can be. In this I-GL, the width and height are restricted to the resolutions of the LCD display available on the FPGA-based ES platform, which are 800 and 480 pixels, respectively.

There are several bitmaps that can help to manage and control the auxiliary data of a frame buffer in a fast-access way. One bit in a bitmap has a corresponding pixel in the frame buffer. Each bitmap is designed for a specified regular buffer, such as a front colour, back colour, depth, or stencil buffer. Whether or not the bitmap is designed relies on whether or not its regular buffer is available in an implementation of the OpenGL ES. For example, in an implementation with a single buffer, a bitmap for a back frame buffer is no use. With a bitmap, the update of a frame buffer can be more efficient. If a bit of the bitmap is dirty, it means the corresponding pixel has to be upgraded. If not, it can save once of access to the frame buffer.

Colour values may be read back from a portion of a frame buffer, copied directly from one portion of the frame buffer to another, or changed with logical operations and blended with another portion. These transfers can involve in decoding or encoding for processing.

Since there are not OpenGL ES commands to configure a window system and frame buffer, and initialise the context for the OpenGL ES, it must depend on an auxiliary interface to do these tasks for the OpenGL ES. The initialisation of an OpenGL ES context is issued when a window system allocates a window for a drawing task. A frame buffer configuration and linkage with a context are also done at the same time.

In addition, how to display the frame buffer contents on a device screen is not the task of the OpenGL ES, either. The OpenGL ES specification does not provide the technique for how to transform frame buffer values into streaming of video flow of a device screen with gamma correction. Although the EGL API defines a portable mechanism for creating OpenGL ES contexts and windows for rendering, there is still a gap that exists outside the OpenGL ES and in connection with the physical window system. This part heavily relies on the physical window systems that can be quite different on one hardware platform from another in the ES world.

Thus, this auxiliary interface is device-dependent and must be implemented in the I-GL. These tasks are crucial because all the effects of OpenGL ES commands on the frame buffer are ultimately passed to a window system. The window system bridges the communication between OpenGL ES commands and the hardware platform. The window system allocates and initialises the frame buffer resources firstly, finds which section of the

frame buffer the OpenGL ES may access at a given time, interprets for the OpenGL ES how the section is manipulated, and makes a final control of drawing a scene on a display screen, all of which an OpenGL ES application specifies.

5.5 Implementation of Mesa-OpenGL on FPGA-based ES

The OpenGL ES, however, does not include some functions that are used in this project. To include these functions, the core and some facilities of a Mesa OpenGL are modified and enhanced for this project. The revision of Mesa-OpenGL has been ported successfully to the FPGA-based ES platform for this project.

These facilities provide not only all the general functions including graphics primitive setting, attribute specifying, geometric transformations, illumination, setting up view and projection matrices, and clipping and projection transformations, but also some advanced computations such as describing complex objects with line and polygon approximations, displaying Bézier curves and surfaces, processing the surface-rendering operations, and other operations. As OpenGL is designed to be hardware-independent, the input and output routines are not included in these facilities. A pivotal interface connecting the OpenGL with the FPGA-based ES has been constructed as well, which make the images display on the LCD screen and user interaction for the surface editing work well.

The general functions have been introduced in the last section. The following is about the additive parts that are unique for this project and different from other OpenGL ES implementations.

5.5.1 Bézier Curves and Surfaces

In the general OpenGL ES specifications, some functions for more advanced evaluation, for example, functions that handle Bézier curves and surfaces are removed. For this project, however, the goal of surface editing requires the evaluation of Bézier curves and surfaces. In addition, functions of Bézier curves and surfaces are device-dependent. To see these issues clearly, let us first inspect the process of surface modelling.

The surface of a complicated 3D object can be divided into a grid of quadrilaterals in two directions, such as horizontal and vertical directions in a local coordinate system. Each quadrilateral can be divided further into two triangles. In reverse order, the surface can be formed with primitives that are triangles by a sequence of operations. Firstly, triangles are linked together to form triangle strips along one direction, for example horizontal direction. In the vertical direction, triangle strips can be combined to form the surface. Since each triangle has three vertices, vertices of all the triangles in each triangle strip can be arranged in a sequence of vertices.

To draw the surface, we must choose a starting point, for instance the left-bottom corner of the surface. The drawing process starts from this corner and moves in the right (horizontal) direction first. In a triangle strip, from left to right, each triangle is drawn one after another. In each triangle, the drawing is done by scanning each line section from left to right and line by line from bottom to top.

After one triangle strip is finished, the drawing process moves in the up (vertical) direction to the next row of the grid of quadrilaterals, that is, the next triangle strip. The triangle strips are drawn row by row from the bottom to the top. The surface drawing is done by traversing all the triangles in the grid. It can be seen that to draw the surface of a complicated 3D object is costly. This may be one of reasons why the OpenGL ES specifications do not include the Bézier curve and surface construction functions.

In this project, a Bézier surface is formed with a grid of quadrilaterals. The commands of *glMap2f*, *glMapGrid2f* and *glEvalMesh2* can be used to call the main functions for Bézier surface evaluation. The *glMap2f(GL_MAP2_VERTEX_3, uMin, uMax, uOffset, uPts, vMin, vMax, vOffset, vPts, ctrlPtsPointer)* is used along with *glEnable(GL_MAP2_VERTEX_3)*. These two commands make a 3D patch map into the *u* and *v* space of Bézier surface. The *u* values are limited in the range of *[uMin, uMax]* and spaced evenly as the *uPts* number of values. Similarly, the *v* values are limited in the range of *[vMin, vMax]* and spaced evenly as the *vPts* number of values. The 3D patch is described with an array of 3D coordinates of vertices whose location is pointed by *ctrlPtsPointer*. The number of elements in the array is the product of *uPts* and *vPts*. In the array, the offset between two elements in *u* direction is set to *uOffset* while the offset between two elements in *v* direction is *vOffset*. These are also shown in Figure 5.5.

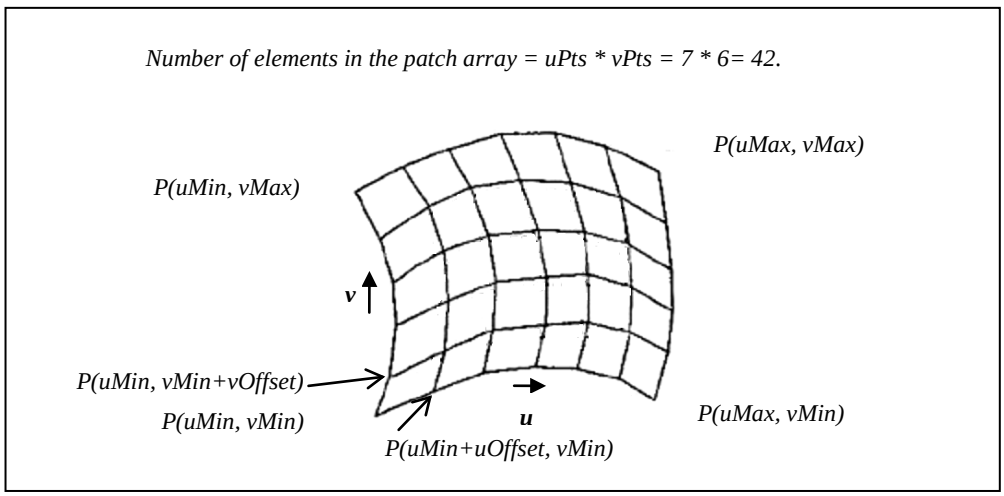


Figure 5.5 A 3D Patch Mapping into Bézier Surface Space

To map a texture into a patch of Bézier surface, similarly, the *glMap2f(GL_MAP2_TEXTURE_COORD_2, uMin, uMax, uOffset, uPts, vMin, vMax,*

vOffset, vPts, texPtsPointer) is used along with *glEnable(GL_MAP2_ TEXTURE_COORD_2)*. The *texPtsPointer* is the pointer that points to the texture coordinate array.

To generate evenly spaced parameter values, the *glMapGrid2f(unum, u1, u2, vnum, v1, v2)* and *glEvalMesh2(mode, un1, un2, vn1, vn2)* are used. Compared with *glMap2f*, they can split the surface into smaller triangle primitives and make the surface fitting more fine. In these two commands, the *unum* is the integer number of equal subdivisions over the range from *u1* to *u2*, the *un1* and *un2* are the integer parameter range corresponding to *u1* and *u2*, and the *vnum, v1, v2, vn1* and *vn2* have the similar meanings as their corresponding *u* parameters. *u* and *v* have the same meanings as the above.

In fact, the function of *glEvalMesh2* is not as simple as the above description. Each surface patch is split into a mesh of more fine sub-patches, and each sub-patch is split further into two triangles with a common edge that is a diagonal of the sub-patch, as shown in Figure 5.6. For each triangle whose bottom edge is not parallel to the *x* axis of the device screen, it is divided further into two sub-triangles, top triangle and bottom triangle, by a middle line that passes through the endpoint met by the top edge and bottom edge, as shown in Figure 5.7.

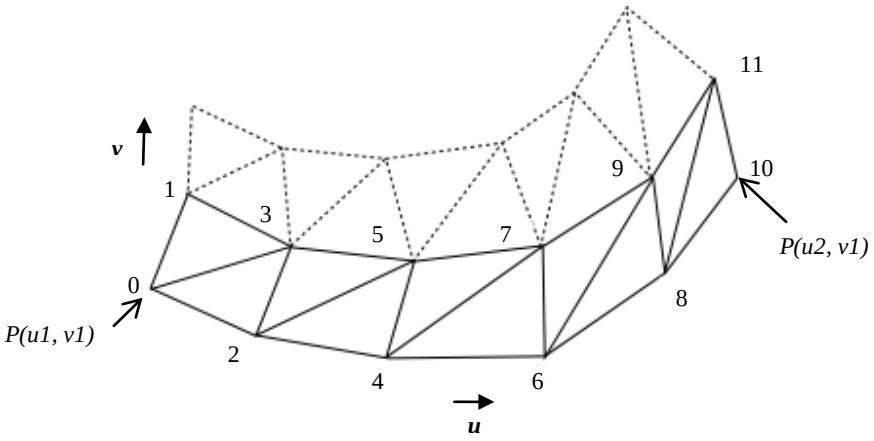


Figure 5.6 Process of Dividing a Surface Patch into Sub-patches along the *u* and *v* Directions with Functions of glMapGrid2f and glEvalMesh2. The Numbers are the Global Indices of new Vertices of Sub-patches.

There are two situations of scanning these split triangles because of the different orientations of triangles, as shown in Figure 5.7. In the left image, the starting points of scanning line by line are changed from one leg to another of the triangle when crossing the middle line. In the right image, the end points for scanning line by line are moved from one leg to another of the triangle when crossing the middle line. These two situations must be considered in programming for the Bézier surface evaluation.

Each sub-triangle is scanned line by line from left to right and from bottom to top.

Without removed by the depth test, stencil test and clipping, a scanned line is written into the frame buffer. Two triangles of each sub-patch are scanned in the order of first the bottom one and then the top one. All the sub-patches of each surface patch are scanned row by row, from left to right and from bottom to top. To facilitate the evaluation of each surface patch, new vertices of sub-patches are indexed globally for the surface patch, as shown in Figure 5.6. The function of glEvalMesh2 contains inner loops that draw all the triangles one by one in the order that they are arranged in triangle strips.

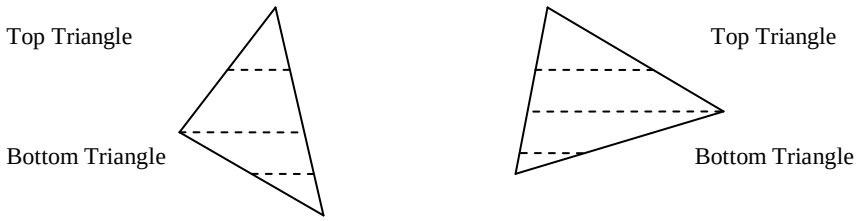


Figure 5.7 Process of Dividing a Triangle. A Split Triangle can be Orientated in Two Ways. The Left One is that the Left Endpoint of Middle Line is a Vertex of the Triangle. The Right One is that the Right Endpoint of Middle Line is a Vertex of the Triangle.

5.5.2 Window System Interface

To create a graphics display using the OpenGL, a display window must be set up first on the LCD video screen of the FPGA-base ES board. The display window is a rectangular area of the screen in which a 2D or 3D image will be displayed. The video screen is a typical output device. If the user interaction is needed by tasks, such as the surface editing, the input devices are required as well, such as a keyboard, mouse, and touch screen. These functions are hardware-dependent, and are not be implemented in the core facilities of the OpenGL.

For PC environments, such as Microsoft Windows, X Window System of UNIX, and the Apple Macintosh, there are additional available libraries that have been developed for OpenGL applications to communicate with the input and output systems. For example, the WGL is the extension for these tasks to the Microsoft Windows; the GLX is the extension to the X Window System; the AGL is the extension to the Apple Macintosh. The GLUT (OpenGL Utility Toolkit) library is an interface of the OpenGL to other device-specific window systems. With the GLUT, the programs of graphics applications can be hardware-independent.

For the FPGA embedded development environments, however, there is no available auxiliary library for these tasks. Therefore, an additional set of functions must be created to bridge between the FPGA-based ES platform and OpenGL. The output device is the

LCD display. Since there is neither a keyboard nor a mouse on the FPGA-based ES board, four simple buttons available on the board are used for user interactions of the exit control and surface editing in this project, as shown in Figure 3.8.

The followings are the extension facilities that are implemented for the connection of the Mesa-OpenGL to the FPGA-based ES platform.

- To allocate a window system to a drawing task. It carries out the allocation and initialisation of a LCD display device and double frame buffers. For user interactions of surfacing editing, the allocation and initialisation of buttons are done as well.
- To create and initialise a context. It carries out creating a context for the new drawing task and allocating the memory space to the context. It also does the general initialisation for building default information for the task.
- To make the context as the current state. It accomplishes binding the context to the LCD display device and double frame buffers.
- To build the drivers for the frame buffer reading and writing. It implements a group of functions for different styles of the frame buffer reading and writing, for example, reading or writing four-byte colour span, reading or writing three-byte colour span, reading or writing mono-colour span, reading or writing index-colour span, reading or writing individual colour pixels, reading or writing individual mono-colour pixels, and reading or writing individual index-colour pixels. It can do more styles of reading or writing if needed.
- To create a user interaction interface. It completes the registration for the interrupt signals of four buttons' being pressed, and the interrupt service routines for pressing each of four buttons.
- To swap between two frame buffers. It swaps the displayed frame buffer with the written frame buffer in order to smooth the video flow.
- To output the frame buffer. It makes the written frame buffer that has been updated to be displayed on the LCD screen as soon as possible.

The line segment scanning for drawing triangle strips in Bézier surfaces must make calls of the function of writing four-byte colour span because the frame data format is 32 bits, as shown in Section 4.7.6.2 and Figure 4.9.

5.5.3 Fixed Point Arithmetic

Another issue is that most of the evaluation of Mesa-OpenGL should be done on float point numbers, but the float point hardware unit is not available in the FPGA-based ES

platform. If using the software float point unit, the computation would increase a lot and the speed would be too slow for surface editing with user interactions. Therefore, the fixed point system is adopted for most of the graphics evaluation. For this reason, the work to modify Mesa-OpenGL from the float point system to the fixed point system must be done. To maintain the accuracy, some mathematical operations have to be created for the fixed point system, such as multiplication, division, square root, dot production, cross production, trigonometric functions, and linear interpolation.

With the hybrid way to construct ESs, a significant benefit is that we can have a flexible control on the data manipulation according to the arithmetic rules. This can support the fixed point system to have a satisfactory accuracy during the arithmetic operations. In this fixed point system, the fixed-point one is set to *0x00000800* and the fixed-point half is *0x00000400* in the hexadecimal representation. There are *11* binary bits for the fractional part. Its accuracy is $1/2048 \approx 0.000488$. Since we can use shift operations directly, we can keep the computation accuracy during computing and avoild the overflow by properly using the left shift or right shift. For example, if three numbers need to be multiplied, we can multiply two of them first, right shift their product by *11* binary bits, multiply the shifted product with the third number, right shift the second product by *11* binary bits, and then obtain the final product. In the similar way, when doing the division in sequence, we use the left shift. Therefore, no matter how many times of multiplication or division in sequence, the accuracy can be kept. This mechanism of keeping the accuracy can support the surface modelling and editing with the PAMA effectively.

Compared with the float point system, one of benefits of the fixed point system is to accelerate the graphics rendering because of its low cost in computation. For the small footprint of ESs, the computation should be simple, feasible and low-cost in memory space and time, rather than complicated and unfeasible. The fixed point can meet the needs.

Applications of this Mesa-OpenGL implementation will be presented in Chapter 8.

5.6 Chapter Summary

In this chapter, the standard of the OpenGL is introduced. A deduced OpenGL that is specified for ESs, the OpenGL ES, is discussed. Because of different views for the implementation of the OpenGL or OpenGL ES, three different roles in the OpenGL ES are discussed in order to obtain a deep insight of OpenGL ES implementations. As the specification of the OpenGL ES does not include the functions of supporting the 3D surface editing, Mesa-OpenGL, a special implementation of the OpenGL, is modified and enhanced for this project.

To implement the novel integration of Mesa-based OpenGL into the FPGA-based ES, more functions have been developed and implemented. Specifically, Bézier curve and surface construction functions are added to this implementation in order to support the PAMA. A dedicated input and output interface has been created for the communication between the OpenGL and the hardware platform. A fixed point system is created in the Mesa OpenGL ES implementation for this project. To meet the accuracy requirement of computation, this fixed point system is equipped with its own functions for multiplication, division, square root, dot production, cross production, trigonometric functions, and linear interpolation.

Chapter 6 Parallelism Implementation in FPGA-based ES

We have investigated the traditional computation parallelism in Section 2.4. Since this project presents a new hybrid way to speed up graphics applications, the traditional computation parallelism must be expanded to meet the new needs of FPGA-based ESs. Thus, in this research, parallelism is seen in a broad and new sense, over and above general parallel computation.

To make a theoretical connection to traditional parallelism, we should clarify the classification of traditional parallelism first. Then we can locate expanded parallelism in a proper position in terms of the traditional parallelism framework.

6.1 Classification of Traditional Parallelism

There are different types of parallelism, including bit-level, instruction-level, data, and task parallelisms. Classical parallelism is based on processor elements to execute in parallel. Strict speaking, data parallelism means splitting data into segments and assigning them to processor elements all of which execute the same instructions. The SIMD and MIMD discussed in Section 2.4.1 belong to data parallelism. Task parallelism means dividing a large algorithm into small pieces of tasks and distributing them to different processor elements to execute concurrently. The operator parallelism introduced in Section 2.4.2 belongs to task parallelism.

Granularity is an important feature and is used to make a comparison among different parallel systems. The granularity can be relative to two kinds of comparison: processing elements and processed data. The former is the comparison between the procedures of each processing element among different parallel systems; the latter is the comparison between the data sections that are processed by each of processing elements in different systems. Three granularity levels are usually defined: fine-grained, medium-grained and coarse-grained. In fine-grained systems, the cost of inter-processor communication is low, and is comparable to a basic arithmetic operation; conversely, the communication cost of coarse-grained systems is higher than that for a basic arithmetic operation. If it is appropriate for an application in a parallel system, one instruction can be the most

fine-grained in the comparison between procedures of processing elements; one bit can be the most fine-grained in the comparison between processed data. Thus, instruction-level parallelism can belong to task parallelism. In this case, the tasks distributed among processing elements are a small set of instructions. Bit-level parallelism can be a member of the data parallelism family, which has a number of datasets processed by each processing element.

Pipelines or cascaded structures can be a variant of task parallelism, and they make a chain of different hardware modules with certain flexibility to attain a synergy as a whole.

From a system perspective, when applications are placed on graphics processing units (GPUs) or ESs with FPGAs, co-processors plays a key role (Cevik 2004, Cheng and Goshtasby 1989, and Sridharan and Priya 2004). Co-processors can be seen as task parallelism in a broad sense.

From a graphics algorithm and implementation's perspective, because of the graphics pipeline goal, the parallel attributes can be naturally deconstructed into the pipeline accomplishment of GPUs or FPGAs. It works similarly to specified graphics pipeline chips in the commercial ESs, whose applications can be in mobile phones, video game players, GPU-based DSP, video distribution, video windowing, and graphics. It can be an extension or refinement of pipelining.

This project maximises task parallelism where possible and appropriate in more operations of elementary arithmetic and logic operators (units) than in processor elements. Since the hardware is involved directly in this research, fine-grained parallelism in hardware system can be carried out in the FPGA-based ES. The co-processors play an important role for such parallelism.

6.2 Analysis on Processing Features in Parallelism

An in-depth analysis of parallelism in a system must be done before exploring how to parallelise the computation and processing in the system. This analysis sees a broader picture of the parallelism than traditional parallel computation. Thus, two perspectives are discussed to view parallelism, the application programmer's view and the hardware builder's view, and two styles of parallelism, spatial and temporal.

6.2.1 Two Perspectives: Application's View and Hardware's View

The study of Grama et al (Grama et al 2003) proposes to view the parallelism from the two different perspectives of logical and physical organisation of parallel platforms. Logical organisation is seen at a high level from an application programmer's view of the platform whereas physical organisation is taken at a low level from an actual hardware builder's

view of the platform.

From the application programmer's view, the goal of parallel computing platforms is how to program in order to gain high performance and portable parallelism. To reach the goal, two factors of parallel computing are critical: one is the parallel control structure that is the mechanism of expressing parallel tasks; the other is the communication model that is the ways for indicating communication between parallel tasks. Instructions, rather than operations, are concerned more in programming.

From the hardware builder's perspective, the goal of parallelism is how to realise parallelism with hardware units at low cost to attain high performance. The operations and mechanisms needed to make these operations continuous and orderly are the focus of hardware building. These result in the technologies of streaming, pipelining, and concurrent processing. In a pipeline or stream, a sequence of operations that can be implemented by different hardware units performs a series of actions on multiple data items. A programmer may be able to execute them all with just one instruction. This parallelism may not be visual for programmers. It does, however, conform to the nature and rules of physical devices and parallelism. In this project, the video pipeline discussed in Section 4.7.6.2 functions in this way.

On the other hand, in a given system architecture, imagine that any processing level could be implemented on hardware specifically designed for it so it can perform its tasks efficiently. It might lead directly to the design of heterogeneous pipelined processing structures for complex applications. It should be noted that information density and operation complexity and flexibility increase from the low level to the high level. But the amount of data to be computed and computing power decrease from the low level to the high level. Up to the high level, parallelism technologies become more abstract and flexible whereas down to the low level, they become more specific and simpler. The operations on data at the high level are changeable whereas operations on multiply data sequences at the low level are fixed and repeatable.

In this research, on the top level, surface modelling and editing with PAMA are applications. The polygons that form the surface can be computed in parallel without limitation of the surface shape. At the low level, the 32-bit data of each pixel can be streamed in the video pipeline and finally displayed on the LCD screen that accepts the stream of 8-bit data for red, green and blue, individually.

Therefore, the traditional computation parallelisms are more likely to be viewed from the programmer's perspective even though they have to be supported by designated hardware platforms. This project mainly adopts the hardware builder's view.

6.2.2 Two Styles: Pipelined Parallelism and Partitioned Parallelism

Dewitt and Gray (Dewitt and Gray 1992) state that the dataflow approach to relational operators in relational database systems offers two styles of parallelism, pipelined and partitioned parallelism. Pipelined parallelism is the technology of streaming the output of one operator into the input of another operator when two operators work in series. Partitioned parallelism is a mechanism for partitioning the input data among multiple operators, each of which has its own processor and memory and works independently on part of the data simultaneously along with other operators. Figure 6.1 shows these two styles. In fact, pipelined parallelism provides a method for the timely processing a stream of a fixed amount of data in series whereas partitioned parallelism offers a way to simultaneously process a large amount of data by partitioning them into small portions and distributing them to multiple operators.

The former provides temporal parallelism. It makes a low-capability processer do a high-strength task by partitioning the task into small portions and transmitting the portions in series at high speed. The latter gives spatial parallelism. It uses several low-capability processers together to complete a high-strength task simultaneously.

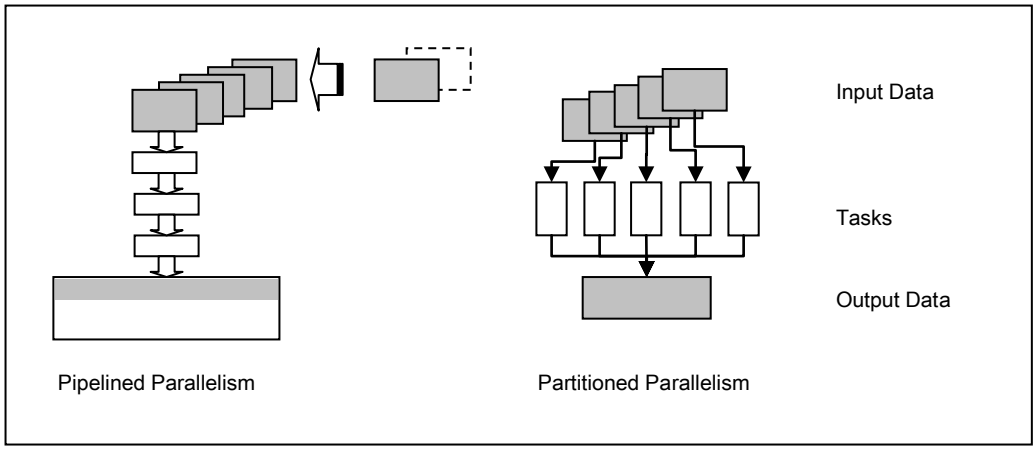


Figure 6.1 Pipelined Parallelism and Partitioned Parallelism

This suggests that sometimes a large amount of data in the spatial space, like the data of a frame buffer of hardware display, can be transformed into a stream of a fixed amount of data to process in series timely. The latter is the division of the former by time.

In this research, the video pipeline is adopted for this reason. A frame buffer composed of 800 X 480 pixels is the high-strength task. The video pipeline plays the role of low-capability processer, which handles timely the data pixel by pixel in order to make the LCD screen scan line by line in real time. When the surface is edited with the PAMA by user interactions on the top application, it is necessary to update the relative pixels on the LCD screen with the data of the changed shape of surface. The temporal parallelism of

video pipeline supports the shape editing.

6.3 Methodologies of Processing in Parallel

To implement parallelism for an algorithm in a system, the computation decomposition of the algorithm at the high level has to be done with the decomposition techniques first. After decomposition, the tasks parsed out by using the techniques have to be mapped onto the processes that can be executed by the operators available in the system.

The computation decomposition techniques cannot settle all the issues in parallelism. For example, in the distribution computation or processing, which is widely used in database systems, the client computers do their tasks, such as transactions, independently, and the server end batches up the query tasks from each client into the database server and processes them in sequence. In this architecture, the tasks of clients are independent of servers. They cannot be treated simply as the decomposition from an algorithm.

On the other hand, at the low level, only the computation decompositions are not enough for parallelism in hardware building. For the hardware building, the operation-dependent relationships between hardware sub-units can have more influence on the parallelism in hardware system. Parallelism decomposition thus has to be done physically and accurately on every bit and every signal at every system clock, rather than logically and roughly on a group of data in an accepted period of time, as is done at the high level.

From the system perspective, co-processors are an alternative option to parallelism. They can do their tasks independently in parallel while the main processor handles the main applications. Their tasks are usually a set of physical behaviour or controls rather than the computation of an algorithm. Therefore, co-processors should be considered in the parallelism of a hardware system.

6.3.1 Computation Decomposition at High Level

Grama et al (Grama et al 2003) also address a method for parallelism analysis of computation at the high level. It is called decomposition techniques. It can provide a good starting point for parallelism analysis in many real computation problems at the high level. It can be used to analyse and decompose the operations in complex problems, which can be executed in parallel, by combining one or more decomposition techniques. Though decomposition techniques may not always bring about the best parallel solution to a problem, they offer a feasible method for turning a common problem into a parallelism problem.

The basic idea of this method is that a given problem has to be split into computation sub-steps that can be executed concurrently and independently, and a task-dependency graph can be defined. This graph can tell which sub-steps in the problem can be executed concurrently.

6.3.1.1 Four Decomposition Techniques

The decomposition technique has four types: data decomposition, recursive decomposition, exploratory decomposition, and speculative decomposition.

The data and recursive decomposition techniques can be applied to general problems whereas the speculative and exploratory decomposition techniques may be used in specific problems.

6.3.1.2 Data Decomposition

Data decomposition can be used in parallelism decomposition on algorithms that operate on a large amount of data. It takes place in two steps: data partition and computation task division. The former is done on the data with their independency; the latter is carried out according to the results of the former to determine which operations should be done on each data portion. The operations on one data portion are relatively fixed and similar. Data decomposition can produce several solutions to a given problem. It has to evaluate among them on their performance and efficiency before choosing one.

Data decomposition can be done on input, output, and intermediate data of a given problem. The data decomposition can also conform to the owner-computes rule. It means that each owner has its own task and data, and all the computation operations of its task are involved in just its own data. For the input or output data decomposition, the relationship between owners and tasks in the owner-computes rule can have different meanings. In input data decomposition, the owner of a portion of input data should be the owner of the task that performs all the operations on this portion of data and produce results. In output data decomposition, the owner of a portion of output data should be the owner of the task that performs all the operations that can produce this portion of output data.

With intuition, it is most natural to start with output data decomposition because each individual part of the output of the problem can be processed independently from other parts of the output. If possible, the operations that are used to yield that part of output should be independent of those used in processing other parts of the output. Thus, the computation task division occurs naturally.

For example, it is well-known that matrix operations on large matrixes can be replaced by the formulation of matrix operations on small block matrixes. The latter can significantly decrease the cost of computing and the requirement for storage. Another benefit of the

latter is that the matrix operations on block matrixes can be applied in parallelism decomposition. This makes matrix operations on large matrixes transform into matrix operations on several small block matrixes, which can be processed in parallel.

A restriction of output data decomposition is that it can work well only if each output of the problem can be computed as a function of the input. Sometimes, it is not the case. It is natural to turn to the input data decomposition to find a solution. If it is found that the input data of a problem can be divided into independent groups and it induces operations on each group that can be performed in parallel, this problem is one that can be addressed in the input data decomposition. The obvious feature of this type of problems is that the operations of the task performing on each independent group of input data are isolated from those on other groups of input data. That is, one task is input data independent of other tasks. For instance, number or alphabet sorting is such a case. Figure 6.2 shows output and input data decompositions.

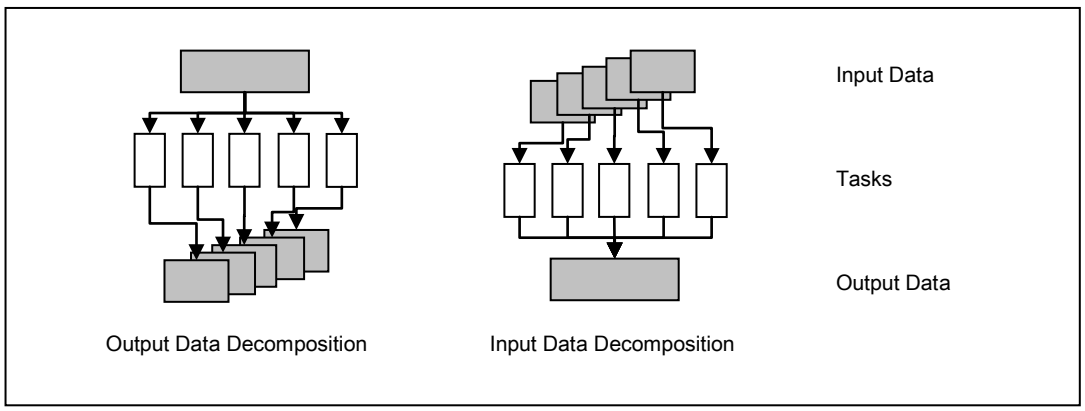


Figure 6.2 Output and Input Data Decompositions

Usually, the problems to solve are not ideally suited to output or input data decomposition. For these problems, the intermediate results may be used for parallelism decomposition. Intermediate data decomposition suits to problems that need to be processed in multiple stages and intermediate data to produce the final output. The intermediate data can be treated as the output data of the operation of an intermediate stage or the input data of the operation of its subsequent stage. Thus, it can be seen as the output or input data decomposition of a sub-problem of the original one. It can focus on looking for the data independence in this sub-problem and carrying out data decomposition.

In reality, problems do not obviously belong to any single type of output, input or intermediate data decomposition. We need to look for a solution by combining output, input and intermediate data decompositions. In some cases, it is possible to gain more concurrency by using input data decomposition after the output data decomposition. In other cases, a serial algorithm may not explicitly fit into any type of data decomposition.

But when the solution structure of the serial algorithm is reorganised, it may yield the chance for the intermediate data decomposition. Intermediate data decomposition requires more exploration and may produce higher parallelism than the output or input data decomposition.

6.3.1.3 Recursive Decomposition

Recursive decomposition can be expressed as a problem that is first divided into several independent sub-problems, meaning that there is no data-dependent relationship between the sub-problems. Each sub-problem is further divided into smaller sub-problems in the same way as the upper division level. At any division level, the results of all the sub-problems are combined for the upper level. In the end, all the results at different division levels are combined at their division levels and finally form the final result at the top level recursively. Thus, the problem can be treated as a divide-and-conquer problem. Along with problem division, the solution algorithm to this problem can be split into sub-sections at different levels as well. At any division level, all the sub-sections can be executed concurrently. Figure 6.3 shows a task-dependency graph for three division levels.

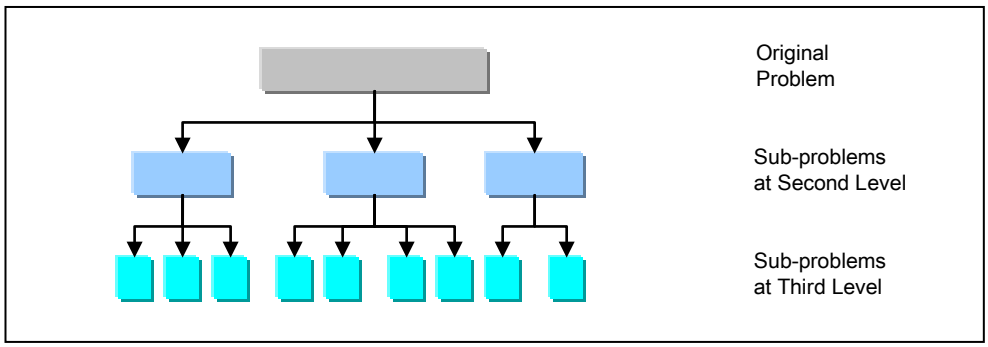


Figure 6.3 Task-Dependency Graph for Three Division Levels

6.3.1.4 Exploratory Decomposition

Exploratory decomposition can be used in problems that carry out a search in a space for solutions. The search process shrinks the space in which the solutions are included. Therefore, the original search space can be divided into small regions, in each of which the search process can be done concurrently until the solutions are found.

In some way, exploratory decomposition is similar to data decomposition of number or alphabet sorting problems. Both of them search in partitioned sets. There are some differences between them, however. Data decomposition has to be done in the entire set of acceptable values and the final solution of the problem relies on the results of all the tasks. Exploratory decomposition can reach the end without waiting for all the tasks to be finished if all the solutions of the problem are found. Consequently, the way in which the original search space is partitioned can seriously affect the parallelism performance of

exploratory decomposition. Poor partitioning means that the parallelism computation may not be better at speed than its corresponding serial algorithm.

6.3.1.5 Speculative Decomposition

Speculative decomposition can be applied to solving problems that one task has different branches of operations because its input has different patterns. The input may be the computation result that is produced by the task just before this one. This dependence relation means that this task cannot start until the result of the previous task is provided. Its behaviour is just like the conditional branch statement in high-level languages.

One way to solve this kind of problem is simply to compute all the possible branches of this task and gain all the possible results without waiting for the last task to yield the 'condition' for this task. Therefore, this task can be performed concurrently with other tasks without waiting. It can speed up the whole processing of the problem. Once the 'condition' is produced, the corresponding result of the correct branch of this task can be output to the next task and the results of other branches become useless and are ignored. It is obvious that the processing of the incorrect branches is pointless. Thus, speculative decomposition issues a compromise solution by simply doing the computation of the most likely conditional branches. It can meet most situations. If one of the non-computing branches is correct, its relative computation must be made up.

One the basis of the statistics, the overall performance of the speculative decomposition may not be lower than its counterpart, a serial algorithm. If there are several stages of speculative decomposition in one problem, their speed-up effect can be enhanced much more.

If we compare speculative decomposition with exploratory decomposition in detail, some differences can be discovered.

In speculative decomposition, there is always more work to be done than its corresponding serial algorithm in order to enlarge the parallelism by pre-performing some tasks that will not be used in the processing. Speculative decomposition is faster than or equal to its corresponding serial algorithm. In exploratory decomposition, since the division of search regions is not unique, it cannot be determined in advance how much the parallelism can speed up and how much more or less work the parallelism can bring about than its corresponding serial algorithm.

In speculative decomposition, the possible set of output is known. It is the results for all the possible conditions. But the input for the correct condition is not known in advance. In exploratory decomposition, the possible set of input is known, and is the search space. But the output is unknown in advance, i.e. the portion which will result in the correct output solution to the search problem.

6.3.1.6 Mixing Decomposition

The above decomposition techniques can be combined during the application. Often, the computation for a problem has a structure with multiple stages and these different stages may have various features that match different decomposition techniques. Thus, it is necessary to apply different types of decomposition in different stages. Hybrid decompositions are acceptable.

This project is a system-level research that includes a hierarchical architecture composed of a wide variety of problems. Mixed decomposition is adopted.

Since there are many matrix operations in the Mesa-OpenGL for FPGA-based ESs, input data decomposition is used in dividing a large matrix into small block matrixes. When an object surface is complicated and has to be represented with a big mesh or grid, its vertex matrix can be large. Data decomposition in large matrixes can decrease the cost of computing (especially multiplication) and storage.

As discussed in Section 5.5.1, for surface editing with the PAMA, the edited surface is a problem at the top level, which can be solved with recursive decomposition. This problem is first divided into small patches, as shown in Figure 5.5. Each patch is then divided into two triangles, as shown in Figure 5.6. Each triangle can be processed independently because its data are independent from those of the others. Thus, the triangles can be processed in parallel.

6.3.2 Parallelism Mapping at High Level

Decomposition techniques are used to identify the concurrency in a problem and decompose it into tasks that can be executed in parallel. After decomposition, the tasks that are parsed out can be mapped onto the processes that can be executed by the operators available in a specific parallelism system. The mapping includes programming the tasks of a parallel algorithm or re-programming a serial algorithm with the parallelism style designated for the specific parallelism system. Except for conforming to the designated parallelism programming style, the above decomposition can provide a lot of fundamental parallelism information for the mapping. The behaviour of the tasks and the interactions between them offers guiding for mapping. The task behaviour can explain how an operator can process the data portion whereas the interaction behaviour can indicate how the operator communicates with others.

A parallelism mapping plot is expected to execute faster than its serial counterpart. Four features of the tasks significantly affect the parallelism mapping plots.

6.3.2.1 Task Running Time

The amount of time that a task requires to complete is the task running time. As is widely

known, the inherent serial section that cannot be replaced by parallel tasks places the most constraint on a parallelism mapping plot. It is ideal for parallelism mapping where all the tasks in parallel take the same amount of time to complete. That is, all the tasks are uniform. The parallelism algorithm will be faster than its serial counterpart.

For example, there are several matrix left multiplication in the Mesa-OpenGL for FPGA-base ES, as shown in Section 5.4.2.7 and Figure 5.4. In these matrix multiplication problems, the tasks parsed out by the parallelism decomposition can be uniform because we can determine the sizes of these matrixes in advance.

Since the complexity of parallelism problem can vary, parallelism decomposition cannot divide every problem into uniform tasks that can be completed in the same amount of time. For instance, the search problem is a non-uniform one. Some task, like the inherent serial section, may take more time than other tasks. This kind of task will particularly influence the effect of the parallelism mapping plot.

For instance, the recursive decomposition of the edited surface, as given in Section 6.3.1.6, cannot be divided into uniform tasks since patches with different shapes are divided into pairs of different triangles. One triangle can have a different shape and area from those of others, which can result in different line segments scanned line by line for each triangle, as shown in Figure 5.7.

6.3.2.2 Knowledge of Task Running Time

If the task running time of a parallelism mapping plot can be estimated, the task running time is known before the task is executed. This information is useful because it can enable assessment of how much faster a parallel algorithm can be than a serial algorithm. For example, in a matrix multiplication problem, the amount of time can be known, derived by the execution of a small block sub-matrix multiplication and scalar algebraic operations. This evaluation can help to assess whether or not a parallelism mapping plot is good enough and whether or not there is room for optimisation.

In some cases, it cannot be known in advance how long a task takes to execute. For example, exploratory decomposition for a search problem may mean that we cannot be sure how many steps to be taken to find the final solution. It depends on every choice made at each stage. Thus, uncontrolled factors make the task running time unobtainable. A good judgement cannot be obtained for this problem .

In this research, it can be known in advance that the running time of the left multiplication of a 4X1 matrix by a 4X4 matrix can be evaluated. The 4X1 matrix represents the coordinates of a vertex. The 4X4 matrix can be one of the transformation matrixes, as shown in Figure 5.4. On the other hand, it cannot be known in advance how long time it will take to draw a surface. The triangles into which the surface is divided can

have various shapes and areas.

6.3.2.3 Known Task and Generated Task

There are two types of task. One is the task known before the algorithm starts execution, called a known task; the other is the task generated during the execution of the algorithm, referred as a generated task.

Known tasks usually can be parsed out by data decomposition. Even though recursive decomposition can generate many sub-tasks by using a task-dependency graph, these sub-tasks are known before the parallelism algorithm executes. They belong to known tasks as well.

Generated tasks may be yielded by decompositions when the parallelism algorithm is executing. At the high level, decomposition techniques may not lead to an explicit detailed task-dependency graph that includes the actual tasks. Recursive decomposition may yield an array of a designated size, but this array can include different number of operators with different capabilities for different parallelism hardware platforms because a task-dependency graph may not indicate detailed information about the hardware platform.

In any case, a generated task should be one that takes a state as its input or its generation condition. Then the task starts to extend by itself with a predefined number of stages, such as the scenario in a search problem, and dynamically generates more tasks to perform the same computation on each of the resulting states until the solution is found and the algorithm terminates. Whether or not the generated tasks are yielded depends on the input of an algorithm when being executed.

In this research, the surface drawing is a problem of recursive decomposition. It was seen in Section 6.3.1.6 that it includes three level tasks, these being the surface on the top, the patches at the middle level, and the triangles at the lowest level. These tasks are known tasks. But underling this recursive decomposition are unknown sub-tasks. We cannot know how many lines should be scanned in each triangle before the surface is evaluated. The number may change dynamically when the surface is edited with user interactions.

6.3.2.4 Related Data

In parallelism problems, a large amount of data usually needs to be processed. The access and movement of a large amount of data can be costly in terms of CPU and memory space, especially when communications between operators or I/O operations are required and the overheads become even worse. Therefore, data related to a task and the size of the data can also affect a parallelism mapping plot.

Sometimes, tasks have an orderly relationship because of data-dependence. The input

of the next task is the output of the last one. Whether or not the related data of a task are available can determine if the task can be performed in parallel with other tasks.

Related data sizes vary. The size of the input data can be different from that of output as well. For example, the size of data in a frame buffer is determined by the number of resolutions and the type of colour pattern. When it is processed with the streaming, the input of a task for a frame buffer is just one or two pixels. But the output can be a whole frame buffer. When it is processed by copying a whole image, the input of a task can be the whole image and its output can be the pixels in a square area of the frame buffer. For a search problem, the input of a task can be the set of search scopes, and the output of a task may be just one number in the set.

In this research, there are various data sizes in the graphics pipeline of OpenGL, as shown in Figure 5.1. In terms of its geometric pipeline, the vertices of a surface are basic elements that are processed and are 4X1 matrixes with four coordinates. The coordinates are fixed-point numbers with 11 bits for the fractional part. Via its fragment pipeline, a frame buffer with 800 X 480 pixels is stored. Each pixel in the frame buffer is an integral of 32 bits for the RGBA format with four components of red, green, blue and alpha and eight bits for each component. When the surface is finally displayed on the device screen, the LCD accepts only the stream of eight bits for each colour element including red, green and blue. These various data sizes restrict the system parallelism, which results in an orderly relationship between the geometric and fragment pipelines, and between steps in each pipeline. Thus, in this project, parallelism decomposition cannot map into the SIMD or MIMD architecture, but it can map onto the pipelined parallelism discussed in Section 6.2.2 effectively and efficiently. That is why the pipelined parallelism is adopted in the system of this project.

6.3.3 Expansion on Parallelism Decomposition

In this section, two ways of expanding parallelism decomposition will be discussed. One is broad expansion, to see a parallel frame in distributed systems with the operators that are independent computers connected with the particular networking infrastructure, no matter whether it is wireless or wired, local or internet. The other is extending down to the low level of the system, the hardware platform, to see how to handle the parallelism among the hardware units.

The co-processors usually applied in ESs and FPGA-based systems are also discussed.

6.3.3.1 Expansion of Parallelism Decomposition to Distributed Systems

Parallelism processing can exist in distributed systems. The distribution computation or processing is widely used in database systems (Jiang et al 2006, Kallman et al 2008,

Mohan et al 1986, and Thomson et al 2012). In these systems, each computer does the same tasks as others but may be located in a remote area. For example, in transaction processing systems, many clients submit requests for the database service. All the client computers do transactions in parallel. Each of the client computers executes its independent transaction separately. Each client machine can be a simple but integral computer system with its CPU, memory, and operating system.

On the other hand, at the database server end, each query from the clients can be translated and batched up into finer jobs to be transmitted to the database server to process. This is just like streaming the tasks from each client into the database server, which also adopts parallelism technologies.

Heterogeneous distributed systems can provide even more chance of parallelism at the system level. They can induce not only co-processor architecture but also networking of various types of computers.

6.3.3.2 Expansion of Parallelism Decomposition to Hardware Building

As mentioned above, at the high level, computation decomposition techniques are based on the data-dependent relationships between sub-problems. For parallelism problems in the hardware building, the operation-dependent relationships between sub-units are another important factor that can influence decomposition. This means that the result of the previous sub-unit can determine the start of the next sub-unit. For digital circuits, the result may be a signal transition from high voltage level to low voltage level, or versa, or a predefined number of system clocks, rather than a meaningful numeral value.

In general, a parallelism hardware platform can be used in one type of problem perfectly but not in others. Sometimes, the effect of its speedup may not be obvious because the problems that are processed on the platform may not fit into the operator configuration of system. Therefore, for a given parallelism hardware platform, applications at the high level have to think about how to make the best use of the parallelism resources in system.

On the other hand, hardware builders usually face the problems on how to make the existent hardware units work well together with the parallelism technologies in a flexible way, rather than in a particular fixed model. The problems to be solved are more detailed, physical and practical.

In the low level of the system, parallelism decomposition needs to be done physically and exactly for each bit, each signal, and each system clock, rather than logically and roughly on a group of data in an accepted period of time. It is a detailed, fine, and accurate piece of work.

6.3.3.3 Co-processor's Role in Parallelism Hardware Building

In the typical co-processor architecture, the main applications execute on the main

processors while the co-processors handle tasks that require a long execution time (El-Ghazawi et al 2008). Co-processors have designated hardware implementations, which can be fine-grained architectures: for instance, SIMD, engines, pipelines, or others. The system can invoke the co-processors to execute the specific tasks.

With ESs and FPGA technologies, it is to be expected that multiply processors exist in a system, and each of them has a designated assignment in the system in terms of assisting a host processor to fulfil a number of functions. Some processors may act as apparent or unobvious co-processors and do their specific tasks automatically without much intervention of the host processor. When these processors undertake their tasks, the host processor does not necessarily stop its own task to control and communicate with them.

The host processor may transmit a small set of instructions to a co-processor to launch it and then do its own main tasks. The co-processor starts doing its pre-designed assignment automatically. For example, there are co-processors that are designed for Fast Fourier Transforms (FFTs), two dimensional Discrete Cosine Transforms (2D DCTs), convolution filters, MPEG-4 main profile visual compositing, image processing, or image registration (Berekovic et al 2000, Dubois and Mattavelli 2003, Huang et al 2009, Kalomiros and Lygouras 2007, and MacLean 2005). These co-processors can share the pre-designated memory space on SRAM or DDR SDRAM banks with the host processor. The co-processors may send their processing results to the memory, and the host processor can access them if necessary. Alternatively, the host processor can put the data in the memory, and a co-processor can use them as input data.

Since co-processors do their tasks without interrupting the host processor's task, the co-processors do their tasks in parallel along with the host processor. Even though they do not share the same large task with the host processor, they have procedures totally different from those of the host processor, and their structures of hardware units are different from those of the host processor as well. Co-processors belong to one of the heterogeneous parallelism architectures.

In this research, the graphics hardware sub-system works independently of the Nios II, the core processor of the FPGA-based ES. It acts as a co-processor in the ES.

6.4 Parallelism in the Graphics Processing in this Research

For this research, in graphics processing, many operations are applied to individual objects rather than a 2D range or entire frame buffer. These operations are different and changeable depending on the objects processed by them. In other words, the proportion of the data processed by the same group of tasks is small in respect of all the vertices of the objects in the scene. These operations cannot be done in the same way as the

processing elements in the SIMD or MIMD architecture in applications of image processing. Pipelining is a good way to apply to graphics processing because one portion of vertices can be processed independently from other portions before being written in the frame buffer. Therefore, much effort is put in pipelining graphics processing in this project.

6.4.1 Pipeline Effect

Before discussing the pipeline effect in graphics processing, let us take a television assembly line as an example. If the assembly of a television set includes 50 component units and each of them takes one minute, each television set assembly takes 50 minutes with a serial assembly line. But if the assembly line is pipelined with ten stages of five units each, the assembly line can produce a television set every five minutes by overlapping these ten stages in production. The pipelined assembly line is ten times faster than the serial one. The pipelining can accelerate the production process.

In a computer system, pipelines are used to improve the instruction execution. The various function stages, including fetch, schedule, decode, operand fetch, execute, store, and others, are pipelined and interwoven to be processed at the same time. The reason behind this is that all the sub-tasks are done in different hardware units of the computer system, just like different work stations in a factory pipeline. When the pipeline runs for a while, all the hardware units are filled with their own sub-tasks. The parallel production style is set up.

After the instructions filled in all stages, different instructions are piped into different function units and executed each clock cycle in parallel. This can increase the execution speed and improve the performance of the whole computer system.

Theoretically, the smaller sub-task units are broken down, the bigger the overlaps between different task units processed at the same time, and the faster the computer system executes. The largest physical atomic sub-task of instructions is ultimately the smallest unit that can be divided in the pipeline.

In the LCD controller subsystem of Altera FPGA-based ESs of Cyclone III Version (Altera 2008b), the display process adopts the pipeline mechanism to match the data format and timing of frame buffer in the DDR SDRAM memory with the video flow of the LCD device, as shown in Figure 4.9. The pipelining process makes the video processing automatic and fast, and achieves high performance. The data format in the frame buffer is 32 bits for each pixel, three bytes for red, green, and blue, respectively, and one byte is not used. The video flow of the LCD device accepts bytes of red, green and blue, byte by byte, in sequence.

6.4.2 Timing and Data Format Matching in Pipelining

Let us give another example. When a tap is turned on to let water run into a container, it takes some time before the water begins to come out. How much water flows out and how quickly it does so is dependent on the size of the tap.

In a computer system, the performance of a program running on a computer relies not only on the speed of the processor but also on the ability of the memory system to transmit data to the processor. There are two factors that can influence the throughput ability of the memory system. First, when the processor transfers data from or to the memory, the memory takes some time before it is ready for reference, that is, the latency of the memory. Second, the rate at which data can be transferred from or to the memory determines how fast the data move between the processor and memory. The latter is the bandwidth of the memory.

The mismatch between the processor and SRAM (or SDRAM) speeds has motivated a number of architectural innovations in memory system design. As mentioned in Section 2.4.4, one such innovation addresses speed mismatch by placing a smaller and faster memory between the processor and memory, which is cache.

The differences between the above two examples are that the size of the tap does not necessarily exactly match that of the container's mouth, but the size of data bus of the memory has to be the same as that of the processor. A slow rate of the water flow does not matter to the container, but a slow transaction speed of the memory system can lower system performance even though the processor has a high speed.

For hardware devices that do not match, it is obviously unacceptable to control the devices simply by connecting them together. It is necessary to know how they work first, then to connect and harness them in the appropriate way.

In the graphics sub-system of Altera FPGA, the data transactions are done between the frame buffer in the DDR SDRAM memory and the LCD display. At one end of the video flow is the frame buffer with a 64-bit data bus. At the other end is the data interface of the LCD display that accepts a flow of three sequential 8-bits. Since it is expected to keep the video data flow under control, it is necessary to build a precise pipeline that has different interface sizes at the two ends between the LCD display and the frame buffer, which is video pipeline. At one end of the pipeline, the size of the bandwidth has to match exactly the bus of DDR SDRAM memory. At the other end of the pipeline, the size of the bandwidth fits into the flow for LCD display.

Compared to the Nios II processor, the LCD display is an output device, which is slower and must be controlled with its abstract registers at its rate and order. The abstract registers are not common device registers, but function similarly. The flow of data is

expected to be put into their holders at the appropriate rate.

Figure 6.4 shows the video pipeline applied in Altera LCD controller (Altera 2008b). The tasks of the video pipeline are to read data from the frame buffer in the DDR SDRAM memory, transform data format, adjust rhythm, and drive video data signals on the LCD data bus. For this project, the frame buffer data are generated by the Nios II processor with the Mesa-OpenGL implementation for the FPGA-based ES platform, discussed in Chapter 5.

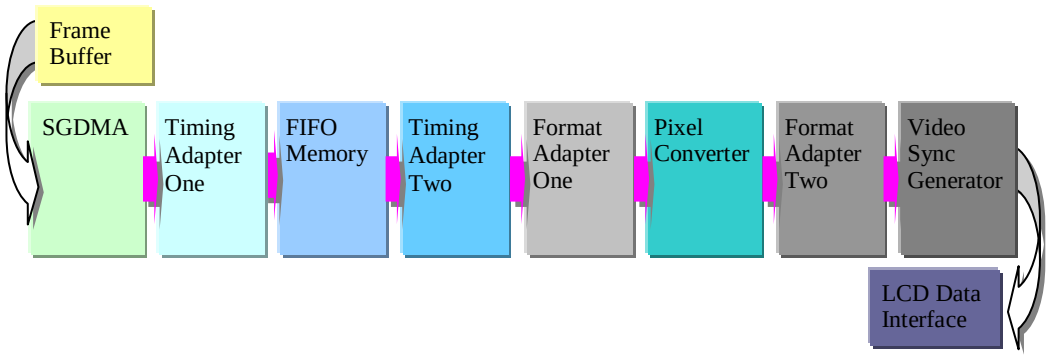


Figure 6.4 Video Pipeline in Altera LCD Controller

The video pipeline consists of eight parts.

- Having been initialised, the SGDMA controller reads pixel colour data from the frame buffer in the DDR SDRAM memory and passes the data to the remainder of pipeline without intervention by the Nios II processor. The SGDMA autonomous operations are controlled by a chain of descriptors. The descriptors do the data transfer at a speed of 64000 bytes per transfer. In the initialisation, the descriptors for a whole frame buffer display are prepared. Thus, the SGDMA controller can process the entire frame buffer and drive the rest of video pipeline continuously. Since the interface width of the DDR SDRAM memory is 64 bits, it is more effective for the SGDMA to read 64 bits at a time, which are units for two pixels.
- The Timing Adaptor One is used to make up the difference between the time length of data latency of the downstream FIFO memory and that of the upstream SGDMA. The former is one unit of latency; the latter is zero units of latency. They rely on the FPGA design.
- The FIFO memory is an on-chip FIFO memory. It uses the memory resource inside the FPGA chip. Because of delays and bus contention problems during the SGDMA's accessing of the DDR SDRAM memory, the SGDMA may not load the pixel data in the pipeline timely. On the other hand, the SGDMA usually loads data more quickly than the rest of video pipeline processes them. The FIFO

memory can provide a data buffering for the rest of video pipeline. During the FPGA design, the FIFO memory is configured to accommodate 128 of 64-bit data from the upstream Timing Adapter One. One piece of data occupies eight symbols, eight bits for each symbol. The rate at which the FIFO passes data to the downstream Timing Adapter Two is one unit of 64 bits per clock cycle. The FIFO is designed with the Altera Avalon-ST backpressure support, which can stop the upstream from loading new data in the FIFO when it is full.

- The Timing Adapter Two is to make up the difference between the time length of data latency of the downstream Format Adapter One and that of the upstream FIFO memory. The former is zero units of latency; the latter is one unit of latency. They also rely on the FPGA design.
- The Format Adapter One is an Altera Avalon-ST data format adapter. It is used to transform 64-bit data to 32-bit data. Each piece of 64-bit data contains two pixel colour values as mentioned above. Each pixel RGB value is encoded in 32 bits by the Mesa-OpenGL implementation for the FPGA-based ES platform.
- The Pixel Converter is used to convert 32-bit pixel colour values to the 24-bit LCD data format. The final bytes in the 32 pixel colour values are intended only for the bit alignment.
- The Format Adapter Two is used to transform a 24-bit pixel RGB value to three separate 8-bit values because the rest of pipeline requires the red, green, and blue values of one pixel to be transferred separately in sequence. The Format Adapter Two accepts one 24-bit pixel data per clock cycle at its input, generates three 8-bit data, and send out one 8-bit data per clock cycle at its output.
- The Video Sync Generator is used to transmit the pixel data to the LCD data interface. It accepts a stream of pixel data from the upstream of pipeline, which have been encoded with eight bits of data stream width and three clock cycles per pixel. The control information that drives the display is added to the input video data to form the output video data. The Video Sync Generator transmits the control and colour data signals to the LCD data bus in sequence.

In this context, parallelism constitutes not only the parallel computation but also different channels and links between different hardware units working collaboratively in parallel in order to get a synergy of different parts in an entire system.

6.4.3 Co-processor in FPGA-based ES

The video pipeline in Altera FPGA-based ES also works as a co-processor. Since the Nios II processor does not take much control on the video pipeline after it starts streaming the

video signals and the video pipeline streams the pixels to the data interface of the LCD display automatically, the roles of the Nios II and the control driver of video pipeline are a host processor and co-processor, respectively.

The Nios II processor and control driver of video pipeline share the frame buffers in one of two DDR SDRAM memory banks that are on the board of Altera ESDK, Cyclone III Edition, outside the Cyclone III FPGA chip. They also share the memory space of the other DDR SDRAM memory bank. Descriptor buffers of the SGDMA in the control driver of video pipeline are located in the latter's bank memory space.

At the initialisation stage of the control driver of video pipeline, the system performs a series of tasks. It allocates the memory space for the frame buffers and descriptor buffers of the control driver of the video pipeline. It sets the resolution of the frame buffers and the descriptors for the frame buffers. It clears all the frame buffers. It opens the SGDMA, and registers the SGDMA callback function. Finally, it starts the control driver of the video pipeline. Typically, two frame buffers are allocated. One is for the Nios II processor to write when the other can be displayed by the video pipeline smoothly without interruption. The former is called the written frame buffer; the latter is the display frame buffer. When the display frame buffer has been displayed, these two frame buffer swap roles with each other. The original display frame buffer turns into the current written frame buffer, and the original written frame buffer is displayed by the video pipeline.

To address the changeable length of frame buffers and descriptor buffers, the memory is allocated from the heap and accommodates all the required number of frames and descriptors. All the descriptors form a descriptor chain in order to drive the streaming of video pipeline automatically.

When setting the descriptor buffers, the system has to calculate the number of bytes requested by the descriptor storage of a particular display in order to dynamically allocate the memory and pointers to a new frame at runtime.

During the drawing process for the Mesa-OpenGL applications of surface modelling and editing with the PAMA, the applications use the commands to call the Mesa-OpenGL auxiliary functions to make the Nios II processor process the image and write the results into the written frame buffer. The applications can also make requests for the video pipeline to display the image on the LCD screen by using a command like *glFlush()*. Since the main loop of an application can be executed repeatedly, both actions, i.e. writing one frame buffer of the Nios II processor and displaying the other frame buffer of the video pipeline, can be done in parallel.

In addition, FPGAs can have as many hardware kernels for the basic operation as possible. They are good for applications that use the integer arithmetic, and are computation-intensive for both spatial and temporal parallelism, albeit without much data

transfer between the FPGAs and microprocessors. Heavy data transfer may limit the capability of the spatial and temporal parallelism of FPGAs.

6.5 Chapter Summary

This chapter has categorised traditional parallelism. Even though this project takes a different view of parallelism, the discussion of traditional parallel computation provides a standpoint from which to extend it. Features of parallel processing are analysed in detail in order to achieve an in-depth understanding of parallelism. Two perspectives on parallelism, the application programmer's view and hardware builder's view, and two styles of parallelism, spatial and temporal, are presented. For methodologies of parallel processing, four decomposition techniques are studied; then two aspects for parallelism extension are presented, one for distributed systems for large data processing, and the other for the low-level hardware building. The role of co-processors in the parallelism is discussed as well. Finally, the pipeline in the graphics processing and co-processor in this research are discussed.

Chapter 7 Novel Algorithm for Surface Modelling and Editing, PAMA

From the discussion in Section 2.5, it has been known that the computer graphics is one of the most active fields in the computer science and technology. It consists of a wide range of research topics, including computational geometry, display algorithms, object modelling, rendering, shading, shadowing, solid representation, texture, 2D curve and 3D surface modelling, and others. 3D surface modelling and editing is one sub-field of the computer graphics. In this project, a novel algorithm for surface modelling and editing has been devised and implemented, which is called PAMA (progressive and mixing algorithm). The detail of PAMA will be discussed in Section 7.2.

In this chapter, in order to maintain the narrative coherence of the main chapters in the thesis, the discussion of the PAMA is kept concise and it focuses on the applications of the PAMA on the general-purpose computer platform. The results of the applications of the PAMA on the FPGA-based ES and the analysis between two group results on the general-purpose computer and FPGA-based ES will be presented in Chapter 8. The contents of this chapter include the preliminary, PAMA, surface modelling and editing with PAMA, different effects of shape parameters, and novel features of PAMA.

The rigorous mathematic exploration of the related theories will be presented in Appendix. Those include the in-depth discussions on the parametric continuities and geometric continuities, the geometric properties of Bézier-spline curves and surfaces, the principle for the construction of control vertices on common boundary curves of Bézier-spline surfaces with the PAMA, the twists and constructions of corner points of the patches with the PAMA, and the constructions of inside points with the PAMA, and the summarisation of the PAMA's continuities.

Before the discussion of the PAMA, the Beta-spline curves are introduced in Section 7.1 in order to provide a reference to the PAMA because the PAMA is an algorithm for the modelling and editing of Beta-like-spline surfaces.

7.1 Preliminary

In Section 2.5, one scheme to connect two cubic Bézier-spline curve sections together with the G^1 and G^2 conditions has been introduced. This method is first presented by Farin (Farin 1982), improved by Boehm (Boehm 1985), and used in Beta-spline curves by Barsky and DeRose (Barsky and DeRose 1989).

Given two Bézier curves, *S(w),* $w \in [0,1]$ and *T(u)*, $u \in [0,1]$ which have the degree of three with control polygons, *[S0, S1, S2, S3]* and *[T0, T1, T2, T3]*, respectively. To stitch them together with constraints of zero-, first-, and second-order geometric continuities, the following conditions should be matched. For the zero-order geometric continuity (G^0), the beginning of *T(u)* should be set as the end of *S(w)*,

T0 = *S3*. (7.1)

For the first-order geometric continuity (G^1), the parameter $\beta_1 > 0$ is involved. The first derivative direction at the beginning of *T(u)* and first derivative direction at the end of *S(w)* meet Equation 7.2,

$$T^{(1)}(0) = \beta_1 S^{(1)}(1) \tag{7.2}$$

where $T^{(1)}(0) = 3(T_1 - T_0)$, and $S^{(1)}(1) = 3(S_3 - S_2)$. From Equations 7.1 and 7.2, *T1* can be deduced from control points, *S2* and *S3*, with Equation 7.3,

$$T_1 = S_3 + \beta_1 (S_3 - S_2). \tag{7.3}$$

For the second-order geometric continuity (G^2), the parameter β_2 is involved. The second derivative direction at the beginning of *T(u)* is restricted by the first and second derivation directions at the end of *S(w)* as Equation 7.4,

$$T^{(2)}(0) = \beta_1^2 S^{(2)}(1) + \beta_2 S^{(1)}(1) \tag{7.4}$$

where $T^{(2)}(0) = 6(T_0 - 2T_1 + T_2)$, and $S^{(2)}(1) = 6(S_1 - 2S_2 + S_3)$. With Equations 7.1, 7.3, and 7.4, *T2* can be deduced from control points, *S1*, *S2*, and *S3* with Equation 7.5,

$$T_2 = \beta_1^2 S_1 - (2\beta_1^2 + 2\beta_1 + \beta_2 / 2) S_2 + (\beta_1^2 + 2\beta_1 + \beta_2 / 2 + 1) S_3. \tag{7.5}$$

Therefore, if Equations 7.1, 7.3, and 7.5 are all met, two curves *T(u)* and *S(w)* can be joined together with the second-order geometrical continuity. As the shape parameters, $\beta_1 > 0$ and β_2 can be adjusted freely, they provide local control on joined sections of a curve for interactive shape editing.

The above is the solution to Beta-spline curve modelling with geometric continuities. This research has extended it to surface modelling with the freedom of shape changing in

two parameter directions independently.

7.2 Progressive and Mixing Algorithm, PAMA

The new algorithm is called PAMA. Since the PAMA is created for the surface modelling and editing with user interactions in real time on the FPGA-based ES platform, there are two factors that must be considered.

The first factor is that its computation is restrained by the limited computation speed and storage space, which are 100 MHz of Nios II processor and 2 X 64 MBytes of DDR2 SDRAM, introduced in Chapter 4. The PAMA must be effective and efficient for the surface modelling and editing via user interactions in real time with a small footprint. Thus, the PAMA focuses on the constructed surfaces with meshes of less than 64K vertices, rather than the existent surfaces with larger grids of more than 1M vertices. The latter are usually taken as benchmarks and processed with the subdivision and deformation methods in the computer graphics. However, they cannot fit into the surface modelling and editing with user interactions in real time on the FPGA-based ES platform.

The other factor is that the applications of PAMA include both the modelling and editing of surfaces. This means that the PAMA can be used to create a surface from scratch with user interactions and edit it in a flexible way. The users can use the PAMA to create a new design and then change it in order to accomplish their work creation. For this purpose, changing the shape of a designed object is a progressive and controllable process. This progressive and controllable process must be guaranteed with the tools that the PAMA provides. The tools of PAMA equip users with the controllable methods to manipulate the shape change of the created object. The measures of the PAMA tools, especially shape parameters of $\beta_{u1}, \beta_{u2}, \beta_{v1}$, and β_{u2}, are varied values that represent varied effects of local shape changes referring to their previous states in the geometric sense, rather than the accuracy in the arithmetic sense. Thus, in the rest of the discussion in the thesis, the numbers of values of shape parameters, $\beta_{u1}, \beta_{u2}, \beta_{v1}$, and β_{u2}, as shown in examples, have to be treated as their geometric effects.

The PAMA can be used to construct smooth surfaces, open or closed, by stitching together bi-cubic Bézier-spline patches with local shape controls.

Given four original control points (shown as hollow-circle points in Figure 7.1), PAMA can be described in three steps,

Step 1 The first blending, to interpolate the first-interpolated points along the *u* and *v* directions with the adjacent original control points, respectively. This step mixes the adjacent original control points with the Beta constraints in the *u* and *v* directions, respectively. In Figure 7.1, the first-interpolated points, shown as square

points, sit on the edges formed with original control points.

Step 2 The second blending, to fit the original control points with the second-interpolated points. This step blends adjacent first-interpolated points by averaging Beta constraints in the *u* and *v* directions. The second-interpolated points, solid circles shown in Figure 7.1, replace original control points when rendered.

Step 3 The third blending, to interpolate the inside third-interpolated points inside each patch formed with original control points. This step blends adjacent first-interpolated points by averaging Beta constraints in the *u* and *v* directions. The third-interpolated points, triangles in Figure 7.1, sit inside the patches formed with original control points. By now, all the new points are generated to construct one bi-cubic Bézier-spline patch with local shape control.

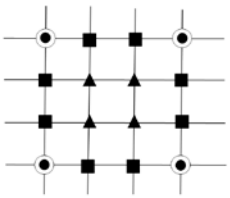


Figure 7.1 One Bi-cubic Bézier-Spline Patch Interpolated in the *u* (Horizontal) and *v* (Vertical) Directions with PAMA. Different Types of Points are Represented with Different Shapes in this Figure: Hollow Circles are Original Control Points; Squares are First-Interpolated Points; Solid Circles are Second-Interpolated Points; Triangles are Third-Interpolated Points.

The above described PAMA is just for one patch. The PAMA can be applied to construct complex surfaces, open or closed, based on original control points that can be designed initially and adjusted progressively through user interactions. As eight first-interpolated points and four third-interpolated points are added, each new patch has twelve more points than the original one. To continue the deduction of mathematic equations, the patch is put in a global surface and is assumed to be formed with control polygon, *[V(i, j),V(i+1, j),V(i+1, j+1),V(i, j+1)]*. The bi-cubic Bézier-spline patches constructed via the PAMA are shown in Figure 7.2.

In Figure 7.2, as the *V(i, j)*'s are the original control points, pairs of first-interpolated points *W(3i+1, 3j)* and *W(3i+2, 3j)* along the *u* direction are formed with the adjacent original control points, *V(i, j)* and *V(i+1, j)*. From the algorithm for drawing a cubic G^2 Beta-spline curve proposed by Barsky and DeRose (Barsky and DeRose 1990), the equation for assessment of *W(3i+1, 3j)* and *W(3i+2, 3j)* can be written as Equations 7.6 and 7.7:

$$W(3i+1,3j)=\frac{\left(1.0+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)\right)\cdot V(i,j)+\gamma_u(i,j)\cdot V(i+1,j)}{1.0+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)}; \tag{7.6}$$

$$W(3i+2,3j)=\frac{\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)\cdot V(i,j)+\left(1.0+\gamma_u(i,j)\right)\cdot V(i+1,j)}{1.0+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)} \tag{7.7}$$

where *W(l, k)* and *V(i, j)* are vectors with three coordinate values, and $\beta_{u1}(i,j)$, $\beta_{u2}(i,j)$ and $\gamma_u(i,j)$ are scalars. It is necessary to make variable substitutions,

$$\gamma_u(i,j)=\frac{2.0+2.0\cdot\beta_{u1}(i,j)}{\beta_{u2}(i,j)+2.0\cdot\beta_{u1}(i,j)\cdot\left(1.0+\beta_{u1}(i,j)\right)}. \tag{7.8}$$

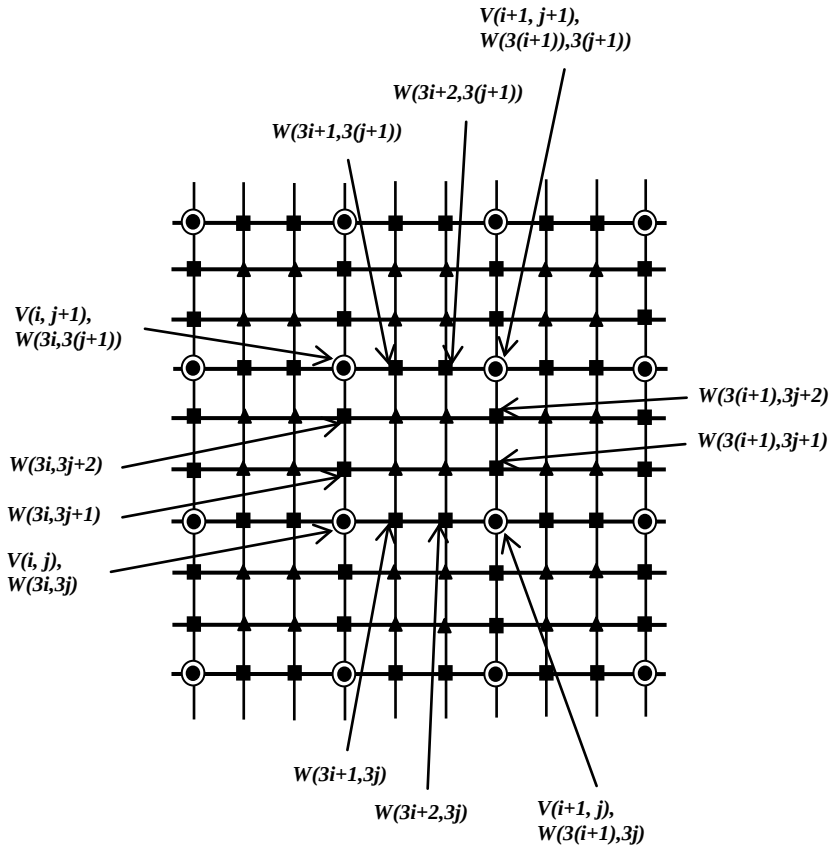


Figure 7.2 Bi-cubic Bézier-Spline Patches Constructed with PAMA in a Global Surface. The Middle Patch is Formed with the Original Control Polygon *[V(i, j),V(i+1, j),V(i+1, j+1),V(i, j+1)].* After Constructed with PAMA, the Middle Patch is a Mesh of 16 Interpolated Points, which are, from Bottom to Top and from Left to Right, *[W(3i, 3j),W(3i+1, 3j),W(3i+2, 3j),W(3(i+1), 3j), W(3i, 3j+1),W(3i+1, 3j+1),W(3i+2, 3j+1),W(3(i+1), 3j+1), W(3i,3j+2),W(3i+1, 3j+2),W(3i+2, 3j+2),W(3(i+1), 3j+2), W(3i, 3(j+1)),W(3i+1, 3(j+1)),W(3i+2, 3(j+1)), W(3(i+1),3(j+1))]*

Pairs of first-interpolated points *W(3i, 3j+1)* and *W(3i, 3j+2)* along the *v* direction are blended with the adjacent original control points, *V(i, j)* and *V(i, j+1)*. The equation for

evaluation of *W(3i, 3j+1)* and *W(3i, 3j+2)* are as Equations 7.9 and 7.10:

$$W(3i,3j+1)=\frac{\left(1.0+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)\right)\cdot V(i,j)+\gamma_v(i,j)\cdot V(i,j+1)}{1.0+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)}; \tag{7.9}$$

$$W(3i,3j+2)=\frac{\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)\cdot V(i,j)+\left(1.0+\gamma_v(i,j)\right)\cdot V(i,j+1)}{1.0+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)} \tag{7.10}$$

where $\beta_{v1}(i,j)$, $\beta_{v2}(i,j)$ and $\gamma_v(i,j)$ are scalars, and $\gamma_v(i,j)$ is written as follows

$$\gamma_v(i,j)=\frac{2.0+2.0\cdot\beta_{v1}(i,j)}{\beta_{v2}(i,j)+2.0\cdot\beta_{v1}(i,j)\cdot\left(1.0+\beta_{v1}(i,j)\right)}.$$

Second-interpolated points *W(3i, 3j)* are evaluated by blending adjacent first-interpolated points *W(3i,3(j-1)+2)*, *W(3i,3j+1)*, *W(3(i-1)+2,3j)*, and *W(3i+1,3j)* and averaging Beta constrains along both *u* and *v* directions. It is written as follows:

$$W(3i,3j)=$$

$$\frac{\beta_{u1}(i,j)\cdot W(3(i-1)+2,3j)+W(3i+1,3j)+\beta_{v1}(i,j)\cdot W(3i,3(j-1)+2)+W(3i,3j+1)}{2.0+\beta_{u1}(i,j)+\beta_{v1}(i,j)}. \tag{7.11}$$

The inside third-interpolated points are evaluated by blending adjacent first-interpolated points and averaging Beta constrains along both *u* and *v* directions with a slight variation to reduce the computational cost. The deduction equations are written as Equations 7.12, 7.13, 7.14, and 7.15,

$$W(3i+1,3j+1)=\frac{1}{2}\cdot\left(\frac{\left(1.0+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)\right)\cdot W(3i,3j+1)+\gamma_u(i,j)\cdot W(3(i+1),3j+1)}{1.0+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)}+\right.$$

$$\left.\frac{\left(1.0+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)\right)\cdot W(3i+1,3j)+\gamma_v(i,j)\cdot W(3i+1,3(j+1))}{1.0+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)}\right); \tag{7.12}$$

$$W(3i+1,3j+2)=$$

$$\frac{1}{2}\cdot\left(\frac{\left(1.0+\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)\right)\cdot W(3i,3j+2)+\gamma_u(i,j+1)\cdot W(3(i+1),3j+2)}{1.0+\gamma_u(i,j+1)+\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)}\right.$$

$$\left.+\frac{\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)\cdot W(3i+1,3j)+\left(1.0+\gamma_v(i,j)\right)\cdot W(3i+1,3(j+1))}{1.0+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)}\right); \tag{7.13}$$

$$W(3i+2,3j+1)=\frac{1}{2}\cdot\left(\frac{\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)\cdot W(3i,3j+1)+\left(1.0+\gamma_u(i,j)\right)\cdot W(3(i+1),3j+1)}{1.0+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)}\right.$$

$$\left.+\frac{(1.0+\beta_{v1}^2(i+1,j+1)\cdot\gamma(i+1,j+1))\cdot W(3i+2,3j)+\gamma_v(i+1,j)\cdot W(3i+2,3(j+1))}{1.0+\gamma_v(i+1,j)+\beta_{v1}^2(i+1,j+1)\cdot\gamma_v(i+1,j+1)}\right); \tag{7.14}$$

and

$$W(3i+2,3j+2)=$$

$$\frac{1}{2}\cdot\left(\frac{\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)\cdot W(3i,3j+2)+\left(1.0+\gamma_u(i,j+1)\right)\cdot W(3(i+1),3j+2)}{1.0+\gamma_u(i,j+1)+\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)}+\right.$$

$$\frac{\beta_{v1}^2(i+1,\, j+1)\cdot\gamma_v(i+1,\, j+1)\cdot W(3i+2,3j)+(1.0+\gamma_v(i+1,\, j))\cdot W(3i+2,3(j+1))}{1.0+\gamma_v(i+1,\, j)+\beta_{v1}^2(i+1,\, j+1)\cdot\gamma_v(i+1,\, j+1)}). \qquad (7.15)$$

In fact, Equations 7.11, 7.12, 7.13, 7.14 and 7.15 are not deduced strictly from the tensor product of Beta-spline blending functions. The whole tensor product of Beta-spline blending functions can cause a large computational cost of multiplication applications. Nonetheless these equations meet the basic demands for parameterisation. The sum of their basic functions is equal to one. For Equation 7.11, it can be proved by the next formula,

$$\frac{\beta_{u1}(i,\, j)+1.0+\beta_{v1}(i,\, j)+1.0}{2.0+\beta_{u1}(i,\, j)+\beta_{v1}(i,\, j)}=1.$$

By now, all the interpolated points for inside patches have been generated with the PAMA. For an open surface, points on the boundaries should be constructed with a slightly adjusted means. Usually, there are three cases for boundary points to be constructed in a degenerated way since not all the interpolated points in the two parameter directions are available for their construction. Figure 7.3 shows the case when the points in just *u* direction are available for its construction. In this case, the construction equation of *W(3i, 3j)* is as follows,

$$W(3i,3j)=\frac{\beta_{u1}(i,\, j)W(3(i-1)+2,3j)+W(3i+1,3j)}{1.0+\beta_{u1}(i,\, j)}. \qquad (7.16)$$

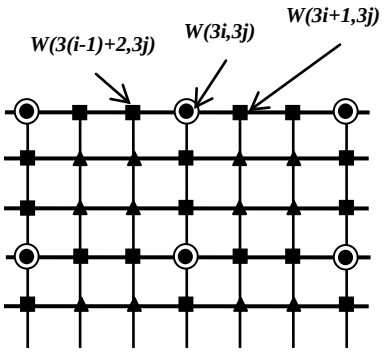


Figure 7.3 Construction of a Point on the Boundary with Interpolated Points Available in the *u* Direction.

Figure 7.4 shows the case when the interpolated points in just *v* direction are available to support its construction. In this case, the construction equation of *W(3i, 3j)* is as follows,

$$W(3i,3j)=\frac{\beta_{v1}(i,\, j)W(3i,3(j-1)+2)+W(3i,3j+1)}{1.0+\beta_{v1}(i,\, j)}. \qquad (7.17)$$

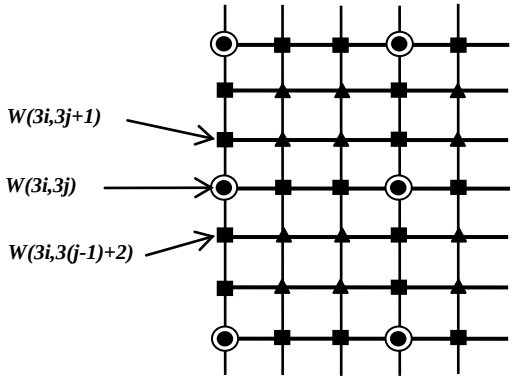


Figure 7.4 Construction of a Point on the Boundary with Interpolated Points Available in the *v* Direction

If *W(3i, 3j)* is located on a corner, as shown in Figure 7.5, where the interpolated points in neither *u* nor *v* directions are enough to support its construction. In this case, *W(3i, 3j)* retains the value of *V(i, j)*.

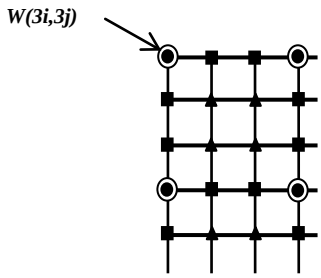


Figure 7.5 Construction of a Point on a Corner

7.3 Surface Modelling and Editing with PAMA

When used in the surface modelling and editing, the PAMA can give interactive users the following tools to shape surfaces. All the shape parameters, $\beta_{u1}(i,j), \beta_{v1}(i,j), \beta_{u2}(i,j),$ and $\beta_{v2}(i,j)$, for each control point, are initialized to *1.0*. This initialisation is done in all the following examples.

- **Varying Position (VP)**: The position of a control point, *V(i, j)*, is changed by varying values of its 3D coordinates. The effect of geometric variation from Figure 7.6(a) to Figure 7.6(b) is generated with VP.
- **Varying Beta-u-one (VBU1) and Varying Beta-v-one (VBV1)**: The former is to change the $\beta_{u1}(i,j)$ parameter of a control point; the latter is to change its $\beta_{v1}(i,j)$ parameter. For each control point, $\beta_{u1}(i,j)$ and $\beta_{v1}(i,j)$ can be

changed independently. The effect of geometric variation from Figure 7.7(b) to Figure 7.7(c) is produced with VBV1 by increasing the $\beta_{v1}(i, j)$ of just the middle control point at the top of the chair back to *20.0*.

- **Varying Beta-u-two (VBU2) and Varying Beta-v-two (VBV2)**: The former is to change the $\beta_{u2}(i, j)$ parameter of a control point; the latter is to change its $\beta_{v2}(i, j)$ parameter. For each control point, $\beta_{u2}(i, j)$ and $\beta_{v2}(i, j)$ can be also varied separately. The effect of geometric variation from Figure 7.7(b) to Figure 7.7(d) is yielded with VBV2 by increasing the $\beta_{v2}(i, j)$'s of three middle control points at the top of the chair back to *70.0*, separately.
- **Varying Group Position (VGP), Varying Group Beta-u-one (VGBU1), Varying Group Beta-v-one (VGBV1), Varying Group Beta-u-two (VGBU2) and Varying Group Beta-v-two (VGBV2)**: All VP, VBU1, VBV1, VBU2 and VBV2 can be applied to control points, separately or in groups. Sometimes, manipulations of a group of control points are uniform. Accordingly, the manipulations can be done in the same way at the same time. Figures 7.6(c) and 7.6(d) are generated with VGP. Figure 7.7(b) is created with VGP, and Figure 7.7(d) is reshaped with VGBV2.
- **Varying Mixing Beta-one (VMB1) and Varying Mixing Beta-two (VMB2)**: The former is to vary both VBU1 and VBV1 of a control point simultaneously; the latter is to change both VBU2 and VBV2 of a control point at the same time. Figures 7.8(c) and 7.8(d) show the use of VMB2. The differences between VBU1, VBV1 and VMB1 and between VBU2, VBV2 and VMB2 will be discussed later in Section 7.4.

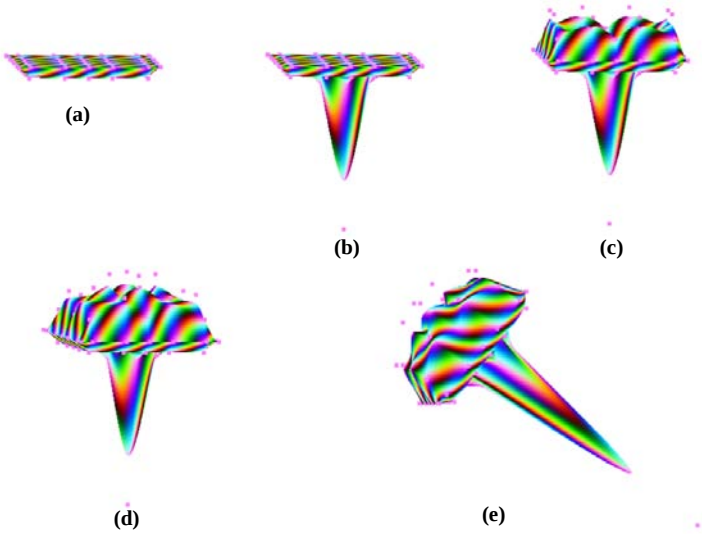


Figure 7.6 Changing from a Flat Box to a Burning Torch. (a) The Flat Box, Initially Modelled Surface; (b) The Torch Handle Shaped with VP; (c) Outer Flames Shaped with VGP; (d) Inner Flames Shaped with VGP; (e) The Burning Torch.

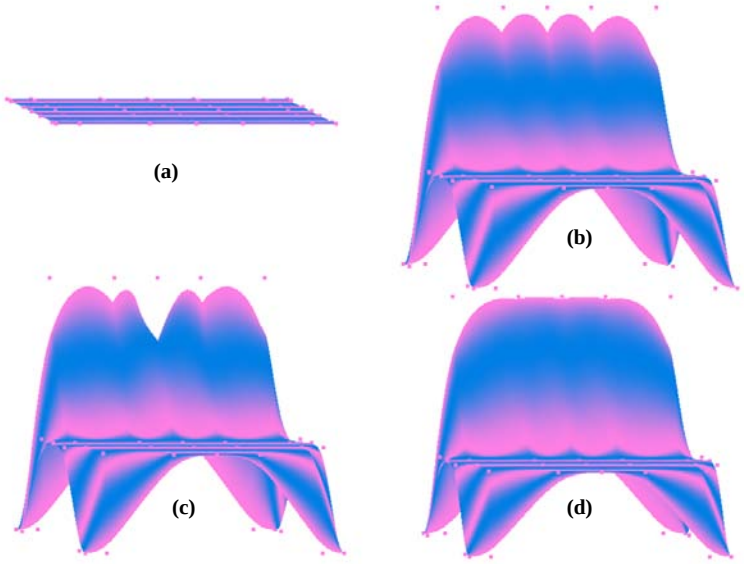


Figure 7.7 Changing from a Flat Board to Chair. (a) The Flat Board, the Initially Modelled Surface; (b) The Semi-Finished Chair Shaped with VGP; (c) The Deformed Chair Shaped with VBV1 by Increasing $\beta_{v1}(i, j)$ of just the Middle Top Control Point on the Chair Back to *20.0*; (d) The Completed Chair Reshaped from (b) with VGBV2 by Increasing $\beta_{v2}(i, j)$'s of Three Middle Control Points at the Top of Chair Back to *70.0*.

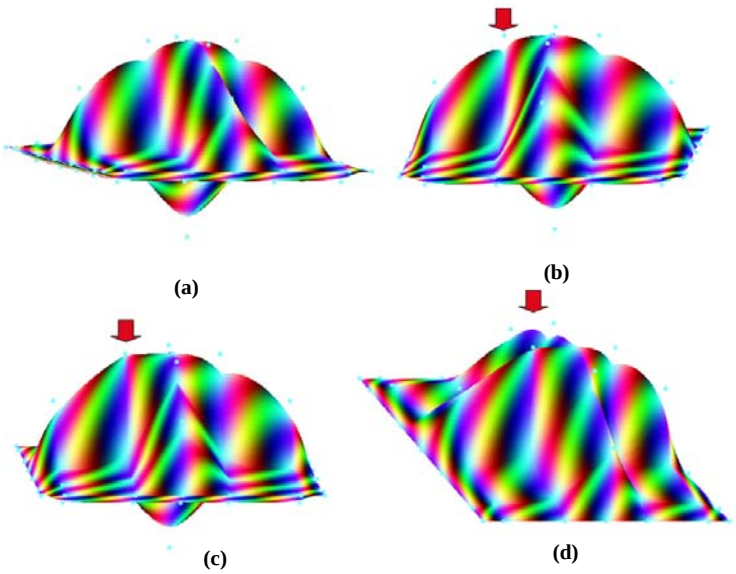


Figure 7.8 Changing of an Imaginary Flying Object. (a) The Imaginary Flying Object; (b) The Flying Object Reshaped with VBU2 by Increasing only the $\beta_{u2}(i, j)$ of the Control Point on the Shoulder Marked with a Red Arrow and with a Notch Left; (c) The Flying Object Reshaped with VMB2 by Increasing $\beta_{u2}(i, j)$ and $\beta_{v2}(i, j)$ of the Same Control Point Together and Without a Notch Left; (d) The Same Fly Object as (c) Viewed in a Different Angle.

To show the shaping process and geometric effects of different tools carefully, four more examples are given. Figure 7.9 presents how an ashtray can be created from a flat board. The flat board is the initial model with necessary control points. The control points can be determined by users for the design purpose and consist of the points that are edited with user interactions. Figure 7.9(b) is shaped with VGP from the flat board. Figure 7.9(c) is reshaped from Figure 7.9(b) with VGBU2 or VGBV2 according to which direction, *u* or *v*, the related control points sit along. Among the manipulated control points, eight are processed with VGBU2 while six processed with VGBV2. Figure 7.9(c) shows the geometric effect of increasing $\beta_{u2}(i, j)$ or $\beta_{v2}(i, j)$ of these control points to *50.0*. Figure 7.9(d) is the completed ashtray viewed from the bottom.

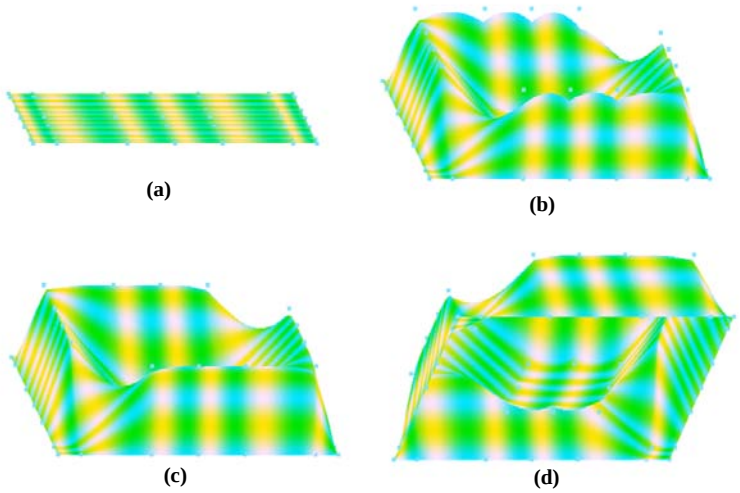


Figure 7.9 Changing from a Flat board to an Ashtray. (a) The Flat Board; (b) The Semi-Finished Ashtray Created with VGP; (c) The Completed Ashtray Shaped with VGBU2 or VGBV2 by Increasing $\beta_{u2}(i, j)$ or $\beta_{v2}(i, j)$ of Relative Control Points to *50.0*; (d) The Completed Ashtray Viewed from the Bottom.

The second example is the construction of a clamshell box, as shown in Figure 7.10. The construction is started from Figure 7.10(a) by pulling upwards control points along three sides of a flat board with VGP. Figure 7.10(b) is generated by pulling upwards control points on the fourth side of the flat board with VGP. By continuing pulling these control points but in the horizontal direction with VGP, a rough clamshell box is shaped, as shown in Figure 7.10(c). Then control points along the connection side between the box body and lid are pulled to their positions with VGP. The connection side is curved, seen in Figure 7.10(d). Finally, the connection side is straightened with VGBU2. Figure 7.10(e) shows the geometric effect of increasing $\beta_{u2}(i, j)$'s of five middle control points to *50.0*. Figure 7.10(f) shows the completed clamshell box viewed from the side.

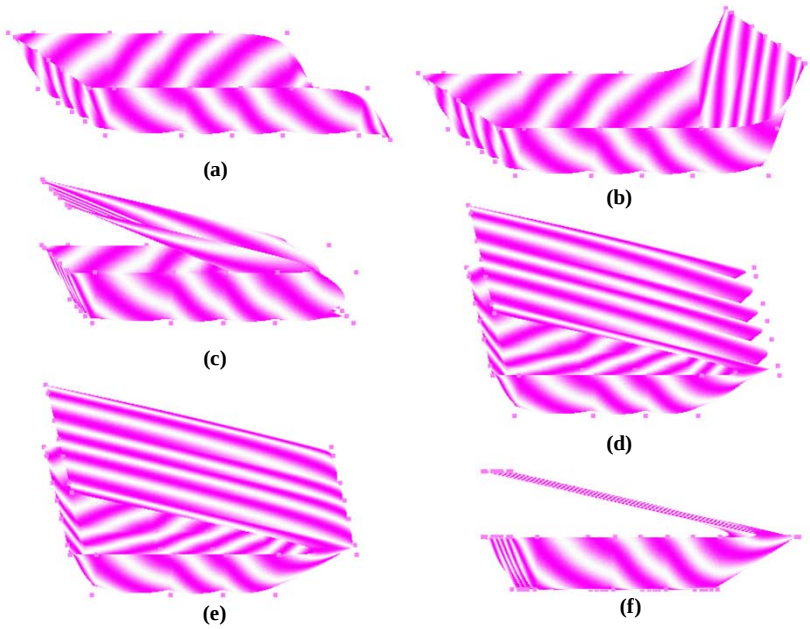


Figure 7.10 Shaping of a Clamshell Box. (a) A Semi-Finished Box Created with the VGP Operation from a Flat Board; (b) A Semi-Finished Box with a Half Lid Reshaped with VGP; (c) A Semi-Finished Box with a Lid Reshaped with VGP; (d) A Semi-Finished Box with a Full Lid Reshaped with VGP; (e) A Finished Clamshell Box Completed with the VGBU2 Operation by Increasing $\beta_{u2}(i, j)$'s of Five Middle Control Points on the Connection Side Between the Box and Lid to *50.0*; (f) The Clamshell Box Viewed from one Side.

The third example is a series of changes from a flat board to a table, a chair and finally a double chair, as shown in Figure 7.11. It begins with a flat board. The flat board is turned into a table with VGP by pulling downwards four legs in Figure 7.11(a). The table is changed into a semi-finished chair with VGP by pulling upwards the chair back in Figure 7.11(b). The chair is completed with VGBV2. Figure 7.11(c) shows the geometric effect of increasing the $\beta_{v2}(i, j)$'s of three middle control points at the top of the chair back to *50.0*. Finally, the completed chair becomes a double chair with VBV1. Figure 7.11(d) shows the geometric effect of increasing $\beta_{v1}(i, j)$ of just the middle control point at the top of the chair back to *4.0,* which is the double chair.

The fourth example is the construction of a loose bud, given in Figure 7.12. The process starts from Figure 7.12(a) that is a disc with necessary control points. Figure 7.12(b) is generated with VGP to pull out the relative control points to the proper positions. Figure 7.12(c) is reshaped with VBU1 (or VBV1) separately. It shows the geometric effect of increasing $\beta_{u1}(i, j)$'s (or $\beta_{v1}(i, j)$'s) of the related control points to *11.0*. Figure 7.12(d) is generated from Figure 7.12(b) with VMB1. The geometric effect is yielded by increasing simultaneously $\beta_{u1}(i, j)$'s and $\beta_{v1}(i, j)$'s of the related control points to *11.0*. Figure 7.12(e) is the completed loose bud viewed from the side.

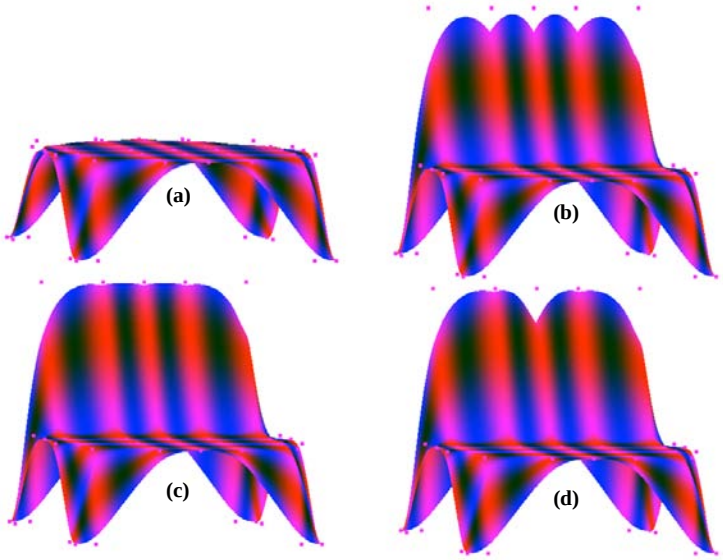


Figure 7.11 A Series of Changing from a Flat Board to a Table, a Chair and Finally a Double Chair. (a) A Table Created with VGP from a Flat Board; (b) The Semi-Finished Chair Reshaped with VGP from the Table; (c) A Completed Chair Reshaped with VGBV2 by Increasing $\beta_{v2}(i, j)$'s of Three Middle Control Points at the Top of the Chair Back to *50.0*; (d) A Completed Double Chair Reshaped with VBV1 by Increasing $\beta_{v1}(i, j)$ of the Middle Control Point at the Top of the Chair Back to *4.0*.

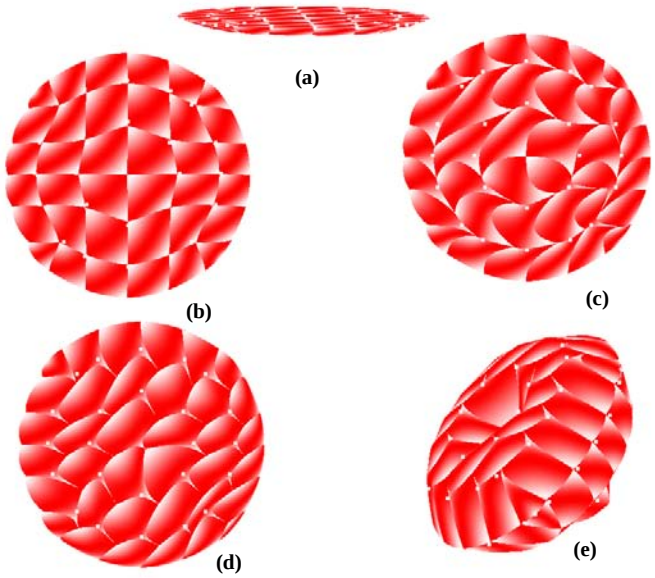


Figure 7.12 Construction of a Loose Bud. (a) A Disc, the Initially Modelled Surface; (b) The Image Reshaped with VGP; (c) Image Reshaped from (b) with VBU1 (or VBV1) by Increasing $\beta_{u1}(i, j)$'s (or $\beta_{v1}(i, j)$'s) of Relative Control Points to *11.0*; (d) Image Reshaped from (b) with VMB1 by Increasing Simultaneously $\beta_{u1}(i, j)$'s and $\beta_{v1}(i, j)$'s of Relative Control Points to *11.0*; (e) Image of the Completed Loose Bud Viewed from a Different Angle.

7.4 Different Effects of Shape Parameters

To clearly describe different effects of shape parameters, further operations of the tools introduced above are discussed in detail.

7.4.1 Skewing in *u* or *v* Directions

Skewing in the *u* and *v* directions can be done with VBU1 and VBV1, respectively. Increasing $\beta_{u1}(i, j)$ makes the skewing depart from the control point in the *u* direction while decreasing $\beta_{u1}(i, j)$ has skewing approach to the control point in the *u* direction. Increasing and decreasing $\beta_{v1}(i, j)$ have the same effect in the *v* direction. Attention must be paid to keep $\beta_{u1}(i, j) > 0$ and $\beta_{v1}(i, j) > 0$.

In Figures 7.13 and 7.14, images are different results reshaped from Figure 7.13(a). All of them are manipulated on the middle control point at the top of the chair back. Figures 7.13(b) and 7.13(d) show the geometric effect of increasing $\beta_{u1}(i, j)$ to *6.0*, viewed from the front and back, respectively. Figures 7.13(c) and 7.13(e) show the geometric effect of the result of decreasing $\beta_{u1}(i, j)$ to *0.1*, seen from the front and back, respectively.

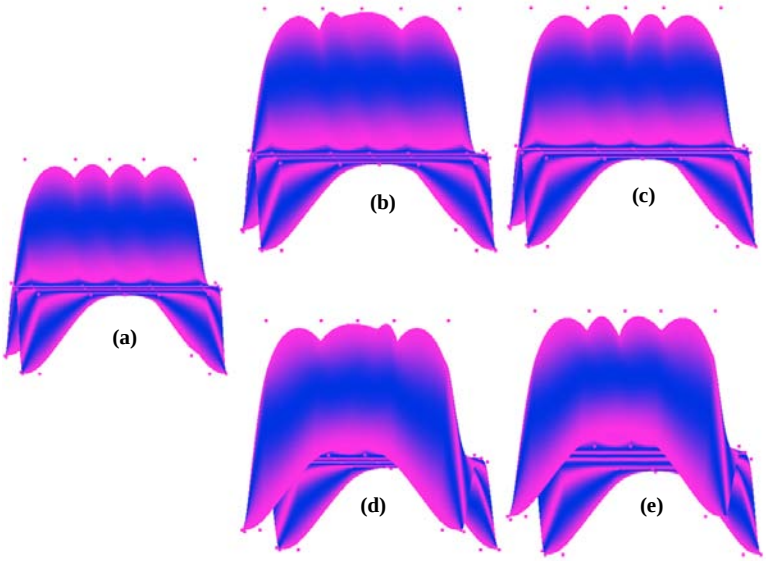


Figure 7.13 Chair Images Reshaped with VBU1 on the Middle Control Point at the Top of Chair Back. (a) The Image Reshaped in the Similar Way as Figure 7.7(b); (b) Image Reshaped by Increasing $\beta_{u1}(i, j)$ of the Control Point to *6.0* with VBU1; (c) Image Reshaped by Decreasing $\beta_{u1}(i, j)$ to *0.1* with VBU1; (d) The Same Chair Image as (b) but Viewed from the Back; (e) The Same Chair Image as (c) but Viewed from the Back.

Figure 7.14(a) and 7.14(c) show the geometric effect of the same result of increasing $\beta_{v1}(i, j)$ to *6.0*, viewed from the front and back, respectively. Figures 7.14(b) and 7.14(d)

show the geometric effect of the result of decreasing $\beta_{v1}(i, j)$ to *0.1*, seen from the front and back, respectively.

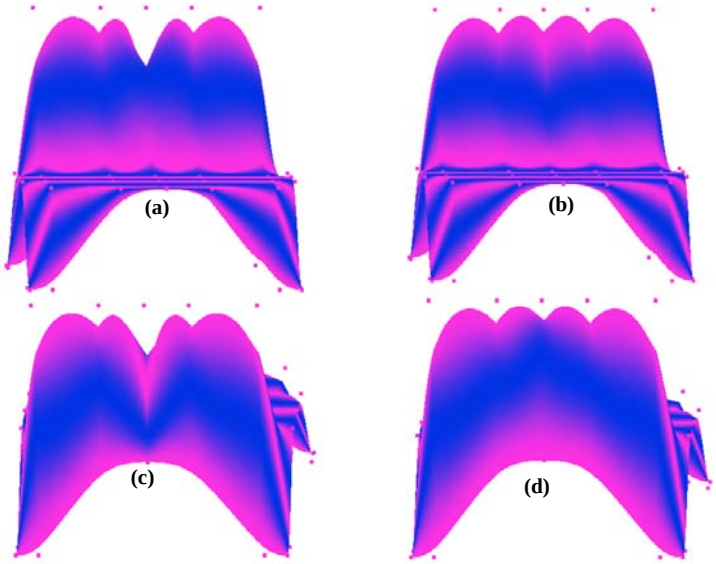


Figure 7.14 Chair Images Reshaped from Figure 7.13(a) with VBV1 on the Middle Control Point at the Top of the Chair Back. (a) The Image Reshaped by Increasing $\beta_{v1}(i, j)$ of the Control Point to *6.0* with VBV1; (b) Image Reshaped by Decreasing $\beta_{v1}(i, j)$ to *0.1* with VBV1; (c) The Same Chair Image as (a) but Viewed from the Back; (d) The Same Chair Image as (b) but Viewed from the Back.

7.4.2 Tenseness in *u* or *v* Directions

The tenseness pulling towards a control point in the *u* direction is done with VBU2 by increasing its $\beta_{u2}(i, j)$. The tenseness towards the control point in the *v* direction is done by using VBV2 by increasing its $\beta_{v2}(i, j)$. The smaller the value of the $\beta_{u2}(i, j)$ or $\beta_{v2}(i, j)$ is, the more obvious the tenseness change is.

Figures 7.15(a)-(d) show the various geometric effects with different values of $\beta_{u2}(i, j)$. Values of $\beta_{u2}(i, j)$ of the five control points on the connection side between the box and lid are *1.0*, *6.0*, *12.0*, and *50.0*, respectively for Figures 7.15(a), (b), (c) and (d). Figures 7.15(b)-(d) are done with VGBU2 while the box construction is shown in Figure 7.10.

7.4.3 Skewing in Both *u* and *v* Directions

Increasing or decreasing $\beta_{u1}(i, j)$ and $\beta_{v1}(i, j)$ together with VMB1 can make the skewing depart from or approach to a control point in both *u* and *v* directions simultaneously.

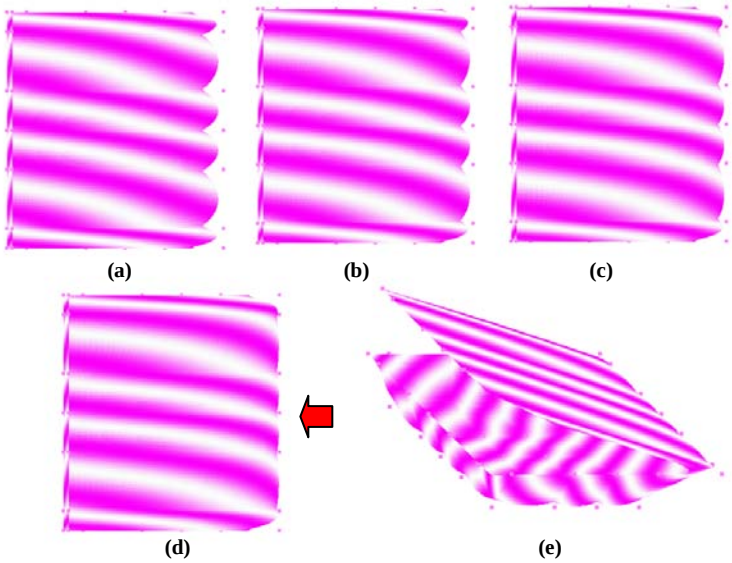


Figure 7.15 Smoothing the Connection Side of a Clamshell Box by Using VGBU2 with Different $\beta_{u2}(i, j)$ Values. (a) The Image for Five Middle Control Points on Connection Side with $\beta_{u2}(i, j)$ Value of *1.0*; (b) Image with $\beta_{u2}(i, j)$ Value of *6.0*; (c) Image with $\beta_{u2}(i, j)$ Value of *12.0*; (d) Image with $\beta_{u2}(i, j)$ Value of *50.0*; (e) Image for Completed Clamshell Box Viewed from a Different Angle.

Sometimes, it is necessary to balance the skewing in both *u* and *v* directions. Figure 7.16(b) is reshaped from Figure 7.16(a) with just VBU1 (or just VBV1). It shows the geometric effect of increasing $\beta_{u1}(i, j)$'s (or $\beta_{v1}(i, j)$'s) of the related control points to *6.0*. Figure 7.16(c) is reshaped from Figure 7.16(a) with VMB1. It also shows the geometric effect of increasing both $\beta_{u1}(i, j)$'s and $\beta_{v1}(i, j)$'s of the related control points to *6.0*. It can be seen that varying $\beta_{u1}(i, j)$ and $\beta_{v1}(i, j)$ together makes petals, each of which is one Bézier-spline patch, more crowded together than varying only $\beta_{u1}(i, j)$ (or $\beta_{v1}(i, j)$) does.

7.4.4 Tenseness in Both *u* and *v* Directions

Increasing or decreasing $\beta_{u2}(i, j)$ and $\beta_{v2}(i, j)$ together with VMB2 can have the tenseness pulling towards or pushing away from a control point in both *u* and *v* directions simultaneously.

In some situations, it is not enough to use only VBU2 or VBV2 on a control point. Figure 7.17(b) (or Figure 7.8(b)) is reshaped with VBU2 by increasing only $\beta_{u2}(i, j)$ of the control point on the shoulder marked with a red arrow of Figure 7.17(a) (or Figure 7.8(a)). There is a notch left, which is marked with the red arrow in Figure 7.17(b). But when VMB2 is used, the surface patch is pulled towards the control point totally without any notch left, as shown in Figure 7.17(c) (or Figure 7.8(c)).

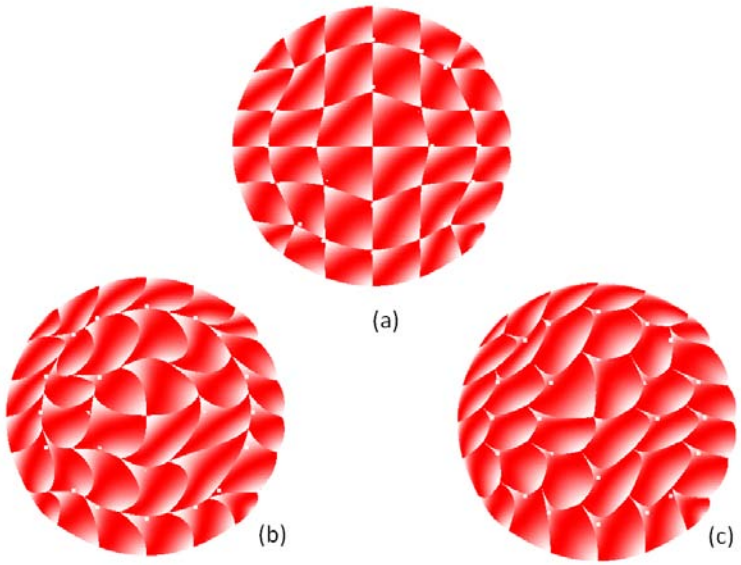


Figure 7.16 Difference between Geometric Effects of VMB1 and VBU1 (or VBV1). (a) The Loose Bud Shaped with VGP; (b) Image Reshaped from (a) with VBU1 (or VBV1) by Increasing only $\beta_{u1}(i, j)$'s (or $\beta_{v1}(i, j)$'s) of Relative Control Points to *6.0*; (c) Image Reshaped from (a) with VMB1 by Increasing Both $\beta_{u1}(i, j)$'s and $\beta_{v1}(i, j)$'s of Relative Control Points to *6.0*.

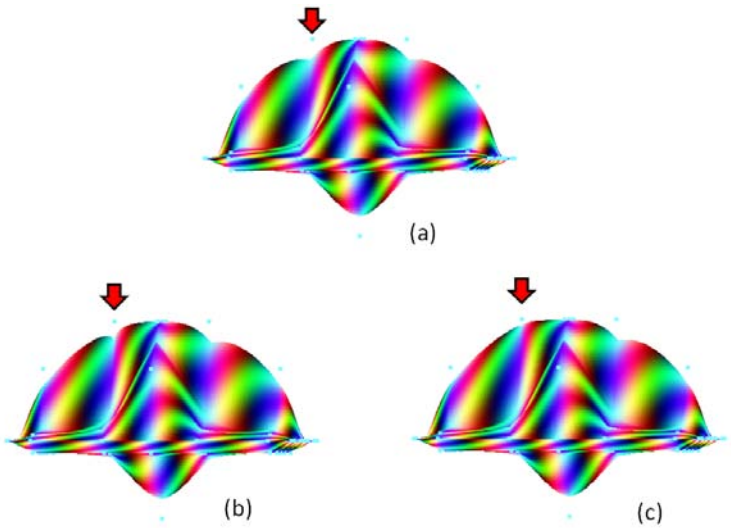


Figure 7.17 Difference Between Geometric Effects of VMB2 and VBU2. (a) The Imaginary Flying Object; (b) Image Reshaped from (a) with VBU2 by Increasing only $\beta_{u2}(i, j)$ of the Control Point on the Shoulder Marked with a Red Arrow and with a Notch Left; (c) Image Reshaped from (a) with VMB2 by Increasing both $\beta_{u2}(i, j)$ and $\beta_{v2}(i, j)$ of the Control Points without any Notch Left.

7.5 Novel Features of PAMA

The PAMA is a novel algorithm which can be applied, independently, in CAD, CAGD, computer-aided art creation, and other related fields for surface modelling and editing. With the PAMA, shape parameters are added to construct surfaces with control points. The surface shapes can be modified by changing the shape parameters for the design purpose. PAMA tools are flexible and suitable for practical applications. The features are outlined below:

- Values of position and shape parameters of each control point are controlled through user interactions. For instance, the user can decide interactively the length of the torch handle in Figures 7.6 and 7.18. The end points of torch handles in Figures 7.18 (b) – (g) are moved in the same direction but by different lengths of 4, *8, 16, 36, 41,* and *60* units, respectively, from the initial position shown in Figure 7.18(a). In Figures 7.15 and 7.19, the user can decide the smoothness extent of the connection side of the clamshell box in two parameter directions, *u* and *v*. In the *u* direction, the curve is straightened smoothly while in the *v* direction, the sharp fold is held. Figure 7.19(a) shows the shape fold in the *v* direction on the connection side. Figures 7.19(b) – (k) show the different smoothness extents of the connection side in the *u* direction, with different $\beta_{u2}(i, j)$ values of *0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 14.0, 24.0, 34.0,* and *50.0*, respectively.

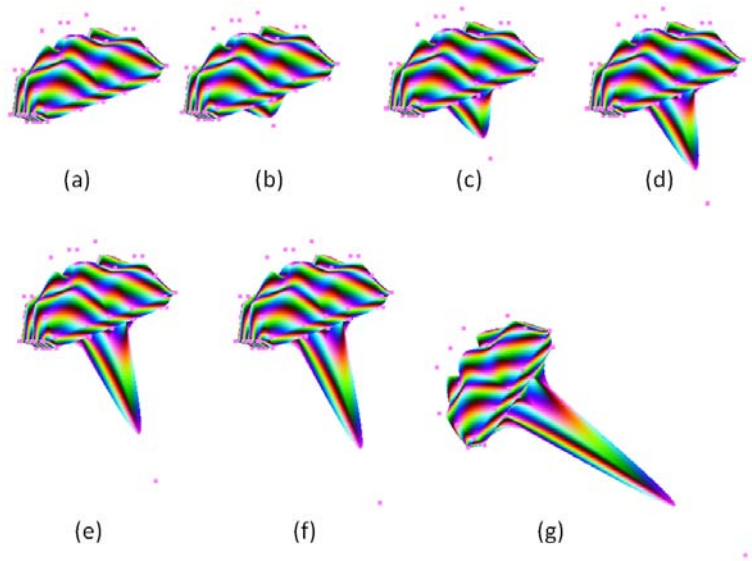


Figure 7.18 Images to Show the Control on the Position of the Control Point at the End of the Torch Handle. (a) The Initial Position; (b) – (g) Positions Moved by *4, 8, 16, 36, 41,* and *60* Units, Respectively, in the Same Direction.

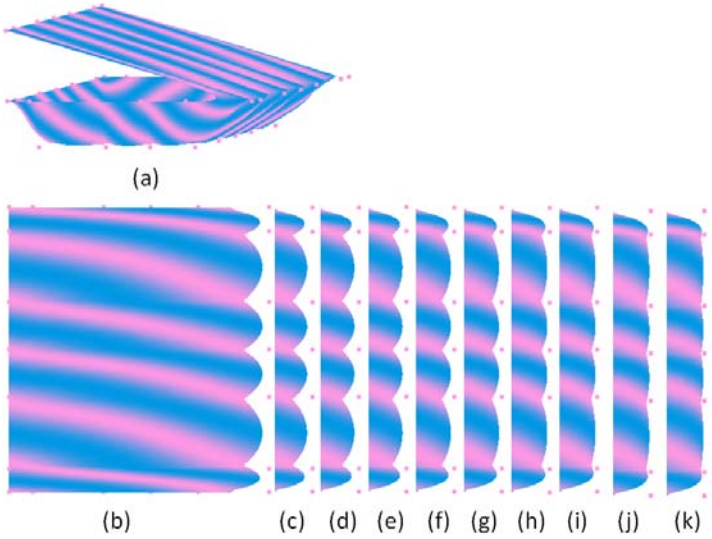


Figure 7.19 Images to Show the Control on the Different Smoothness Extents of the Connection Side of the Clamshell Box. (a) The Shape Fold in the *v* Direction of the Connection Side; (b) – (k) The Curves in the *u* Direction of the Connection Sides Reshaped with Different $\beta_{u2}(i, j)$ Values of *0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 14.0, 24.0, 34.0,* and *50.0*, respectively.

- The user can adjust the object shape at any time by interweaving different tools. The sequence of processing steps does not affect the final result if the sum of application of each used tool is kept uniform in different processes. For example, Figures 7.20 and 7.21 show two different processing chains to create an ashtray with the same shape, respectively. Figure 7.20 is the chain that the applications of VGP are kept first until the four sides are formed, and then the applications of VGBU2 or VGBV2 are done continually until the shape of ashtray is accomplished. Figure 7.21 is a different chain from Figure 7.20. The chain in Figure 7.21 is that the applications of VGP and VGBU2 (or VGBV2) are done four times in turn. The condition that the same shape of ashtray can be attained in both ways, as shown in Figures 7.20(h) and 7.21(h), is that the two ways have the same total amounts of VGP, VGBV2, and VGBV2 applications, respectively, for each of control points. That is, the sum of varied amount of VGP applications for each of control points of all the steps of Figure 7.20 is equal to the sum of VGP applications to the same control point of all the steps of Figure 7.21. The sum of amount of VGBU2 (or VGBV2) applications for each of control points of all the steps in Figure 7.20 are equal to one of VGBU2 (or VGBV2) applications for the same point of all the steps in Figure 7.21.
- PAMA tools can help the user accumulate experience and inspiration in the modelling process. The user can edit the model to satisfy the artistic intuition or

creative purpose.

- The control points can be defined by users. It provides them the freedom to manipulate the work at the very beginning of designs.
- The PAMA can be applied in modelling and editing of open and closed surfaces. In the examples, Figures 7.7, 7.9 and 7.10 are open surfaces while Figures 7.6, 7.8, and 7.12 are closed surfaces.

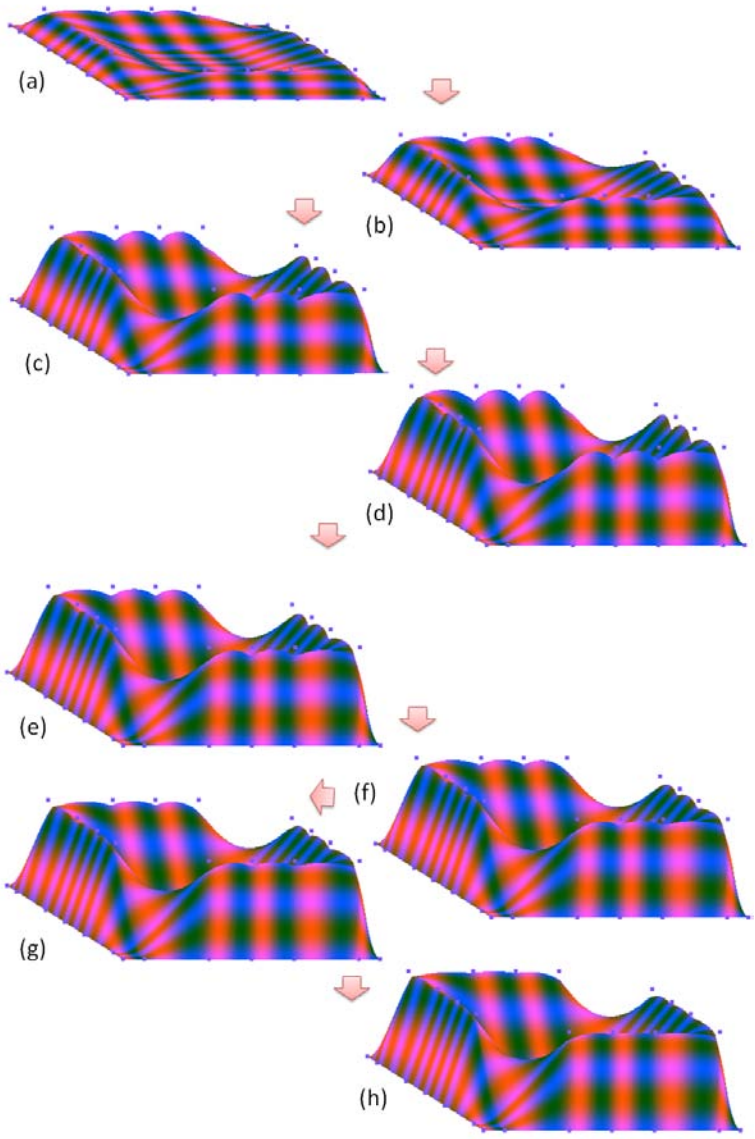


Figure 7.20 A Processing Chain to Create an Ashtray. (a) – (d) Images Reshaped with a List of VGPs; (e) – (h) Images Reshaped from (d) with a List of VGBU2s or VGBV2s. (h) The Completed Ashtray.

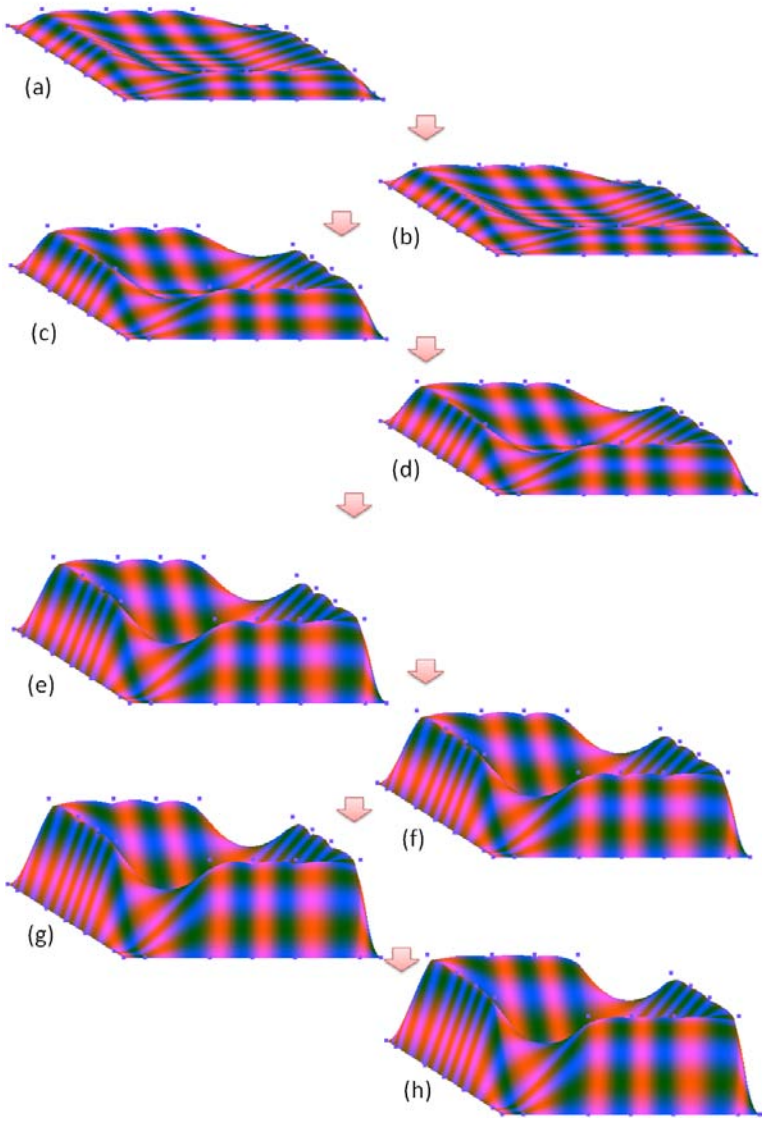


Figure 7.21 A Different Processing Chain from Figure 7.20 to Create an Ashtray with the Same Shape as Figure 7.20(h). (a) Image Shaped with VGP; (b) Image Reshaped from (a) with VGBU2 or VGBV2; (c) Image Reshaped from (b) with VGP; (d) Image Reshaped from (c) with VGBU2 or VGBV2; (e) Image Reshaped from (d) with VGP; (f) Image Reshaped from (e) with VGBU2 or VGBV2; (g) Image Reshaped from (f) with VGP; (h) The Completed Ashtray Reshaped from (g) with VGBU2 or VGBV2.

7.6 Chapter Summary

In this chapter, the novel algorithm for surface modelling and editing, PAMA, is presented. Before the presentation of PAMA, the preliminary is introduced. The PAMA algorithm and surface modelling and editing with PAMA are introduced. Different effects of shape parameters are also discussed. Finally, the novel features of PAMA are outlined.

The applications of the PAMA in this chapter are programmed and verified on a general

purpose computer environment of the SAMAUNG's R480 laptop computer. The PAMA is also ported to the FPGA-based ES that is discussed in the previous chapters. Chapter 8 will detail the results of the PAMA applications on the FPGA-based ES.

Chapter 8 Results of Surface Modelling and Editing with PAMA on FPGA-based ES

Applications on surface modelling and editing with the PAMA on the general-purpose computer environment have been presented in Chapter 7. In this chapter, results of surface modelling and editing with the PAMA on the FPGA-based ES are presented and analysed.

First, the verification methodology is introduced. Results of the PAMA application on the FPGA-based ES are given along with the corresponding results on a laptop computer. Then, graphics applications on the FPGA-base ES are analysed and summarised in order of system layers from the hardware system up to applications.

8.1 Verification Methodology

Since the solution to graphics applications on the FPGA-based ES presented in this research is a hybrid one including software and hardware, it is necessary to analyse applications of surface modelling and editing with the PAMA in the FPGA-based ES from a system perspective. Each layer in this system from the hardware system to the applications makes its contribution and has its effect on the results. Thus, the analysis on results must take all of them into consideration.

In the next section, two groups of results are presented and compared. One group are the results of surface modelling and editing with the PAMA on the FPGA-based ES. The other group are the results of surface modelling and editing with the PAMA on the general-purpose computer – a laptop computer. The latter are generated on a platform where the hardware system, operating system, and OpenGL implementation have been verified and wide-accepted before this project and will be detailed in Section 8.3. The PAMA is mathematically proved in Appendix. The comparison between the above two groups of results can show the effect of surface modelling and editing with the PAMA on the FPGA-based ES. Furthermore, the FPGA-based ES and its parallel processing are verified when the PAMA is applied to the FPGA-based ES platform.

8.2 Results of PAMA Applications in the FPGA-based ES

With the Mesa-OpenGL implementation, surface modelling and editing with PAMA have been done in parallel and verified on the FPGA-based ES.

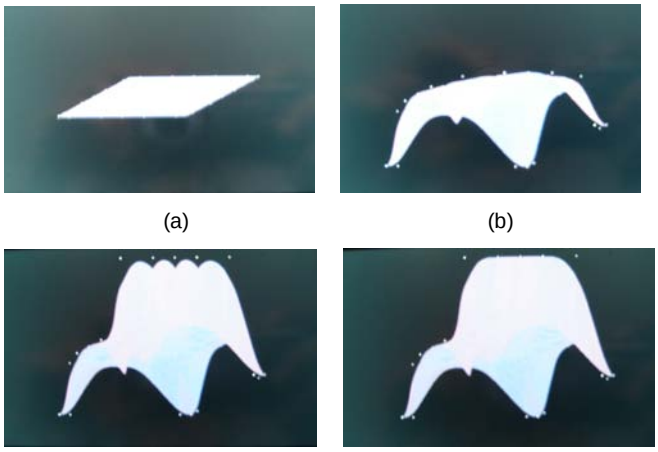
To make comparisons easy and equitable, the following results on the FPGA-based ES are presented along with the corresponding results on the laptop computer. Two groups of images are taken with the same resolution, 800 X 480 pixels, and the same black background when programs of PAMA are executed. The resolution for the result images of FPGA-based ES is the one of the LCD (shown on the right in Figure 3.4 and in Figure 8.1) while the resolution for the result images in the laptop computer is that of the window for the graphics applications running on the laptop.

Since the HAL and operating system of FPGA-base ES do not include the facility of print screen that operating systems in general-purpose computers usually provide, the images displayed on the LCD on the FPGA-based ES cannot be printed out with the print screen facility when programs of PAMA run in the ES. The images for results of the laptop computer are obtained with the print screen facility of Microsoft Windows. In addition, the figures for results on the FPGA-based ES are pictures taken of the LCD with a digital camera of Sony Cyber-shot DSC-P73 when programs of surface modelling and editing are executed in the FPGA-based ES. Figure 8.1 is a whole picture of table taken of the LCD. Other figures of the FPGA-base ES in the rest of this chapter are main parts cut from the original pictures of the LCD in order to facilitate the comparison. Therefore, this technique distinction between these two groups of figures exists.

Figure 8.1 A Picture of Table Taken of the LCD when Programs of Surface Modelling and Editing with PAMA Run in the FPGA-based ES.

Figures 8.2(a)-(d) show the changing process of the table and chair on the FPGA-based

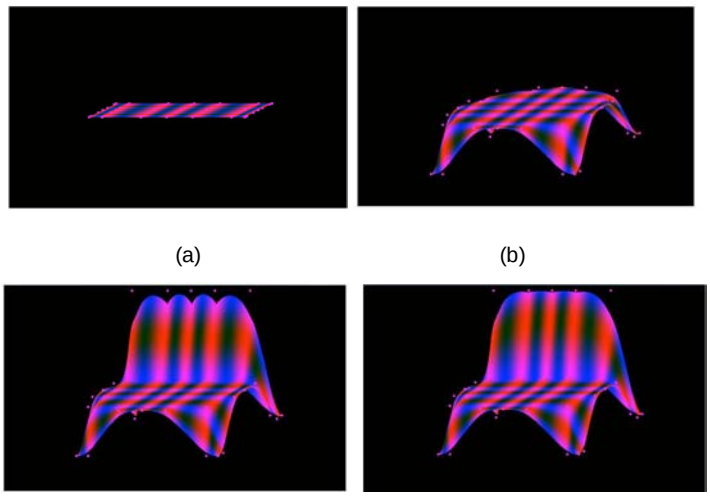
ES. These figures are main parts of pictures taken of the LCD. Figures 8.3(a)-(d) show the corresponding results of changing process of the table and chair that are done on the laptop computer, as shown in Figure 7.11. They are printed out with the print screen facility of Microsoft Windows.



(a) (b)

(c) (d)

Figure 8.2 A Series of Changing from a Flat Board to a Table and Chair with PAMA on the FPGA-based ES. (a) The Flat Board; (b) The Table; (c) The Semi-finished Chair; (d) The Completed Chair.



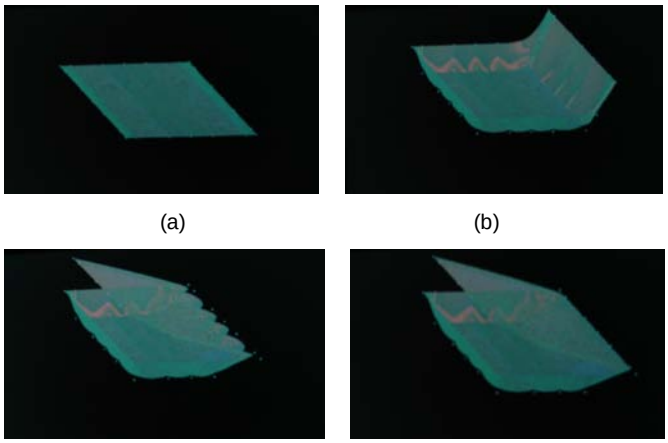
(a) (b)

(c) (d)

Figure 8.3 A Series of Changing from a Flat Board to a Table and Chair with PAMA on the Laptop Computer. (a) The Flat Board; (b) The Table; (c) The Semi-finished Chair; (d) The Completed Chair.

Figures 8.4(a)-(d) show the changing process of a clamshell box on the FPGA-based ES. These figures are main parts of pictures taken by the digital camera. Figures 8.5(a)-(d) show the corresponding results on the laptop computer, which are also shown in Figure

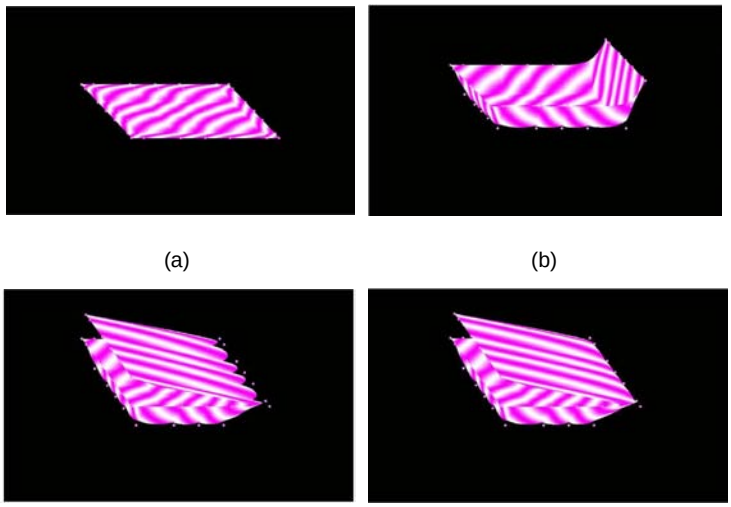
7.10. The latter are printed out with the print screen facility of Microsoft Windows.



(a) (b)

(c) (d)

Figure 8.4 A Series of Changing from a Flat Board to a Clamshell Box with PAMA on the FPGA-based ES. (a) The Flat Board; (b) The Semi-finished Box with a Half Lid; (c) The Semi-finished Box with a Full Lid; (d) The Completed Clamshell Box.

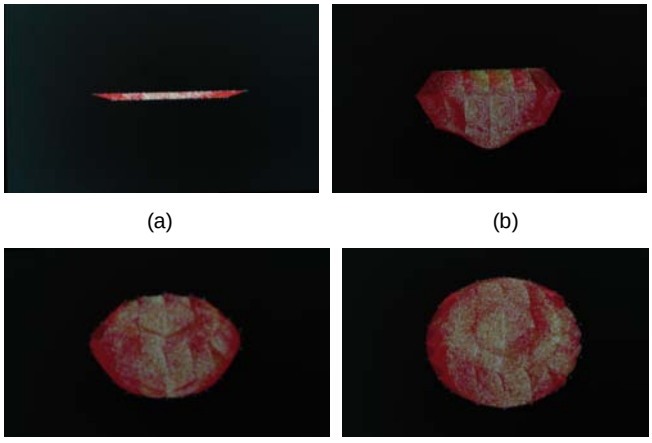


(a) (b)

(c) (d)

Figure 8.5 A Series of Changing from a Flat Board to a Clamshell Box with PAMA on the Laptop Computer. (a) The Flat Board; (b) The Semi-finished Box with a Half Lid; (c) The Semi-finished Box with a Full Lid; (d) The Completed Clamshell Box.

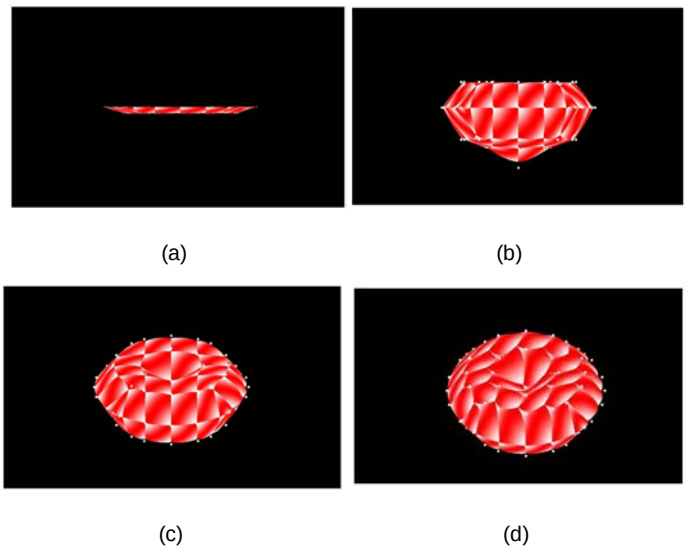
Figures 8.6(a)-(d) show the changing process of a flower bud on the FPGA-based ES. These figures are main parts of pictures taken with the digital camera. Figures 8.7(a)-(d) show the corresponding results on the laptop computer, which are also shown in Figure 7.12. The latter are printed out with the print screen facility of Microsoft Windows.



(a) (b)

(c) (d)

Figure 8.6 A Series of Changing from a Flat Box to a Flower Bud with PAMA on the FPGA-based ES. (a) The Flat Box; (b) The Semi-finished Flower Bud; (c) The Semi-finished Flower Bud Viewed in a Different Angle; (d) The Completed Flower Bud.



(a) (b)

(c) (d)

Figure 8.7 A Series of Changing from a Flat Box to a Flower Bud with PAMA on the Laptop Computer. (a) The Flat Box; (b) The Semi-finished Flower Bud; (c) The Semi-finished Flower Bud Viewed in a Different Angle; (d) The Completed Flower Bud.

8.3 Discussions of Graphics Applications on FPGA-based ES

As graphics applications, the programs of surface modelling and editing with the PAMA are used to verify the entire hybrid ES – FPGA-based ES. As shown in Figure 1.1, this hybrid ES, from the top down to the bottom, includes the algorithm for surface modelling and editing (PAMA), Mesa-OpenGL implementation, combination of Mesa-OpenGL to HAL, and FPGA-based embedded hardware system. Figures 8.1, 8.2, 8.4, and 8.6 are the test results on this platform.

In the corresponding environment of general-purpose computer, the hardware system is

the SAMAUNG's R480 laptop computer. The operating system is Microsoft Windows 7. The OpenGL implementation is the one for Microsoft Windows. All of them are verified and accepted by many researches in academic and industry societies before this project. Figures 8.3, 8.5, and 8.7 are the results generated on this platform.

Compared with the images printed out on the laptop computer, as shown in Figures 8.3, 8.5 and 8.7, Figures 8.2, 8.4 and 8.6 (of FPGA-based ES) seem to have poorer performance. However, a comparison of technical specifications between the two environments in Section 8.3.1 will indicate that the FPGA-based ES achieves the graphics application goal by balancing the system performance with the computation, storage, and power costs.

The user interaction for surface editing has been carried out with four on-board buttons (as shown in Figure 3.8). By pressing buttons, the surface shape can be changed.

The following discussions of graphics applications on the FPGA-based ES are summarised in order of system layers from the hardware system up to applications of the PAMA.

8.3.1 Distinction between Two Hardware Systems

Table 8.1 presents a comparison of technical specifications between the two environments. It can show the very limited resources, memory and processor speed that the FPGA-based ES has, compared with the laptop computer.

Table 8.1 Comparisons between Environments of Laptop Computer and FPGA-based ES

	Laptop Computer	***FPGA-based ES***
Microprocessor	Intel Core i5 CPU M 460	Altera Nios II
Microprocessor Speed	2.53 GHz	100 MHz
Main Memory Space	3.36 GB (4 GB Available)	2 X 64 MB
Display Adapter	ATI Mobility Radeon HD 5470 GPU	None
Display Memory	2 GB	None
Maximum Resolution (pixels)	2560 X 1600	800 X 480

In Table 8.1, the processor of FPGA-based ES is the Altrea Nios II at the speed of 100 MHz whereas the microprocessor of the laptop computer is the Intel Core i5 CPU M 460 at the speed of 2.53 GHz. The main memory space in the FPGA-based ES is 2 X 64 MBytes while that in the laptop computer is 3.36 GBytes. There is a display adapter of ATI Mobility Radeon HD 5470 GPU in the laptop computer while there is not in the FPGA-based ES.

There is a memory of 2 GB specified for display in the laptop computer, but there is not in the FPGA-based ES. The maximum resolution of screen on the laptop computer is 2560 X 1600 pixels while that of the LCD in the FPGA-based ES is 800 X 480 pixels.

8.3.2 Difference between Two OpenGL Implementations

In two environments of the general-purpose computer and FPGA-based ES, except for the distinctions between their hardware systems and APIs, there is another difference between them. It is the difference between their OpenGL implementations.

In the general-purpose computer environment, the OpenGL implementation is the OpenGL for Microsoft Windows Operating System, which has been widely accepted in Microsoft Windows and popularly applied to the computer graphics.

In the FPGA-based ES, the Mesa-OpenGL implementation for this platform is first applied after its development. Thus, the applications of surface modelling and editing with the PAMA to the general-purpose computer environment are references for the verification of the entire FPGA-based ES for the surface modelling and editing with the PAMA. From the perspective of system architecture, the former provide references for not only the PAMA but also the Mesa-OpenGL implementation on the FPGA-based ES platform.

For the Mesa-OpenGL implementation, there are three issues that need to be analysed in detail. They are the texture mapping, fixed point system and graphics processing bottleneck.

8.3.2.1 Texture Mapping

Compared with Figures 8.3, 8.5 and 8.7, in Figures 8.2, 8.4 and 8.6, the texture mapping has not provided as good performance as on the laptop computer yet. One reason for this is that the LCD resolution is lower than needed. Since the surface is divided into many tiny triangles, each scan line in a triangle often includes just one or two pixels. It results in the texture pattern being linearly degraded, and many details cannot be mapped in this tiny range. Another reason is that the transition between the tiny triangles is not so detailed for a high visual performance. A more effective scheme is needed for antialising.

8.3.2.2 Fixed Point System

The images shown in Figures 8.2, 8.4 and 8.6 give the similar geometric sense and visual performance as in Figures 8.3, 8.5 and 8.7. It indicates that the accuracy of the fixed point system can satisfy the computations of the Bézier-spline surface fitting algorithm and PAMA. In fact, the floating point issue in this project is important and common for FPGA implementations. Many FPGA systems are less successful in processing applications based on floating point arithmetic, especially double-precision (El-Ghazawi et al 2008). The disadvantage of floating point is its heavy usage of FPGA resources. According to the

study of El-Ghazawi et al (El-Ghazawi et al 2008), integer arithmetic applications can achieve high performance in fine-grained parallelism under area constraints.

8.3.2.3 Graphics Processing Bottleneck

The graphics processing speed of FPGA-based ES is slower than that of the laptop, in which all the editing operations can be real time. To modify the object shape on the FPGA-based ES, it takes around nine seconds for the table and chair and half a minute for the flower.

In the study of Kuehne et al 2005, they argue that no matter how well a program is written, one section of the program will always execute more slowly than any of the rest of the program. The slowest part of the program is called the bottleneck. They also give three possibilities of bottleneck in an OpenGL implementation, being filling limited, vertex limited, and application limited.

For this project, the bottleneck is vertex limited and partly application limited. Since the size of full-screen window on the LCD is only 800 X 480 pixels, it is able to flush and re-flush all the pixels on the screen in real time continuously for still 3D rendering. Thus, the bottleneck is not filling limited. If the application is only to display still surfaces rather than the surface editing, which requires a lot of vertex transformation, the 3D rendering can be real time. Thus, the bottleneck results partly from applications.

The evaluation of texture mapping and vertex transformation of the triangle strips costs most of the execution time. The object deformation takes a longer time than real time. It is about nine seconds, less or more dependent on the complexity of the objects. The triangle strips are needed when the Bézier-spline surface is drawn. For the example of the table and chair in Figure 8.2, the triangle strips consist of 42 triangles for each strip, 21 strips for each Bézier patch, and 49 patches and 43218 triangles in total. As regards the flower example in Figure 8.6, there are 91 patches and 80262 triangles in total. The object surfaces are broken down into such a tiny extent that some scanning lines are composed of just one or two pixels when the triangles are written into the frame buffer.

The evaluation of vertex transformation is limited by the small memory of 2 X 64 MBytes of the FPGA board of Altera ESs Development Kit, Cyclone III Edition.

8.3.3 Storage and Computing Costs of PAMA

Discussions of the previous chapters have shown that the computer graphics often requires large storage and computing costs, especially for 3D rendering. In this project, the hybrid solution with the FPGA-base ES has a restricted storage space, and the shape editing with user interactions in real time is heavily influenced by the computing cost.

Compared to subdivision and deformation methods discussed in Section 2.5, the PAMA

can eliminate the large storage requirement and computing cost of intermediate processes because the interpolated points are generated mainly from the original control points without intermediate processing.

Compared specifically with the subdivision work by Kazakov (Kazakov 2007) and Bolz and Schroder (Bolz and Schroder 2002), the PAMA has better performances in terms of input data, computing cost, and vertex buffer storage, which are detailed as follows:

- With the PAMA, the input data include the original control points of the control mesh while subdivisions include a list of vertices making up the control mesh and a variable-size list of vertices from its 1-ring neighbourhood.
- The PAMA requires 52 multiplications, 18 divisions, and 52 additions per control patch. Kazakov's work (Kazakov 2007) requires 123 multiplications and 248+2k additions per control mesh tessellated to a two-level subdivision, where k is the valence of the only extraordinary vertex. Bolz and Schroder's work (Bolz and Schroder 2002) requires 200+75k multiplications and 175+75k additions. For a simple recursive application of subdivision rules, given by Kazakov's work (Kazakov 2007), it requires 40 multiplications and 89+4k additions per face for tessellation to a two-level subdivision.
- The PAMA requires four times of buffer storage for the interpolated points as that for the original control points. One- and two-level subdivision meshes require four and sixteen times of vertex buffer storage as that of a control mesh.

8.4 Chapter Summary

In this chapter, the verification methodology is introduced first. Then, two groups of results of surface modelling and editing with the PAMA on two environments, which are the FPGA-based ES and the laptop computer, are presented together. The discussions of graphics applications on the FPGA-base ES are detailed in order of system layers from the hardware system up to the applications.

In general, when considering the graphics rendering, speed-up is the first requirement, which is achieved at a high price of the memory space and processor ability that are designated specifically for graphics processing, such as, GPU in the laptop. From a practical standpoint, there are many situations that may limit the speed, such as the ES applications. FPGAs are highly desirable from an economic perspective given the cost difference between FPGAs and microprocessors, and they are particularly suitable for special-purpose systems.

Although the FPGA looks like a low-cost and low-quality solution to the graphics rendering with OpenGL compared with the general-purpose computer with the high-end

GPU, the FPGA does give an alternative option for graphics applications with the low computation, storage and power costs. For academic researches, a new exploration in a new direction looks imperfect at the beginning, just like what has been done in this project. This research, however, presents a new hybrid solution to graphics applications by using the FPGA platform to draw and edit 3D objects with user interactions.

Chapter 9 Future Work

Since this research involves the FPGA-based ES for graphics applications, Mesa-OpenGL implementation, parallelism processing, and surface modelling and editing with the PAMA, the future work is discussed in five aspects. These five aspects are the future work for the methodology of hybrid design of application-specific ESs with software and hardware components, FPGA-based ESs, OpenGL implementations, parallelism processing and the PAMA, respectively.

9.1 Future Work for Methodology of Hybrid Design of Application-specific ESs with Software and Hardware Components

The hybrid way of solving specified applications with FPGA-based ESs presented in this thesis is different from those of the separate software and hardware solutions. The future work for this hybrid scheme is as follows.

- This hybrid methodology needs to be advanced and completed so that application-specific ESs can be designed and implemented with software and hardware components via a practical process for not only academic researches but also electronic commodity developments.
- System-level verification methods need to be developed and enhanced for the hybrid design and implementation of application-specific ESs in order to guarantee that the hybrid design and implementation of an ES meet requirements of its target applications.
- A new mathematic system for this hybrid way of constructing a system for specified applications needs to be established. The new mathematic system needs to combine or include the digital and analogue domains so that the hybrid methodology can have a mathematic foundation to support it.

9.2 Future Work for FPGA-based ESs

Since the hybrid scheme needs to consider both software and hardware during design and implementation of an FPGA-based ES, the conventional environments for design of separate software and hardware cannot satisfy the requirements of an entire system design that needs to consider both software and hardware during the process of design and implementation. To promote this hybrid scheme, the future work needs to be done as follows.

- Maintaining and advancing of design environments for FPGA-base ESs with the hybrid way need to accumulate more libraries and tools so that the facilities in these environments can be as available and convenient as those of software programming.
- Developing and advancing of hybrid design environments also need raw recruits to observe that more opportunities will benefit ESs than software programming if the potential of ESs can be explored by more researchers and engineers. The lower popularity and higher speciality of recent ESs can lead to openings for good ES designers and developers. These can help the ES community grow quickly.

9.3 Future Work for OpenGL Implementations

The standards of OpenGL and OpenGL ES can be implemented in varied hardware platforms. When they are implemented to realise 3D rendering with user interactions on a new hardware platform, some tasks must be undertaken. These tasks are listed as follows.

- User interactions need the support of a simple keyboard, which can provide adequate keys and allow users to manipulate graphics objects flexibly. A touch screen is an alternative tool of user interactions for portable gadgets, but its device drivers for 3D rendering must be developed before it functions as an effective input and output device for user interactions of 3D graphics applications.
- High-quality 3D images need the support of an effective antialiasing technique. This technique needs to produce smooth transition between tiny patches that are usually used in complex objects' rendering, such as tiny triangles in Bézier-spline surface fitting.
- High-quality 3D images also need a high-resolution LCD screen (such as 1024 X 512 pixels or more). It needs to display details of complex objects and allow proper texture mapping.
- A hardware solution to an advanced algorithm is required to speed up graphics

processing if this algorithm is repeated multiple times during 3D rendering. For instance, the part of triangle processing and writing in the frame buffer for Bézier-spline surfaces needs to be transformed to a hardware solution with the C2H tool of the Altera ESDK. This part needs to be integrated with the FPGA-based embedded hardware system to accelerate the 3D rendering because it is at the low layer of the software system, close to the HAL, and repeated many times during the execution of the PAMA.

9.4 Future Work for Parallel Processing

The pipelines and co-processors need to receive an emphasis in the parallel computing community because they can effectively speed up computing and processing and help to enhance the entire performance of a system. They need to gradually become parallelism mainstreams of the modern parallel theory in order to be widely used by designers during the process of system design and implementation.

9.5 Future Work for PAMA

As the increasing density of FPGA chips, more complex and higher-quality FPGA devices than the Altera Cyclone III 3C120F780 adopted by this research are available. They can provide larger storage space and faster processor units. In this situation, the PAMA needs to be enhanced in several aspects as follows.

- More complex objects (with a grid of 1M vertices) need to be modelled and edited with the PAMA. An effective technique of manipulating the shape of complex objects via user interactions needs to be found because a complex object may have more control points than a simple one. Each of control points has its position and four shape parameters to be manipulated. When the number of control points increases, the number of parameters to be changed increase five times as many as the number of control points. This issue must be handled in an effective way.
- The constructions of four inside interpolated points of a Bézier-spline patch with the PAMA can be changed according to conditions of G^0, G^1, or G^2. These changes can produce different geometric effects on the common boundary curves between patches. Furthermore, they can meet the needs for different design purposes.
- An effective scheme for inputting or outputting a large amount of processed data of vertices of a complex object from or to a file needs to be found. This scheme is helpful for the situation where the work creation cannot be completed once by a designer and needs to be refined several times.

Chapter 10 Conclusions

This research has demonstrated an integrated hybrid method for graphics and video processing by using both hardware and software. Surface modelling and editing with the PAMA have been carried out and applied to the FPGA-based ES, which has been built in the hybrid way of hardware and software. For the hybrid method, several facilities have been established. They are included in four main parts of this research as follows:

- The FPGA-based ES offers an alternative solution to graphics and video processing, which is different from a general-purpose computer platform with GPUs.
- The Mesa-OpenGL implementation for the FPGA-based ES completes not only specifications of the general OpenGL ES but also additional functions needed for surface modelling and editing.
- The parallel processing of pipelines and co-processors is used to accelerate the computing and processing of graphics and video;
- The novel algorithm for surface modelling and editing, PAMA, has been created and verified.

Conclusions for this research are summarised in five sections. The first section presents conclusions for the methodology for hybrid design and implementation of ESs. The other four sections are used for the conclusions of the above four main parts, respectively.

10.1 Conclusions of Methodology for Hybrid Design and Implementation of ESs

Along with the accomplishment and verification of the goal of this project, the hybrid method of solving the graphics applications has been validated. As regards this method, several conclusions can be summarised as follows.

- The new design platform and development flow for FPGA-based ES design in the hybrid way presented in this thesis can provide facilities for designers and developers of application-specific ESs to implement the system construction for

designated applications in the hybrid way even though the prevailing design environments have not been developed completely for this method. A set of techniques for the hybrid system design and construction, which must take both hardware and software into consideration of the design flow and system construction, are created. These techniques include the pools of building blocks for hardware and software, the hierarchy of design process, and the processing order of the system construction and verification.

- The novel hybrid approach offers new solutions to computer graphics applications. This hybrid methodology can provide an effective and efficient strategy to obtain an adequate performance of an entire ES with low costs of computing, storage and power. It can allow designers to treat the design and implementation of ESs from a system perspective and construct them in a systematic way, rather than from the conventional partial perspective. In this hybrid way, designers can find a solution with a proper performance and low costs for target applications.
- This hybrid methodology can make good use of the strengths of both hardware and software. The parallel feature of hardware can be used during the construction of the hardware system and the complexity and logicality of software can be exploited during software programming.
- The system verification method presented in the thesis can provide a verification method for the FPGA-based ES design for specified applications. It can use the completed applications to test whether or not an ES implementation meets requirements of the targeted applications.

10.2 Conclusions of FPGA-based ES

The FPGA-based ES for graphics applications presented in this thesis offers a new solution for computer graphics applications. Compared with GPUs, this solution has lower costs of computing, storage and power, and its lower performance can be enhanced by replacing relative devices and refining related algorithms. The FPGA-based ES platform also allows one or several developers to accomplish an entire ES design for specific applications, which usually needs a team including hardware engineers, device driver developers and application developers to be accomplished if the conventional ES development method is taken.

10.3 Conclusions of Mesa-OpenGL Implementation

The Mesa-OpenGL implementation provided in the thesis is established for the FPGA-base ES. It increases the set of existing implementations of OpenGL ES. Its

addition of Bézier-spline curve and surface algorithms extends the standard specifications of OpenGL ES. Its fixed point system satisfies the computations of Bézier-spline curve and surface algorithms and the PAMA. Its small footprint meets the needs of the computing and storage limitations of FPGA-based ESs.

10.4 Conclusions of Parallel Processing

Rather than the application programmer's view, the hardware builder's view taken in this research provides a proper standpoint from which the parallelism possibility in the hybrid system construction with software and hardware can be identified accurately and used effectively. This develops the fine-grained task parallelism in the traditional parallelism that emphasised the data parallelism of SIMD and MIMD.

Compared with the partitioned parallelism that uses the spatial division of data processing, the pipelined parallelism employed in this research can fit more into the FPGA-based ES for the graphics applications. The latter makes the best use of the temporal division of pipeline and speeds up graphics processing.

The video pipeline of the FPGA-based ES also functions as a co-processor, which can automatically work without the Nios II processor's intervention.

10.5 Conclusions of PAMA

The conclusions about the PAMA presented in this thesis can be summarised as follows.

- The PAMA has a small footprint that meets the requirements of storage, computing and power costs of the FPGA-based ES.
- It allows users to designate control points of a surface that can be open or closed for their design purposes. This is necessary for the original creation of work.
- It provides tools that users can use to change the shapes of object surfaces freely by changing independently the position, and four shape parameters of each control point. These tools can help users accumulate the design experience during surface modelling and editing.
- A global composite surface joined with the PAMA is G^0 (and C^0). This results in the freedom for shape variation by user interactions. It also brings varied shapes of not only G^1 but also G^0. The G^0 can offer a sharp fold on a common boundary curve between two patches whereas the G^1 can only provide a smooth transition crossing two patches.

Appendix Continuities of PAMA

In this appendix, the continuities of the PAMA will be explored.

Since the PAMA is an algorithm that is used to generate a composite surface by joining bi-cubic Bézier-spline surface patches together with geometric continuities on common boundary curves, two relevant issues must be investigated before the exploration of PAMA's geometric continuities. One is that parametric continuities and geometric continuities must be ascertained further. The other is that some geometric properties of Bézier-spline curves and surfaces, which will be used later in the PAMA's discussion, must be identified as well.

In this appendix, Sections A.1 discusses parametric continuities and geometric continuities. From Section A.2 to Section A.6, the geometric properties of Bézier-spline curves and surfaces will be studied. From Section A.7, the exploration into the PAMA will proceed. To articulate it, in Section A.7, the principle for the construction of control vertices on common boundary curves of Bézier-spline surfaces with the PAMA is presented after the introduction of Beta-spline curves. In Section A.8, the twists and constructions of corner points of patches with the PAMA are discussed. In Section A.9, the constructions of inside points with the PAMA are inspected. Finally, Section A.10 is used to summarise PAMA's continuities.

In the rest of discussion, Bézier-spline curves and surfaces will be simplified as Bézier curves and surfaces.

A.1 Parametric Continuities and Geometric Continuities

In the study of Hoschek and Lasser (Hoschek and Lasser 1993), the deductive reasoning of the parametric continuities and geometric continuities is provided in a mathematical way, which is palpable for a thorough investigation of these two types of continuities.

Given two parametric curves, *T(u),* $u \in [u_0, u_1]$, and *S(w)*, $w \in [w_0, w_1]$, in R^d (*d*-dimensional Riemannian space), which meet at a common point $P = T(u_0) = S(w_1)$,

we say these two curves meet with the *r*-order geometric continuity (G^r), if there exists an algebraic curve which meets both curves *T(u)* and *S(w)* at *P* with contact of order *r* in the sense that the first *r* terms in the Taylor series expansions about the point *P* of the two given curves and the algebraic curve all agree at *P* (Hoschek and Lasser 1993).

To compare the Taylor series of *T(u)* and *S(w)* with each other, if they are developed with respect to the same parameter, we can find the derivatives with respect to the same parameter and derive the conditions of *r*-order parametric continuities(C^r), written as

$$\frac{d^i}{du^i}T(u)|_{u_0^+} = \frac{d^i}{dw^i}S(w)|_{w_1^-}, \text{ where } u_0 = w_1, \text{ and } i = 0, \ldots, r. \quad \text{(A.1)}$$

If they join with the parametric continuities with respect to the natural arc length parametrisation and with the assistance of the chain rule, *T(u)* and *S(w)* meet the conditions of geometric continuities. For the zero-, first-, and second-order geometric continuities (G^0, G^1 and G^2), the conditions are written as the following equations, respectively,

$$T(u_0^+) = S(w(u_0^-)) = S(w_1^-), \quad \text{(A.2a)}$$

$$\frac{d}{du}T(u)|_{u_0^+} = \left(\frac{d}{du}w(u)|_{u_0^-}\right)\left(\frac{d}{dw}S(w)|_{w_1^-}\right) = \beta_1\left(\frac{d}{dw}S(w)|_{w_1^-}\right), \quad \text{(A.2b)}$$

and

$$\frac{d^2}{du^2}T(u)|_{u_0^+} = \left(\frac{d}{du}w(u)|_{u_0^-}\right)^2\left(\frac{d^2}{dw^2}S(w)|_{w_1^-}\right) + \left(\frac{d^2}{du^2}w(u)|_{u_0^-}\right)\left(\frac{d}{dw}S(w)|_{w_1^-}\right)$$

$$= \beta_1^2\left(\frac{d^2}{dw^2}S(w)|_{w_1^-}\right) + \beta_2\left(\frac{d}{dw}S(w)|_{w_1^-}\right), \quad \text{(A.2c)}$$

where $w_1^- = w(u_0^-)$, $\beta_1 = \frac{d}{du}w(u)|_{u_0^-}$, and $\beta_2 = \frac{d^2}{du^2}w(u)|_{u_0^-}$.

Equation A.2a is the condition of zero-order geometric continuity (G^0) of adjoining parametric curves, Equation A.2b is of first-order (G^1), and Equation A.2c is of second-order (G^2), respectively.

Through the above process of deductive reasoning, since the Taylor series expansions of *T(u)* and *S(w)* have remainders, we can say that no matter they meet the *r*-order parametric continuities or the *r*-order geometric continuities, the algebraic curve agrees with *T(u)* and *S(w)* at *P* in an approximate means.

For surfaces, the study of Hoschek and Lasser (Hoschek and Lasser 1993) states that

for C^1 continuity, the first partial derivatives agree along and across the common boundary curve between two neighbouring Bézier patches. That study also provides the lucid mathematical expressions of conditions of geometric continuities of parametric surfaces. If two parametric surfaces, *T(u,v)* and *S(w,t)*, meet at a point, $P = T(\hat{u},\hat{v}) = S(\hat{w},\hat{t})$, after the reparametrisation of $u \mapsto u(w,t)$ and $v \mapsto v(w,t)$, by using the theory of manifolds, the conditions for the zero-, first- and second-order geometric continuities are written as

$$T(\hat{u},\hat{v}) = T(u(\hat{w},\hat{t}), v(\hat{w},\hat{t})) = S(\hat{w},\hat{t}), \tag{A.3a}$$

$$T^{'} = \Psi_{11} S^{'}, \tag{A.3b}$$

and $$T^{''} = \Psi_{22} S^{''} + \Psi_{12} S^{'}, \tag{A.3c}$$

where $\hat{u} = u(\hat{w},\hat{t})$, $\hat{v} = v(\hat{w},\hat{t})$, $\Psi_{11} = \begin{pmatrix} \frac{\partial u}{\partial w}|_{(\hat{w},\hat{t})} & \frac{\partial v}{\partial w}|_{(\hat{w},\hat{t})} \\ \frac{\partial u}{\partial t}|_{(\hat{w},\hat{t})} & \frac{\partial v}{\partial t}|_{(\hat{w},\hat{t})} \end{pmatrix}$, $\Psi_{22} = \begin{pmatrix} \frac{\partial^2 u}{\partial w^2}|_{(\hat{w},\hat{t})} & \frac{\partial^2 v}{\partial w^2}|_{(\hat{w},\hat{t})} \\ \frac{\partial^2 u}{\partial w \partial t}|_{(\hat{w},\hat{t})} & \frac{\partial^2 v}{\partial w \partial t}|_{(\hat{w},\hat{t})} \\ \frac{\partial^2 u}{\partial t^2}|_{(\hat{w},\hat{t})} & \frac{\partial^2 v}{\partial t^2}|_{(\hat{w},\hat{t})} \end{pmatrix}$,

$$\Psi_{12} = \begin{pmatrix} (\frac{\partial u}{\partial w}|_{(\hat{w},\hat{t})})^2 & 2(\frac{\partial u}{\partial w}|_{(\hat{w},\hat{t})})(\frac{\partial v}{\partial w}|_{(\hat{w},\hat{t})}) & (\frac{\partial v}{\partial w}|_{(\hat{w},\hat{t})})^2 \\ (\frac{\partial u}{\partial w}|_{(\hat{w},\hat{t})})(\frac{\partial u}{\partial t}|_{(\hat{w},\hat{t})}) & (\frac{\partial u}{\partial w}|_{(\hat{w},\hat{t})})(\frac{\partial v}{\partial t}|_{(\hat{w},\hat{t})}) + (\frac{\partial u}{\partial t}|_{(\hat{w},\hat{t})})(\frac{\partial v}{\partial w}|_{(\hat{w},\hat{t})}) & (\frac{\partial v}{\partial w}|_{(\hat{w},\hat{t})})(\frac{\partial v}{\partial t}|_{(\hat{w},\hat{t})}) \\ (\frac{\partial u}{\partial t}|_{(\hat{w},\hat{t})})^2 & 2(\frac{\partial u}{\partial t}|_{(\hat{w},\hat{t})})(\frac{\partial v}{\partial t}|_{(\hat{w},\hat{t})}) & (\frac{\partial v}{\partial t}|_{(\hat{w},\hat{t})})^2 \end{pmatrix},$$

$$T^{'} = \begin{bmatrix} \frac{\partial}{\partial u} T \\ \frac{\partial}{\partial v} T \end{bmatrix}, \quad S^{'} = \begin{bmatrix} \frac{\partial}{\partial w} S \\ \frac{\partial}{\partial t} S \end{bmatrix}, \quad T^{''} = \begin{bmatrix} \frac{\partial^2}{\partial u^2} T \\ \frac{\partial^2}{\partial u \partial v} T \\ \frac{\partial^2}{\partial v^2} T \end{bmatrix}, \text{ and } \quad S^{''} = \begin{bmatrix} \frac{\partial^2}{\partial w^2} S \\ \frac{\partial^2}{\partial w \partial t} S \\ \frac{\partial^2}{\partial t^2} S \end{bmatrix}.$$

Equation A.3a is the condition of G^0 of adjoining parametric surfaces, Equation A.3b is of G^1, and Equation A.3c is of G^2, respectively. The first-order condition has a straightforward geometric meaning. That is, it is that two adjoining surface patches with G^1 have common tangent planes on their common boundary curve. We will discuss it in detail in Section A.4.

Since the Ψ_{22} and Ψ_{12} have a variety of first- and second-order partial derivatives and their varied compositions do not have definite geometric meanings, it is difficult to find a practical way to construct a composite surface that meets the condition of G^2.

A.2 Geometric Properties of Bézier Curves

In this section, we concentrate on the geometric properties of Bézier curves that we will apply to the exploration of PAMA's geometric continuities.

A Bézier curve is formed with a control polygon that is composed of a series of control points. A Bézier curve of degree *n* is written in a Bernstein polynomial form as

$$x(u) = \sum_{k=0}^{n} b_k B_k^n(u), \tag{A.4}$$

where $0 \le u \le 1$, and $B_k^n(u) = \binom{n}{k} u^k (1-u)^{n-k}$. Its control polygon is $[b_0, b_1, b_2, ..., b_n]$ and $b_0, b_1, ..., b_n \in E^3$ (three-dimensional Euclidean space). We can state some geometric properties of the Bézier curve.

Property 1. *The Bézier curve x(u) of Equation A.4 passes through two control points at two ends of the curve, which are two endpoints,* b_0 *and* b_n, *of its control polygon.*

The proof of Property 1 is straightforward by replacing the value of parameter *u* with *0* and *1* in Equation A.4 and having $x(0) = b_0$ and $x(1) = b_n$.

Property 2. *Both of two lines passing, respectively, through the first two control points,* b_0 *and* b_1, *and through the last two control points,* b_{n-1} *and* b_n, *are tangents of the Bézier curve x(u) of Equation A.4.*

Proof of Property 2. The derivative of the Bézier curve of Equation A.4 is written as

$$\frac{d}{du} x(u) = n \sum_{k=0}^{n-1} \Delta b_k B_k^{n-1}(u), \tag{A.5}$$

where $\Delta b_k = b_{k+1} - b_k$. Two special cases of Equation A.5 are obtained by *u = 0* and *u = 1* as

$$\frac{d}{du} x(0) = n(b_1 - b_0), \tag{A.6}$$

and

$$\frac{d}{du} x(1) = n(b_n - b_{n-1}). \tag{A.7}$$

Equation A.5 shows that the derivative of the Bézier curve is also a Bézier curve. But its coefficients are not points in E^3. They are differences of points of E^3, or vectors in R^3 (three-dimensional Riemannian space).

Equation A.6 states that b_1 and b_0 determine the tangent at *u = 0* while Equation A.7 indicates that b_n and b_{n-1} define the tangent at *u = 1*. Thus Property 2 is proved. □

Then, we attempt to find a tangent at an arbitrary point of the Bézier curve. The de Casteljau algorithm offers an effective scheme for the Bézier curve computation, which is a process including multiple repeated steps (Farin 1993, and Hoschek and Lasser 1993). Figure A.1 shows the process to yield a point on a cubic Bézier curve with the de Casteljau algorithm.

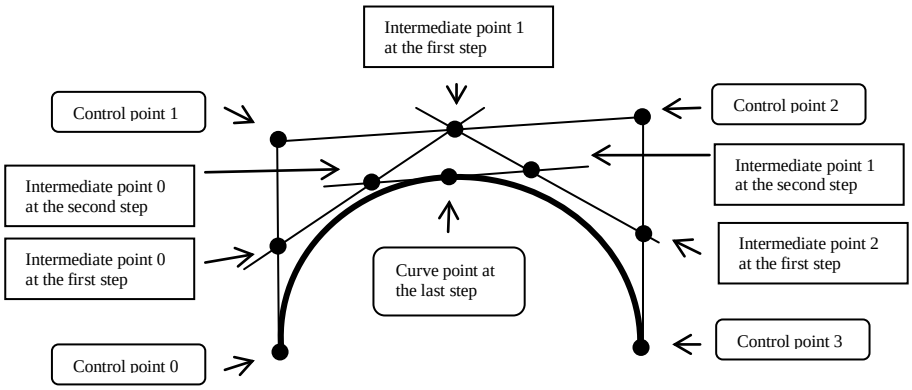


Figure A.1 Process of Generating a Point on the Cubic Bézier Curve with the de Casteljau Algorithm. Properties 2 and 3 can be observed. Property 2 is that Two Lines Passing Through Control Points 0 and 1, and Through Control Points 2 and 3, respectively, are Tangent to the Bézier Curve. Property 3 is that the Line Passing through the Intermediate Points 0 and 1 at the Second Step (also the Second Last Step) is Tangent to the Cubic Bézier Curve.

The de Casteljau algorithm can generate the tangent at any point of the Bézier curve. The number of steps of this algorithm is the number of control points of the Bézier curve minus one.

Property 3. *For a given value of parameter u, the de Casteljau algorithm yields the point on the Bézier curve of Equation A.4 at the last step and the tangent of the curve at that point with the difference of two intermediate points that are generated at the second last step.*

Proof of Property 3. The derivative of the Bézier curve of Equation A.4 has been given by Equation A.5.

With the de Casteljau algorithm, the intermediate de Casteljau points are obtained as

$$b_k^r(u) = (1-u)b_k^{r-1}(u) + ub_{k+1}^{r-1}(u), \tag{A.8}$$

where $\begin{cases} r = 1,2,...,n \\ k = 0,1,...,n-r \end{cases}$, and $b_k^0(u) = b_k$. $b_0^n(u)$ is the point with the parameter value *u* on

the Bézier curve. $b_0^n(u)$ is generated at the last step of the de Casteljau algorithm and has the formula as

$$b_0^n(u) = (1-u)b_0^{n-1}(u) + ub_1^{n-1}(u). \quad \text{(A.9)}$$

The intermediate de Casteljau points can be written in terms of Bernstein polynomials of degree *r*:

$$b_k^r(u) = \sum_{i=0}^{r} b_{i+k} B_i^r(u), \quad \text{(A.10)}$$

where $\begin{cases} r = 1,2,...,n \\ k = 0,1,...,n-r \end{cases}$.

Let us inspect two intermediate de Casteljau points at the second last step of the de Casteljau algorithm, where *r = n-1*,

$$b_0^{n-1}(u) = \sum_{i=0}^{n-1} b_i B_i^{n-1}(u), \quad \text{(A.11)}$$

and

$$b_1^{n-1}(u) = \sum_{i=0}^{n-1} b_{i+1} B_i^{n-1}(u). \quad \text{(A.12)}$$

Subtracting Equation A.12 and Equation A.11 and replacing *i* with *k*, we can obtain

$$b_1^{n-1}(u) - b_0^{n-1}(u) = \sum_{i=0}^{n-1} b_{i+1} B_i^{n-1}(u) - \sum_{i=0}^{n-1} b_i B_i^{n-1}(u) = \sum_{k=0}^{n-1} (b_{k+1} - b_k) B_k^{n-1}(u). \quad \text{(A.13)}$$

Compare Equation A.13 with Equation A.5. The difference between them is the multiple, *n*. After normalised, they are the same unit vector in R^3. Since Equation A.5 indicates the tangent of the Bézier curve of Equation A.4, so does Equation A.13. The point, $b_0^n(u) = x(u)$, is also a point of this tangent, which is observed in Equation A.9. Therefore, $b_0^{n-1}(u)$ and $b_1^{n-1}(u)$ determine the tangent passing through the point of $b_0^n(u)$ that can be any point on the Bézier curve of Equation A.4 as $0 \le u \le 1$.

Thus Property 3 is proved. □

The tangents of Properties 2 and 3 for the cubic Bézier curve are shown in Figure A.1.

A.3 Composite Bézier Surfaces

Following the discussion of the PAMA in Chapter 6, we choose one patch of a composite bi-cubic Bézier surface, which is formed with a control net with four corner vertices, *[W(3i, 3j),W(3i, 3(j+1)),W(3(i+1),3(j+1)),W(3(i+1), 3j)]*, as shown in Figure A.2. The patch consists of eight points on four boundaries, two on each boundary. There are four points inside the

patch. The number of overall control vertices of one Bézier patch is sixteen. We stitch these patches together with the PAMA to form a composite Bézier surface and will ascertain the geometric continuities of the composite Bézier surface later.

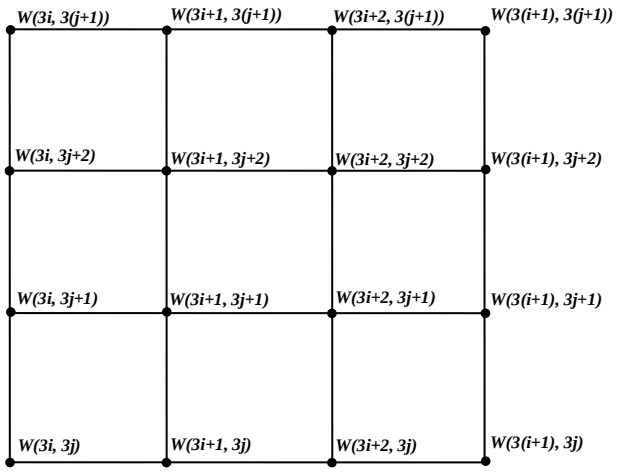


Figure A.2 Control Net of one Patch of the Composite Bi-cubic Bézier Surface, Formed with Four Corner Control Vertices, *[W(3i, 3j), W(3i, 3(j+1)), W(3(i+1), 3(j+1)), W(3(i+1), 3j)]*.

Let us explore a special case of a Bézier surface (or patch). The tense product form of the Bézier patch with degree (3, 3) is defined as

$$x_{i,j}(u,v) = \sum_{k=0}^{3}\sum_{l=0}^{3} W(3i+k, 3j+l) B_k^3(u) B_l^3(v), \tag{A.14}$$

where $0 \le u, v \le 1$, $B_k^3(u) = \binom{3}{k} u^k (1-u)^{3-k}$, and $B_l^3(v) = \binom{3}{l} v^l (1-v)^{3-l}$.

The patch $x_{ij}(u,v)$ is a map of the unit square $0 \le u, v \le 1$ in the u, v-plane, as shown in Figure A.3. This unit square is the domain of parameters u and v while the patch $x_{ij}(u,v)$ is its range.

To derive the geometric properties of this tense-product Bézier patch, we must analyse further the construction of Equation A.14. According to Farin 1993, a tense-product Bézier patch is the locus of a Bézier curve that is moving through space along another Bézier curve while changing its shape at the same time. Take the Equation A.14 as the example. The former Bézier curve is a generating curve, which is,

$$x_g(u) = \sum_{k=0}^{3} b_k B_k^3(u).$$

Let each b_k traverse another Bézier curve,

$$b_k(v) = \sum_{l=0}^{3} b_{k,l} B_l^3(v). \tag{A.15}$$

We derive the tense-product Bézier patch of Equation A.14 by combining the above two formulas and replacing with $b_{k,l} = W(3i+k, 3j+l)$.

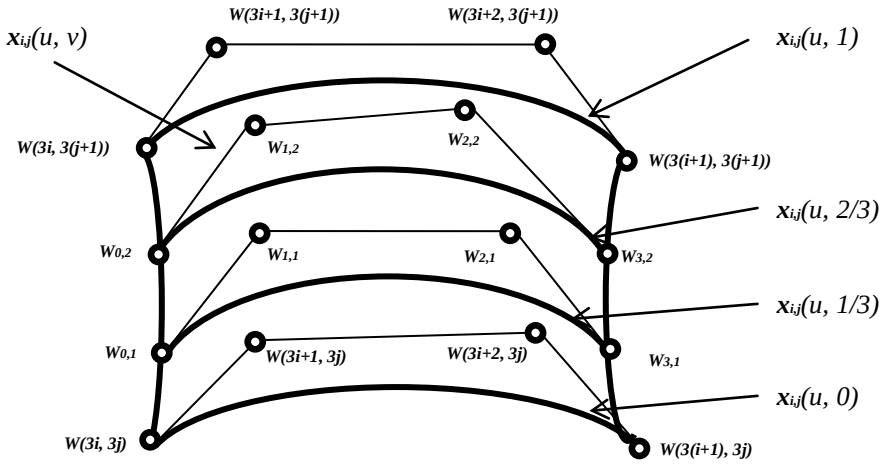


Figure A.3 A Patch, $x_{i,j}(u, v)$, of the Composite Bézier Surface and its *v*-Isoparametric Curves. *[W(3i, 3j), W(3i+1, 3j), W(3i+2, 3j), W(3(i+1), 3j)]* are the Control Polygon of the Isoparametric Curve, $x_{i,j}(u, 0)$. *[$W_{0,1}$, $W_{1,1}$, $W_{2,1}$, $W_{3,1}$]* are the Control Polygon of the Isoparametric Curve, $x_{i,j}(u, 1/3)$. *[$W_{0,2}$, $W_{1,2}$, $W_{2,2}$, $W_{3,2}$]* are the Control Polygon of the Isoparametric Curve, $x_{i,j}(u, 2/3)$. *[W(3i, 3(j+1)), W(3i+1, 3(j+1)), W(3i+2, 3(j+1)), W(3(i+1), 3(j+1))]* are the Control Polygon of the Isoparametric Curve, $x_{i,j}(u, 1)$.

By setting the parameter *u* or *v* as a constant in Equation A.14, we obtain a *u*- or *v*-isoparametric curve, respectively. As each of four coefficients of any isoparametric curve traversing a Bézier curve in the way of Equation A.15, we can obtain an arbitrary isoparametric curve of the patch of Equation A.14.

In general, we can state the following property.

Property 4. *Given a Bézier surface* of *degree (m, n) in* E^3,

$$x(u,v) = \sum_{k=0}^{m} \sum_{l=0}^{n} b_{k,l} B_k^m(u) B_l^n(v), \text{ where } 0 \le u, v \le 1, \tag{A.16}$$

its u- or *v-isoparametric curves are Bézier curves.*

Proof of Property 4. Set *u* as a constant, saying *c*, in Equation A.16. We obtain

$$x(c,v) = \sum_{k=0}^{m} \sum_{l=0}^{n} b_{k,l} B_k^m(c) B_l^n(v) = \sum_{l=0}^{n} [\sum_{k=0}^{m} b_{k,l} B_k^m(c)] B_l^n(v). \tag{A.17}$$

To make it obvious, we define new coefficients as the bracketed terms as follows

$$\widehat{b_l} = \sum_{k=0}^{m} b_{k,l} B_k^m(c). \tag{A.18}$$

Substituting the bracketed terms in Equation A.17 with Equation A.18, we obtain

$$x(c,v) == \sum_{l=0}^{n} \hat{b}_l B_l^n(v). \tag{A.19}$$

New coefficients can be seen as the control points of control polygon, $[\hat{b}_0, \hat{b}_1, ..., \hat{b}_n]$. Equation A.19 represents an arbitrary *u*-isoparametric curve of the Bézier surface *x(u,v)* and is a Bézier curve with degree *n*.

Analogously, by setting *v* as a constant, *c*, we can obtain a *v*-isoparametric curve of the Bézier surface *x(u,v)* that is a Bézier curve with degree *m* as the following equation

$$x(u,c) == \sum_{k=0}^{m} \breve{b}_k B_k^m(v), \tag{A.20}$$

where $\breve{b}_k = \sum_{l=0}^{n} b_{k,l} B_l^n(c)$. (A.21)

Therefore, Property 4 is true. □

Let us return to the previous bi-cubic Bézier patch case. Four *v*-isoparametric curves of the patch, which having the constant values of *v,* are chosen. With Property 4, we can set the parameter *v* to an arbitrary value that $v \in [0,1]$. Without loss of generality, these values are assigned as *0*, *1/3*, *2/3*, and *1.0*, respectively. The control polygons of these four isoparmetric curves are computed as follows.

For *v=0*, the control polygon of the isoparametric Bézier curve $x_{i,j}(u,0)$ is *[W(3i, 3j),W(3i+1, 3j),W(3i+2,3j),W(3(i+1), 3j)].* These four points are the control points of the boundary of the Bézier patch, $x_{i,j}(u,v)$.

For *v=1/3*, the control polygon of the isoparametric Bézier curve $x_{i,j}(u,1/3)$ consists of four control points that are intermediate points computed by replacing *v* with *1/3* in Equation A.14 and as the follows,

$$W_{k,1} = \sum_{l=0}^{3} W(3i+k, 3j+l) B_l^3\left(1/3\right)$$

$$= \frac{8}{27} W(3i+k, 3j) + \frac{4}{9} W(3i+k, 3j+1) + \frac{2}{9} W(3i+k, 3j+2) + \frac{1}{27} W(3i+k, 3(j+1)), \tag{A.22}$$

where *k=0*,*1*, *2*,and *3*.

For *v=2/3*, four intermediate points of the control polygon of the isoparametric Bézier curve $x_{i,j}(u,2/3)$ are computed by

$$W_{k,2} = \sum_{l=0}^{3} W(3i+k, 3j+l) B_l^3\left(2/3\right)$$

$$= \frac{1}{27} W(3i+k, 3j) + \frac{2}{9} W(3i+k, 3j+1) + \frac{4}{9} W(3i+k, 3j+2) + \frac{8}{27} W(3i+k, 3(j+1)), \tag{A.23}$$

where *k=0*,*1*, *2*,and *3*.

For *v=1*, the control polygon of the Bézier curve $x_{i,j}(u,1)$ is *[W(3i, 3(j+1)),W(3i+1, 3(j+1)),W(3i+2,3(j+1)),W(3(i+1), 3(j+1))].* These four vertices are the control points of the boundary of the Bézier patch, $x_{i,j}(u,v)$.

These four isoparametric curves are also illustrated in Figure A.3.

In the analogous manner, four *u*-isoparametric curves of the patch, which are assigned the parameter *u* with *0*, *1/3*, *2/3*, and *1.0*, respectively, can be obtained as follows.

For *u=0*, the control polygon of the isoparametric Bézier curve $x_{i,j}(0,v)$ is *[W(3i, 3j),W(3i, 3j+1),W(3i,3j+2),W(3i, 3(j+1))].*

For *u=1/3*, the control polygon of the isoparametric Bézier curve $x_{i,j}(1/3,v)$ consists of four control points that are intermediate points computed by replacing *u* with *1/3* in Equation A.14 and these control points are

$$\overline{W}_{1,l} = \sum_{k=0}^{3} W(3i+k, 3j+l) B_k^3\left(1/3\right)$$

$$= \frac{8}{27} W(3i, 3j+l) + \frac{4}{9} W(3i+1, 3j+l) + \frac{2}{9} W(3i+2, 3j+l) + \frac{1}{27} W(3(i+1), 3j+l), \qquad \text{(A.24)}$$

where *l=0*,*1*, *2*,and *3*.

For *u=2/3*, four intermediate points of the control polygon of the isoparametric Bézier curve $x_{i,j}(2/3,v)$ are computed by

$$\overline{W}_{2,l} = \sum_{k=0}^{3} W(3i+k, 3j+l) B_k^3\left(2/3\right)$$

$$= \frac{1}{27} W(3i, 3j+l) + \frac{2}{9} W(3i+1, 3j+l) + \frac{4}{9} W(3i+2, 3j+l) + \frac{8}{27} W(3(i+1), 3j+l), \qquad \text{(A.25)}$$

where *l=0*,*1*, *2*,and *3*.

For *u=1*, the control polygon of the Bézier curve $x_{i,j}(1,v)$ is *[W(3(i+1), 3j),W(3(i+1), 3j+1),W(3(i+1),3j+2),W(3(i+1), 3(j+1))].*

A.4 First-order Geometric Continuity of Composite Bézier Surfaces

In this section, we will explore the geometric meaning of composite parametric surfaces with the common boundary curves of G^1. In the studies of Farin, and Hoschek and Lasser (Farin 1993, and Hoschek and Lasser 1993), the condition of two adjacent patches with G^1 is expressed in a lucid geometric manner as Definition 1, rather than in Equation A.3b.

Definition 1. *Two patches with a common boundary curve are called* G^1 *if they have a continuously varying tangent plane along this boundary curve.*

To investigate the continuities of tangent planes of the points on the common boundaries, we have to look for the tangent planes at points that sit on the boundary of two adjoining patches. Without loss of generality, three patches, $x_{i,j}(u,v)$, $x_{i-1,j}(u,v)$, and $x_{i,j-1}(u,v)$, are chosen, as shown in Figure A.4. They are generated with three control nets, each of which has four corner control vertices, being *[W(3i, 3j), W(3i, 3(j+1)), W(3(i+1), 3(j+1)), W(3(i+1), 3j)]*, *[W(3(i-1), 3j), W(3(i-1), 3(j+1)), W(3i, 3(j+1)), W(3i, 3j)]*, and *[W(3i, 3(j-1)), W(3i, 3j), W(3(i+1),3j), W(3(i+1), 3(j-1))]*, respectively. Their common boundary curves are Bézier curves, $x_{i,j}(u,0)$ and $x_{i,j}(0,v)$.

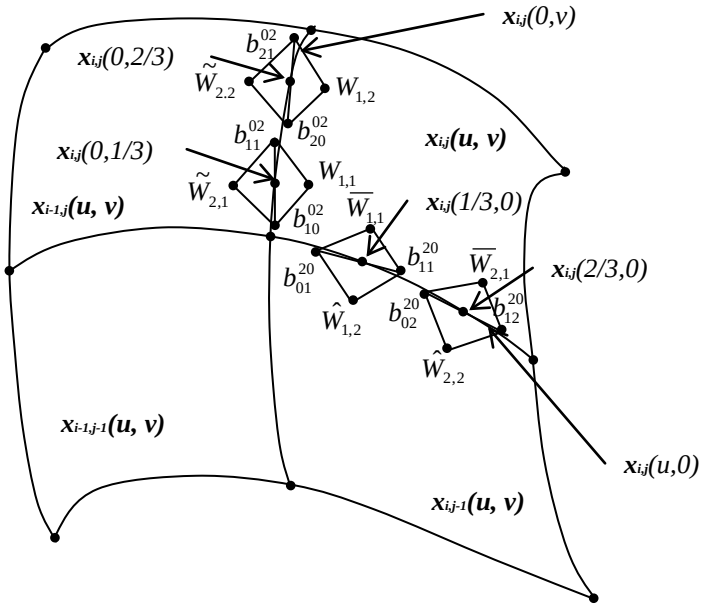


Figure A.4 Tangent Planes on Common Boundary Curves of Neighbouring Bézier Patches, $x_{i,j}(u, v)$, $x_{i-1,j}(u, v)$, and $x_{i,j-1}(u, v)$.

Let us inspect the common boundary, $x_{i,j}(u,0)$. Consider two points on this curve. They are $x_{i,j}(1/3,0)$ and $x_{i,j}(2/3,0)$. With the properties in Section A.1, we investigate the tangent planes at these two points on both sides of this common boundary between two Bézier patches, $x_{i,j}(u,v)$ and $x_{i,j-1}(u,v)$.

With Property 3 and the de Casteljau algorithm, we can obtain two intermediate points passed through by the line that is tangent to the isoparametric Bézier curve $x_{i,j}(u,0)$ at the curve point $x_{i,j}(1/3,0)$. The curve point is written as

$$x_{i,j}(1/3,0) = \frac{8}{27}W(3i,3j) + \frac{4}{9}W(3i+1,3j) + \frac{2}{9}W(3i+2,3j) + \frac{1}{27}W(3(i+1),3j). \quad \text{(A.26)}$$

In the instructive way, the computation process of a point on a bi-cubic Bézier surface with the de Casteljau algorithm is shown in Figure A.5. In this figure, the two superscripts of intermediate coefficients *b*'s represent the order numbers of steps for the de Casteljau

algorithm, the left superscript one is for the parameter *u* and the right one for the *v*; the two subscripts means the order numbers of intermediate coefficients, the left subscript one is for the parameter *u* and the right one for the *v*.

Figure A.5 illustrates the process that the recursive computation is firstly performed on the parameter *u* and then on the parameter *v*. The upper part can generate four coefficients of a *u*-isoparametric curve that the parameter *u* is set to a constant. If the computation order is exchanged by being done firstly on *v* then on *u*, the computation result will not be varied. This algorithm provides a scheme to yield an arbitrary isoparameteric curve as well.

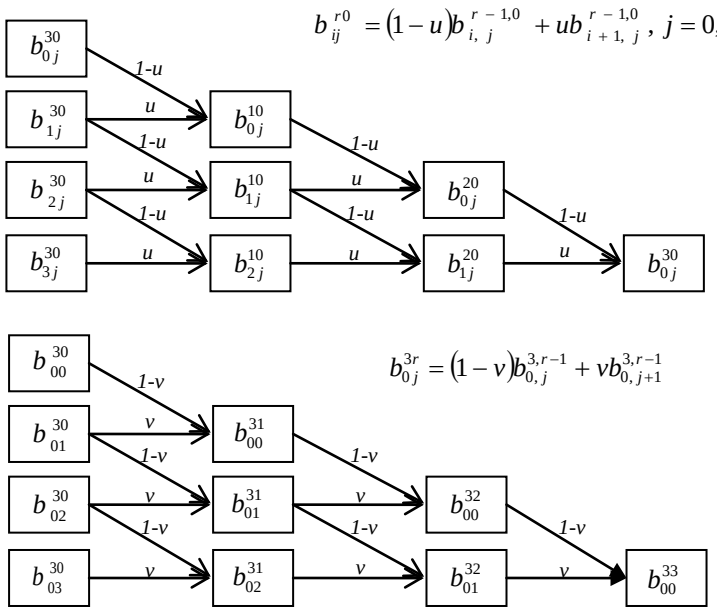


Figure A.5 The de Casteljau Algorithm for Computation of a Point on a Bi-cubic Bézier Surface Consists of Two Parts. The First Part is the Upper Half that is Used to Compute the Coefficients on the *u*-Isoparametric Curves where the Parameter *u* is Set as a Constant. The Second Part is the Lower Half that is Used to Compute the Points on the Bi-cubic Bézier Surface. These Two Parts can be Exchanged with Modifications on Superscripts and Subscripts for First *v* Then *u*. In that Way, the Coefficients on the *v*-Isoparametric Curves, where the Parameter *v* is Set as a Constant, are Computed Firstly. In Each Part, the Computing Process Proceeds from Left to Right by Computing with the Formula at the Right Top of that Part. The Two Subscripts and Superscripts at the Right-hand Sides of Formulas Have Commas in Between Them just for Expressions Without any Ambiguity, but They Have the Same Meanings as Ones without Commas in Other Parts in this Figure and Section.

In Section A.3, we have observed that the control polygon of the isoparametric Bézier curve $x_{i,j}(u,0)$ is *[W(3i, 3j),W(3i+1, 3j),W(3i+2,3j),W(3(i+1), 3j)].* Because we compute the intermediate points on the curve $x_{i,j}(u,0)$, which is both the boundary and isoparametric Bézier curve, the process for computation on the parameter *v* with the de Casteljau

algorithm can be saved. The two intermediate points at the second last step on the parameter *u* at the curve point $x_{i,j}(1/3,0)$ are

$$b_{01}^{20} = \frac{4}{9}W(3i,3j) + \frac{4}{9}W(3i+1,3j) + \frac{1}{9}W(3i+2,3j), \tag{A.27}$$

and

$$b_{11}^{20} = \frac{4}{9}W(3i+1,3j) + \frac{4}{9}W(3i+2,3j) + \frac{1}{9}W(3(i+1),3j). \tag{A.28}$$

Analogously, the curve point $x_{i,j}(2/3,0)$ and its two relevant intermediate points are written as the follows,

$$x_{i,j}(2/3,0) = \frac{1}{27}W(3i,3j) + \frac{2}{9}W(3i+1,3j) + \frac{4}{9}W(3i+2,3j) + \frac{8}{27}W(3(i+1),3j), \tag{A.29}$$

$$b_{02}^{20} = \frac{1}{9}W(3i,3j) + \frac{4}{9}W(3i+1,3j) + \frac{4}{9}W(3i+2,3j), \tag{A.30}$$

and

$$b_{12}^{20} = \frac{1}{9}W(3i+1,3j) + \frac{4}{9}W(3i+2,3j) + \frac{4}{9}W(3(i+1),3j). \tag{A.31}$$

With Properties 1 and 2 and Equation A.24, on the $x_{i,j}(u,v)$ patch side, we can find another tangent at the point $x_{i,j}(1/3,0)$ along the *v* direction, which passes through the following point,

$$\overline{W}_{1,1} = \frac{8}{27}W(3i,3j+1) + \frac{4}{9}W(3i+1,3j+1) + \frac{2}{9}W(3i+2,3j+1) + \frac{1}{27}W(3(i+1),3j+1). \tag{A.32}$$

With Equation A.25, on the $x_{i,j}(u,v)$ patch side, a tangent at the point $x_{i,j}(2/3,0)$ along the *v* direction passes through the following point,

$$\overline{W}_{2,1} = \frac{1}{27}W(3i,3j+1) + \frac{2}{9}W(3i+1,3j+1) + \frac{4}{9}W(3i+2,3j+1) + \frac{8}{27}W(3(i+1),3j+1). \tag{A.33}$$

The $x_{i,j}(1/3,0)$, $x_{i,j}(2/3,0)$, b_{01}^{20}, b_{11}^{20}, b_{02}^{20}, b_{12}^{20}, $\overline{W}_{1,1}$ and $\overline{W}_{2,1}$ are shown in Figure A.4.

On the $x_{i,j-1}(u,v)$ patch side, by using Equations A.24 and A.25 and the Λ mark for distinguishing from the coefficients on the $x_{i,j}(u,v)$ patch side, the tangents at the point $x_{i,j-1}(1/3,1)$ and $x_{i,j-1}(2/3,1)$ along the *v* direction pass, respectively, through the following points,

$$\hat{W}_{1,2} = \frac{8}{27}W(3i,3j-1) + \frac{4}{9}W(3i+1,3j-1) + \frac{2}{9}W(3i+2,3j-1) + \frac{1}{27}W(3(i+1),3j-1), \tag{A.34}$$

and

$$\hat{W}_{2,2} = \frac{1}{27}W(3i,3j-1) + \frac{2}{9}W(3i+1,3j-1) + \frac{4}{9}W(3i+2,3j-1) + \frac{8}{27}W(3(i+1),3j-1). \quad \text{(A.35)}$$

Next let us inspect the common boundary, $x_{i,j}(0,v)$. Consider two points on this curve, these being $x_{i,j}(0,1/3)$ and $x_{i,j}(0,2/3)$. We investigate tangent planes at these two points on both sides of the common boundary between two Bézier patches, $x_{i-1,j}(u,v)$ and $x_{i,j}(u,v)$.

With Property 3 and the de Casteljau algorithm, we can obtain two intermediate points passed through by the line that is tangent to the isoparametric Bézier curve $x_{i,j}(0,v)$ at the curve point $x_{i,j}(0,1/3)$. The curve point and its two relative intermediate points are

$$x_{i,j}(0,1/3) = \frac{8}{27}W(3i,3j) + \frac{4}{9}W(3i,3j+1) + \frac{2}{9}W(3i,3j+2) + \frac{1}{27}W(3i,3(j+1)), \quad \text{(A.36)}$$

$$b_{10}^{02} = \frac{4}{9}W(3i,3j) + \frac{4}{9}W(3i,3j+1) + \frac{1}{9}W(3i,3j+2), \quad \text{(A.37)}$$

and

$$b_{11}^{02} = \frac{4}{9}W(3i,3j+1) + \frac{4}{9}W(3i,3j+2) + \frac{1}{9}W(3i,3(j+1)). \quad \text{(A.38)}$$

Analogously, the curve point $x_{i,j}(0,2/3)$ and its two relevant intermediate points are

$$x_{i,j}(0,2/3) = \frac{1}{27}W(3i,3j) + \frac{2}{9}W(3i,3j+1) + \frac{4}{9}W(3i,3j+2) + \frac{8}{27}W(3i,3(j+1)), \quad \text{(A.39)}$$

$$b_{20}^{02} = \frac{1}{9}W(3i,3j) + \frac{4}{9}W(3i,3j+1) + \frac{4}{9}W(3i,3j+2), \quad \text{(A.40)}$$

and

$$b_{21}^{02} = \frac{1}{9}W(3i,3j+1) + \frac{4}{9}W(3i,3j+2) + \frac{4}{9}W(3i,3(j+1)). \quad \text{(A.41)}$$

On the $x_{i,j}(u,v)$ patch side, with Equation A.22, we can find another tangent at the point $x_{i,j}(0,1/3)$ along the u direction, which passes through the following point,

$$W_{1,1} = \frac{8}{27}W(3i+1,3j) + \frac{4}{9}W(3i+1,3j+1) + \frac{2}{9}W(3i+1,3j+2) + \frac{1}{27}W(3i+1,3(j+1)). \quad \text{(A.42)}$$

On the $x_{i,j}(u,v)$ patch side, with Equation A.23, a tangent at the point $x_{i,j}(0,2/3)$ along the u direction passes through the following point,

$$W_{1,2} = \frac{1}{27}W(3i+1,3j) + \frac{2}{9}W(3i+1,3j+1) + \frac{4}{9}W(3i+1,3j+2) + \frac{8}{27}W(3i+1,3(j+1)). \quad \text{(A.43)}$$

On the $x_{i-1,j}(u,v)$ patch side, with Equations A.22 and A.23 and by using the $\tilde{}$ mark for distinguishing from the coefficients on the $x_{i,j}(u,v)$ patch side, the tangents at the point $x_{i-1,j}(1,1/3)$ and $x_{i-1,j}(1,2/3)$ along the u direction pass, respectively, through the following points,

$$\tilde{W}_{2,1} = \frac{8}{27}W(3i-1,3j)+\frac{4}{9}W(3i-1,3j+1)+\frac{2}{9}W(3i-1,3j+2)+\frac{1}{27}W(3i-1,3(j+1)), \quad \text{(A.44)}$$

and

$$\tilde{W}_{2,2} = \frac{1}{27}W(3i-1,3j)+\frac{2}{9}W(3i-1,3j+1)+\frac{4}{9}W(3i-1,3j+2)+\frac{8}{27}W(3i-1,3(j+1)). \quad \text{(A.45)}$$

Now, we make four remarks: the plane formed with three points, $\overline{W}_{1,1}$, b_{01}^{20} and b_{11}^{20}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(1/3,0)$ on the common boundary $x_{i,j}(u,0)$; the plane formed with three points, $\overline{W}_{2,1}$, b_{02}^{20} and b_{12}^{20}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(2/3,0)$ on the common boundary $x_{i,j}(u,0)$; the plane formed with three points, $\hat{W}_{1,2}$, b_{01}^{20} and b_{11}^{20}, is a tangent plane of the $x_{i,j-1}(u,v)$ patch at the point $x_{i,j-1}(1/3,1)$ on the common boundary $x_{i,j-1}(u,1)$; the plane formed with three points, $\hat{W}_{2,2}$, b_{02}^{20} and b_{12}^{20}, is a tangent plane of the $x_{i,j-1}(u,v)$ patch at the point $x_{i,j-1}(2/3,1)$ on the common boundary $x_{i,j-1}(u,1)$. Notice that the $x_{i,j}(1/3,0)$ and $x_{i,j-1}(1/3,1)$ are the same point on the common boundary, $x_{i,j}(u,0)$ (or $x_{i,j-1}(u,1)$), and so are the $x_{i,j}(2/3,0)$ and $x_{i,j-1}(2/3,1)$. These tangent planes are illustrated in Figure A.4.

Analogously, we can make four remarks of tangent planes on the common boundary $x_{i,j}(0,v)$. The plane formed with three points, $W_{1,1}$, b_{10}^{02} and b_{11}^{02}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(0,1/3)$ on the common boundary $x_{i,j}(0,v)$; the plane formed with three points, $W_{1,2}$, b_{20}^{02} and b_{21}^{02}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(0,2/3)$ on the common boundary $x_{i,j}(0,v)$; the plane formed with three points, $\tilde{W}_{2,1}$, b_{10}^{02} and b_{11}^{02}, is a tangent plane of the $x_{i-1,j}(u,v)$ patch at the point $x_{i-1,j}(1,1/3)$ on the common boundary $x_{i-1,j}(1,v)$; and the plane formed with three points, $\tilde{W}_{2,2}$, b_{20}^{02} and b_{21}^{02}, is a tangent plane of the $x_{i-1,j}(u,v)$ patch at the point $x_{i-1,j}(1,2/3)$ on the common boundary $x_{i-1,j}(1,v)$. The $x_{i,j}(0,1/3)$ and $x_{i-1,j}(1,1/3)$ are the same point on the common boundary, $x_{i,j}(0,v)$ (or $x_{i-1,j}(1,v)$), and so are the $x_{i,j}(0,2/3)$ and $x_{i-1,j}(1,2/3)$. These tangent planes are also shown in Figure A.4.

In the next section, we will give the proofs for the above remarks. Before proving these remarks, we require a definition of a tangent plane of a Bézier surface.

A.5 Tangent Planes of a Bézier Surface

According to the study of Dubuluowen et al 2006 (Dubuluowen et al 2006), the tangent plane of a Bézier surface is expressed as Definition 2.

Definition 2. *Given a Bézier surface* of *degree (m, n) in* E^3,

$$x(u,v)=\sum_{k=0}^{m}\sum_{l=0}^{n}b_{k,l}B_k^m(u)B_l^n(v), \text{ where } 0\le u,v\le 1, \quad \text{(A.46)}$$

its partial derivatives with respect of variates u and v are written as, respectively,

$$\frac{\partial x}{\partial u}=\frac{\partial}{\partial u}x(u,v)$$

and

$\frac{\partial x}{\partial v}=\frac{\partial}{\partial v}x(u,v)$, *both of which are vectors in* R^3. *An arbitrary tangent vector of the surface x(u, v) at the point (u, v) is a linear combination of two partial derivative vectors,* $\frac{\partial x}{\partial u}$ and $\frac{\partial x}{\partial v}$. *The subspace of* R^2 *space (two-dimensional Riemannian space) spanned by two partial derivative vectors,* $\frac{\partial x}{\partial u}$ and $\frac{\partial x}{\partial v}$, *is the tangent plane of the surface x(u, v) at the point (u, v).*

Equations A.16 and A.46 are identical. To copy it in Definition 2 is just for facilitating the discussion.

In the Bézier curve case, Equation A.5 is the derivative. Analogously, the *u*-partial derivative of the Bézier surface in the form of Equation A.46 is computed with a series of equations as follows (Farin 1993),

$$\frac{\partial x}{\partial u}=\frac{\partial}{\partial u}x(u,v)=\sum_{l=0}^{n}\left[\frac{\partial}{\partial u}\sum_{k=0}^{m}b_{k,l}B_k^m(u)\right]B_l^n(v)=m\sum_{k=0}^{m-1}\sum_{l=0}^{n}\Delta^{1,0}b_{k,l}B_k^{m-1}(u)B_l^n(v), \quad \text{(A.47)}$$

where $\Delta^{1,0}b_{k,l}=b_{k+1,l}-b_{k,l}$ and the one of left superscript represents the differential operation performed only on the first subscripts of control vertices, $b_{k,l}$. Its *v*-partial derivative is computed in the next equation,

$$\frac{\partial x}{\partial v}=n\sum_{k=0}^{m}\sum_{l=0}^{n-1}\Delta^{0,1}b_{k,l}B_k^m(u)B_l^{n-1}(v), \quad \text{(A.48)}$$

where $\Delta^{0,1}b_{k,l}=b_{k,l+1}-b_{k,l}$ and the one of right superscript means the differential operation performed only on the second subscripts of $b_{k,l}$.

Property 5. *For a Bézier surface* of *degree (m, n) in* E^3 *with the form of Equation A.46, a v-partial derivative of this surface at a point (u,v) is a tangent vector of a u-isoparametric curve of this surface at this point, and a u-partial derivative of this surface at the point (u,v) is a tangent vector of its v-isoparametric curve at this point*.

Proof of Property 5. Let us prove the *v*-partial derivative case. With Property 4 and Equations A.17 and A.19, a *u*-isoparametric curve of Equation A.46 (or A.16) is a Bézier curve with degree *n*. Thus, Properties 1, 2, and 3 are true for a *u*-isoparametric curve, and the de Casteljau algorithm for a Bézier curve is true for a *u*-isoparametric curve as well. To facilitate the discussion, we copy Equations A.17 and A.19 here,

$$x(c,v) = \sum_{k=0}^{m}\sum_{l=0}^{n} b_{k,l} B_k^m(c) B_l^n(v) = \sum_{l=0}^{n}[\sum_{k=0}^{m} b_{k,l} B_k^m(c)] B_l^n(v)$$

and

$$x(c,v) == \sum_{l=0}^{n} \hat{b}_l B_l^n(v), \text{ where } \hat{b}_l = \sum_{k=0}^{m} b_{k,l} B_k^m(c).$$

With the de Casteljau algorithm for a Bézier curve and referring to Equation 13 in the Proof of Property 3, we obtain a tangent of the *u*-isoparametric curve *x(c,v)* as follows,

$$\hat{b}_1^{n-1}(v) - \hat{b}_0^{n-1}(v) = \sum_{l=0}^{n-1}(\hat{b}_{l+1} - \hat{b}_l) B_l^{n-1}(v),$$

where the superscript *n-1* of $\hat{b}_1^{n-1}$ is the order number of the de Casteljau algorithm, which also means the second last step.

By substituting $\hat{b}_l$ and $\hat{b}_{l+1}$ with Equation A.18, we further obtain the following equation,

$$\hat{b}_1^{n-1}(v) - \hat{b}_0^{n-1}(v) = \sum_{l=0}^{n-1}\sum_{k=0}^{m} (b_{k,l+1} - b_{k,l}) B_k^m(c) B_l^{n-1}(v). \quad \text{(A.49)}$$

Let us inspect *v*-partial derivatives that are obtained with Equation A.48,

$$\frac{\partial x}{\partial v} = n \sum_{k=0}^{m}\sum_{l=0}^{n-1} \Delta^{0,1} b_{k,l} B_k^m(u) B_l^{n-1}(v), \text{ where } \Delta^{0,1} b_{k,l} = b_{k,l+1} - b_{k,l}.$$

When we take *v*-partial derivatives, we can treat the *u* parameter as a constant, saying *c*. Taking *u* = *c* in Equation A.48, compare Equation A.48 to Equation A.49. After normalised, these two equations represent the same unit vector. Since *c* is arbitrary, Equation A.48 is a *v*-partial derivative of the Bézier surface *x(u,v)* at the point *(u,v)* and Equation A.49 indicates the tangent of a *u*-isoparametric curve of the Bézier surface *x(u,v)* at this point.

The *u*-partial derivative case can be proved straightforward by exchanging *u* and *v* and modifying the relative superscripts and subscripts. We will not give details of its proof.

The proof of Property 5 is accomplished. □

Let us return the proof of remarks at the end of Section A.4.

With Definition 2 and Properties 2, 3, 4, and 5, since both the line passing through b_{01}^{20} and b_{11}^{20} and line through $\overline{W_{1,1}}$ and $x_{i,j}$*(1/3,0)* in Figure A.4 are tangents on the $x_{i,j}$*(u,v)*

patch side, the plane formed with three points, $\overline{W}_{1,1}$, b_{01}^{20} and b_{11}^{20} is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(1/3,0)$ on the common boundary $x_{i,j}(u,0)$. Thus, the first remark is true. With the similar reasons, the other remarks are true as well. □

To articulate them, we list these remarks palpably.

- On the $x_{i,j}(u,v)$ side, the plane formed with three points, $\overline{W}_{1,1}$, b_{01}^{20} and b_{11}^{20}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(1/3,0)$ on the common boundary $x_{i,j}(u,0)$.
- The plane formed with three points, $\overline{W}_{2,1}$, b_{02}^{20} and b_{12}^{20}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(2/3,0)$ on the common boundary $x_{i,j}(u,0)$.
- On the $x_{i,j-1}(u,v)$ patch side, the plane formed with three points, $\hat{W}_{1,2}$, b_{01}^{20} and b_{11}^{20}, is a tangent plane of the $x_{i,j-1}(u,v)$ patch at the point $x_{i,j}(1/3,0)$ on the common boundary $x_{i,j}(u,0)$.
- The plane formed with three points, $\hat{W}_{2,2}$, b_{02}^{20} and b_{12}^{20} is a tangent plane of the $x_{i,j-1}(u,v)$ patch at the point $x_{i,j}(2/3,0)$ on the common boundary $x_{i,j}(u,0)$.
- On the $x_{i,j}(u,v)$ side, the plane formed with three points, $W_{1,1}$, b_{10}^{02} and b_{11}^{02}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(0,1/3)$ on the common boundary $x_{i,j}(0,v)$.
- The plane formed with three points, $W_{1,2}$, b_{20}^{02} and b_{21}^{02}, is a tangent plane of the $x_{i,j}(u,v)$ patch at the point $x_{i,j}(0,2/3)$ on the common boundary $x_{i,j}(0,v)$.
- On the $x_{i-1,j}(u,v)$ side, the plane formed with three points, $\tilde{W}_{2,1}$, b_{10}^{02} and b_{11}^{02}, is a tangent plane of the $x_{i-1,j}(u,v)$ patch at the point $x_{i,j}(0,1/3)$ on the common boundary $x_{i,j}(0,v)$.
- The plane formed with three points, $\tilde{W}_{2,2}$, b_{20}^{02} and b_{21}^{02}, is a tangent plane of the $x_{i-1,j}(u,v)$ patch at the point $x_{i,j}(0,2/3)$ on the common boundary $x_{i,j}(0,v)$.

A.6 Analysis of First Order Geometric Continuities

To construct a sufficient condition for two adjacent bi-cubic Bézier patches, we refer to the scheme of Farin (Farin 1993) to construct a sufficient condition for two adjacent triangular Bézier patches to be G^1. According to Definition 1, a sufficient condition for two adjacent bi-cubic Bézier patches is that these two patches have the same tangent planes along their common boundary curve.

Following the discussion of Section A.5, we have four cases,

- For two patches, $x_{i,j}(u,v)$ and $x_{i,j-1}(u,v)$, the condition of the same tangent plane on the point $x_{i,j}(1/3,0)$ of the common boundary curve $x_{i,j}(u,0)$ is equivalent to that four points, $\overline{W}_{1,1}$, $\hat{W}_{1,2}$, b_{01}^{20} and b_{11}^{20}, are coplanar.
- The condition of the same tangent plane on the point $x_{i,j}(2/3,0)$ is equivalent to that four points, $\overline{W}_{2,1}$, $\hat{W}_{2,2}$, b_{02}^{20} and b_{12}^{20}, are coplanar.
- For two patches, $x_{i,j}(u,v)$ and $x_{i-1,j}(u,v)$, the condition of the same tangent plane on the point $x_{i,j}(0,1/3)$ of the common boundary curve $x_{i,j}(0,v)$ is equivalent to that four points, $W_{1,1}$, $\tilde{W}_{2,1}$, b_{10}^{02} and b_{11}^{02}, are coplanar.
- The condition of the same tangent planes on the point $x_{i,j}(0,2/3)$ is equivalent to that four points, $W_{1,2}$, $\tilde{W}_{2,2}$, b_{20}^{02} and b_{21}^{02}, are coplanar.

All the above four cases are shown in Figure A.4.

Take the case of $\overline{W}_{1,1}$, $\hat{W}_{1,2}$, b_{01}^{20} and b_{11}^{20}. Let us inspect b_{01}^{20} and b_{11}^{20}. It has been observed that points, b_{01}^{20} and b_{11}^{20}, are collinear with $x_{i,j}(1/3,0)$. Referring to Figure A.4, if the coplanarity of $\overline{W}_{1,1}$, $\hat{W}_{1,2}$, b_{01}^{20} and b_{11}^{20} is held, this means that when a point moving and crossing the point $x_{i,j}(1/3,0)$ from one patch (such as $x_{i,j-1}(u,v)$) to another ($x_{i,j}(u,v)$) along the v direction, the moving direction keeps in the same tangent plane without change. Otherwise, the moving direction changes to another tangent plane and the point's locus develops a corner when the point crosses the common boundary between patches, $x_{i,j-1}(u,v)$ and $x_{i,j}(u,v)$. This leads to a fold along the boundary curve $x_{i,j}(u,0)$ when all the points of a v-isoparametric curve moving from one patch (such as $x_{i,j-1}(u,v)$) to another ($x_{i,j}(u,v)$).

The sufficient condition of the same tangent plane is further equivalent to that numbers α and θ exist such that

$$\alpha b_{01}^{20} + (1-\alpha) b_{11}^{20} = \theta \overline{W}_{1,1} + (1-\theta)\hat{W}_{1,2} . \tag{A.50}$$

The geometric meaning of Equation A.50 is illustrated in Figure A.6. The line passing through two points $\overline{W}_{1,1}$ and $\hat{W}_{1,2}$ has an intersection (P) with the line passing through two points b_{01}^{20} and b_{11}^{20}. The ratio of lengths of two line segments, $[\overline{W}_{1,1}, P]$ and $[P, \hat{W}_{1,2}]$ is $(1-\theta):\theta$, and the ratio of lengths of two line segments, $[b_{01}^{20}, P]$ and $[P, b_{11}^{20}]$ is $(1-\alpha):\alpha$.

To inspect the other three cases, $[\overline{W}_{2,1}, \hat{W}_{2,2}, b_{02}^{20}, b_{12}^{20}]$, $[W_{1,1}, \tilde{W}_{2,1}, b_{10}^{02}, b_{11}^{02}]$, and

[$W_{1,2}$, $\tilde{W}_{2,2}$, b_{20}^{02}, b_{21}^{02}*],* we should investigate whether or not different pairs of numbers of α and θ exist to meet the condition that is replaced with the corresponding coefficients in Equation A.50 for each of them. Furthermore, considering all the points on a common boundary curve, to meet the condition of first-order geometric continuity, we have to find pairs of numbers of α and θ for them. The study of Hoschek and Lasser 1993 provides the special functions for α and θ to create a practical method to construct control vertices. The authors of Hoschek and Lasser 1993 state that the special choice of the factors α and θ may cause that free parameters inside patches are not sufficient to meet all the boundary conditions. For example, if two adjoining patches are distinct in shape, or if along their common boundary they have very different 'radii', the special choice of factors α and θ can be too restricted to allow a sharp shape change when points' crossing the boundary.

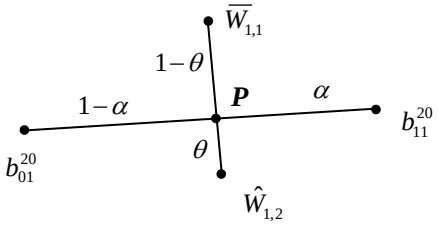


Figure A.6 The Geometric Meaning of the Coplanarity of $\overline{W}_{1,1}$, $\hat{W}_{1,2}$, b_{01}^{20} and b_{11}^{20}.

In addition, if factors, α and θ, adopt polynomial functions, they can meet the conditions of higher-order geometric continuities on common boundaries. However, the authors of Hoschek and Lasser 1993 indicate that raising the polynomial degrees of factor functions does not really increase the number of available degrees of freedom for the design purpose.

By substituting b_{01}^{20}, b_{11}^{20}, $\overline{W}_{1,1}$, and $\hat{W}_{1,2}$ with Equations A.27, A.28, A.32, and A.34 in Equation A.50, we derive

$$\alpha\left(\frac{4}{9}W(3i,3j)+\frac{4}{9}W(3i+1,3j)+\frac{1}{9}W(3i+2,3j)\right)$$

$$+(1-\alpha)\left(\frac{4}{9}W(3i+1,3j)+\frac{4}{9}W(3i+2,3j)+\frac{1}{9}W(3(i+1),3j)\right)$$

$$=\theta\left(\frac{8}{27}W(3i,3j+1)+\frac{4}{9}W(3i+1,3j+1)+\frac{2}{9}W(3i+2,3j+1)+\frac{1}{27}W(3(i+1),3j+1)\right)$$

$$+(1-\theta)\left(\frac{8}{27}W(3i,3j-1)+\frac{4}{9}W(3i+1,3j-1)+\frac{2}{9}W(3i+2,3j-1)+\frac{1}{27}W(3(i+1),3j-1)\right). \quad \text{(A.51)}$$

If we tried to make four points, b_{01}^{20}, b_{11}^{20}, $\overline{W}_{1,1}$, and $\hat{W}_{1,2}$, coplanar, equation A.51 should be met rigorously. Even worse, each point along common boundary curves should meet their own conditions similar to Equation A.51 with replacement of corresponding coefficients. This is not practical for the design purpose. For this reason, the PAMA does not follow this condition on purpose. The following sections present the constructions of control points with the PAMA.

A.7 Construction of Control Points on Common Boundaries with PAMA

Since cubic Bézier curves are parametric curves, with Equations A.2a-c, we can construct a composite curve with two cubic Bézier curve segments. In Section 6.2 of Chapter 6, we have discussed conditions of two Bézier curve segments, *T(u)* and *S(w)*, $0 \le u, w \le 1$, meeting G^0, G^1 and G^2. The condition for G^0 is that the end point of the first segment meets with the start point of the second segment. That is, these two points have the same position. The condition for G^1 is that these two segments have the same unit tangent vector at their common point. The condition for G^2 is that these two segments have the same curvature vector at their common point. Referring to Equations A.2a-c, these conditions are written as the following equations, which are also called Beta constraints (Barsky and DeRose 1989),

$$T(0) = S(1), \tag{A.52a}$$

$$\frac{d}{du}T(0) = \beta_1 \frac{d}{du}S(1), \tag{A.52b}$$

and

$$\frac{d^2}{du^2}T(0) = \beta_1^2 \frac{d^2}{du^2}S(1) + \beta_2 \frac{d}{du}S(1). \tag{A.52c}$$

These three equations can also be written as Equations 7.1, 7.2 and 7.4. With Beta constraints, these being Equations A.52a-b, and the geometric approach that is presented by Farin (Farin 1982), improved by Boehm (Boehm 1985), and applied to Beta-spline curves by Barsky and DeRose (Barsky and DeRose 1989), the control polygons of cubic Bézier curve segments with shape parameters β_1 and β_2 can be generated. Given original control points *V(i)*'s, a cubic Bézier segment is constructed between each adjoining pair of *V(i) and V(i+1)*. The control points, *W(j)*'s, of cubic Bézier curve segments are written as the following equations,

$$W(3i+1) = \frac{(1+\beta_1^2(i+1)\gamma(i+1))V(i) + \gamma(i)V(i+1)}{1+\gamma(i)+\beta_1^2(i+1)\gamma(i+1)}, \tag{A.53a}$$

$$W(3i+2)=\frac{\beta_1^2(i+1)\gamma(i+1)V(i)+(1+\gamma(i))V(i+1)}{1+\gamma(i)+\beta_1^2(i+1)\gamma(i+1)}, \quad \text{(A.53b)}$$

and

$$W(3i)=\frac{\beta_1(i)W(3(i-1)+2)+W(3i+1)}{1+\beta_1(i)}, \quad \text{(A.53c)}$$

where $\gamma(i)=\frac{2(1+\beta_1(i))}{\beta_2(i)+2\beta_1(i)(1+\beta_1(i))}$.

The geometric meaning is illustrated in Figure A.7. *[W(3i),W(3i+1),W(3i+2),W(3(i+1))]* are the control polygon of a cubic Bézier segment. The ratio of lengths of three line segments, *[V(i),W(3i+1)]*, *[W(3i+1),W(3i+2)]*, and *[W(3i+2),V(i+1)]*, is $\gamma(i):1:\beta_1^2(i+1)\gamma(i+1)$. The ratio of lengths of two line segments, *[W(3(i-1)+2),W(3i)]* and *[W(3i) ,W(3i+1)]*, is $1:\beta_1(i)$.

Following the discussion in Section A.1, we can state that the condition of G^2 of composite surfaces (Equation A.3c) cannot provide a practical design scheme to join two adjoining parametric surface patches. However, since the isoparametric curves of Bézier surfaces are Bézier curves, we can let these curves meet G^2 by setting the special conditions on their second-order partial derivatives. Therefore, in the rest of this section, we will involve in the $\partial^2/\partial u^2$ and $\partial^2/\partial v^2$, and in the next section, we will discuss the $\partial^2/\partial u\partial v$, also called twist.

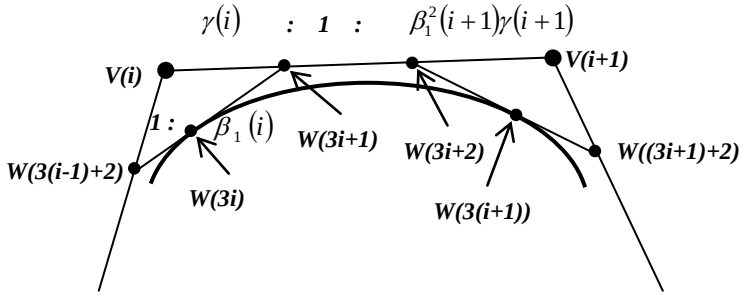


Figure A.7 Control Polygon of a Cubic Bézier Segment Generated with the Geometric Approach to Meet G^2.

Referring to three Equations A.52a-b, we can write the equations for points on common boundary curves of two bi-cubic Bézier patches along the *u* or *v* direction, respectively. Along the *u* direction, they are

$$x_{i,j}(0,v)=x_{i-1,j}(1,v), \quad \text{(A.54a)}$$

$$\frac{\partial}{\partial u} x_{i,j}(0,v) = \beta_{u1}(i,j) \frac{\partial}{\partial u} x_{i-1,j}(1,v), \tag{A.54b}$$

and

$$\frac{\partial^2}{\partial u^2} x_{i,j}(0,v) = \beta_{u1}^2(i,j) \frac{\partial^2}{\partial u^2} x_{i-1,j}(1,v) + \beta_{u2}(i,j) \frac{\partial}{\partial u} x_{i-1,j}(1,v). \tag{A.54c}$$

Along the *v* direction, they are

$$x_{i,j}(u,0) = x_{i,j-1}(u,1), \tag{A.55a}$$

$$\frac{\partial}{\partial v} x_{i,j}(u,0) = \beta_{v1}(i,j) \frac{\partial}{\partial v} x_{i,j-1}(u,1), \tag{A.55b}$$

and

$$\frac{\partial^2}{\partial v^2} x_{i,j}(u,0) = \beta_{v1}^2(i,j) \frac{\partial^2}{\partial v^2} x_{i,j-1}(u,1) + \beta_{v2}(i,j) \frac{\partial}{\partial v} x_{i,j-1}(u,1). \tag{A.55c}$$

In the similar way as the curve case, with Equations A.54a-b and setting *v = 0*, we can generate points on the common boundary, *W(3i+1,3j)* and *W(3i+2,3j),* as shown in Figure A.8 (a),

$$W(3i+1,3j) = \frac{(1+\beta_{u1}^2(i+1,j)\gamma(i+1,j))V(i,j) + \gamma_u(i,j)V(i+1,j)}{1+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\gamma_u(i+1,j)}, \tag{A.56a}$$

and

$$W(3i+2,3j) = \frac{\beta_{u1}^2(i+1,j)\gamma_u(i+1,j)V(i,j) + (1+\gamma_u(i,j))V(i+1,j)}{1+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\gamma_u(i+1,j)}, \tag{A.56b}$$

where $\gamma_u(i,j) = \dfrac{2(1+\beta_{u1}(i,j))}{\beta_{u2}(i,j)+2\beta_{u1}(i,j)(1+\beta_{u1}(i,j))}$.

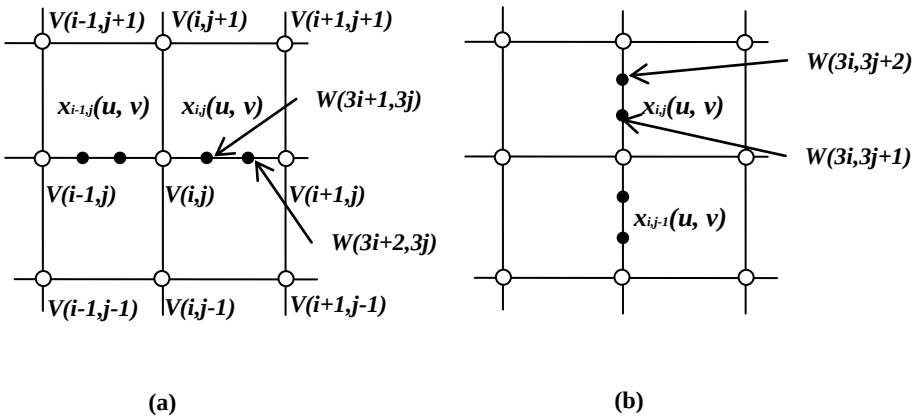


Figure A.8 Generation of Points on Common Boundaries of Bézier Patches. (a) For Points Along the *u* Direction; (b) For Points Along the *v* Direction.

With Equations A.55a-b and setting *u = 0*, we can derive points on the common boundary, *W(3i,3j+1)* and *W(3i,3j+2),* as shown in Figure A.8 (b),

$$W(3i,3j+1)=\frac{(1+\beta_{v1}^2(i,j+1)\gamma(i,j+1))V(i,j)+\gamma_v(i,j)V(i,j+1)}{1+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\gamma_v(i,j+1)}, \quad \text{(A.56c)}$$

and

$$W(3i,3j+2)=\frac{\beta_{v1}^2(i,j+1)\gamma_v(i,j+1)V(i,j)+(1+\gamma_v(i,j))V(i,j+1)}{1+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\gamma_v(i,j+1)}, \quad \text{(A.56d)}$$

where $\gamma_v(i,j)=\dfrac{2(1+\beta_{v1}(i,j))}{\beta_{v2}(i,j)+2\beta_{v1}(i,j)(1+\beta_{v1}(i,j))}$.

These equations are just part of the scheme used to yield the first-interpolated points with the PAMA. They have been given in Section 7.3 of Chapter 7. Equations A56a-d are also Equations 7.6, 7.7, 7.9, and 7.10.

A.8 Twists and Constructions of Corner Points with PAMA

Twists are the mixed partial derivatives, $\dfrac{\partial^2}{\partial u\partial v}$. The twists of the Bézier surface of degree *(m, n)* expressed in Equation A.46 can be deduced with Equation A.47 and the following equations,

$$\frac{\partial^2 x}{\partial u\partial v}=\frac{\partial}{\partial v}\left(\frac{\partial x}{\partial u}\right)=m\sum_{k=0}^{m-1}B_k^{m-1}(u)\frac{\partial}{\partial v}\left[\sum_{l=0}^{n}\Delta^{1,0}b_{k,l}B_l^n(v)\right]=mn\sum_{k=0}^{m-1}\sum_{l=0}^{n-1}\left(\Delta^{1,0}b_{k,l+1}-\Delta^{1,0}b_{k,l}\right)B_k^{m-1}(u)B_l^{n-1}(v)$$

$$=mn\sum_{k=0}^{m-1}\sum_{l=0}^{n-1}\Delta^{1,1}b_{k,l}B_k^{m-1}(u)B_l^{n-1}(v), \quad \text{(A.57)}$$

where $\Delta^{1,1}b_{k,l}=b_{k+1,l+1}-b_{k,l+1}-b_{k+1,l}+b_{k,l}$. (A.58)

Let us inspect four points in the twist vector $\Delta^{1,1}b_{k,l}$, $b_{k+1,l+1}, b_{k,l+1}, b_{k+1,l}$, and $b_{k,l}$. If one of these four points is a corner of a patch, the twist is an important factor for the geometric property of the composite surface, as shown in Figure A.9. In this figure, *W(3i,3j)* is the corner point.

For each of four patches, $x_{i-1,j-1}(u,v)$, $x_{i-1,j}(u,v)$, $x_{i,j-1}(u,v)$, and $x_{i,j}(u,v)$, *W(3i,3j)* has different indices for Equation A.58. Take the $x_{i,j}(u,v)$ as an example. According to Farin 1993, the geometric interpretation of the twist at the patch corner, *W(3i,3j),* is the deviation of the corner subquadrilateral [*W(3i,3j), W(3i,3j+1), W(3i+1,3j+1), W(3i+1,3j)*] of the control net from the tangent plane formed with three boundary points, [*W(3i,3j), W(3i,3j+1), W(3i+1,3j)*]. The twist vector $\Delta^{1,1}b_{k,l}$ is a measure for the deviation of the inside point

(*W(3i+1,3j+1)*) from the tangent plane at the corner (*W(3i,3j)*). Analogously, we can write the twist vectors for four patches, $x_{i-1,j-1}(u,v)$, $x_{i,j-1}(u,v)$, $x_{i-1,j}(u,v)$, and $x_{i,j}(u,v)$ respectively, as

$$\Delta^{1,1}W(3(i-1)+2,3(j-1)+2) = W(3i,3j) - W(3(i-1)+2,3j) - W(3i,3(j-1)+2) + W(3(i-1)+2,3(j-1)+2),$$

$$\Delta^{1,1}W(3i,3(j-1)+2) = W(3i+1,3j) - W(3i,3j) - W(3i+1,3(j-1)+2) + W(3i,3(j-1)+2),$$

$$\Delta^{1,1}W(3(i-1)+2,3j) = W(3i,3j+1) - W(3(i-1)+2,3j+1) - W(3i,3j) + W(3(i-1)+2,3j),$$

and

$$\Delta^{1,1}W(3i,3j) = W(3i+1,3j+1) - W(3i,3j+1) - W(3i+1,3j) + W(3i,3j).$$

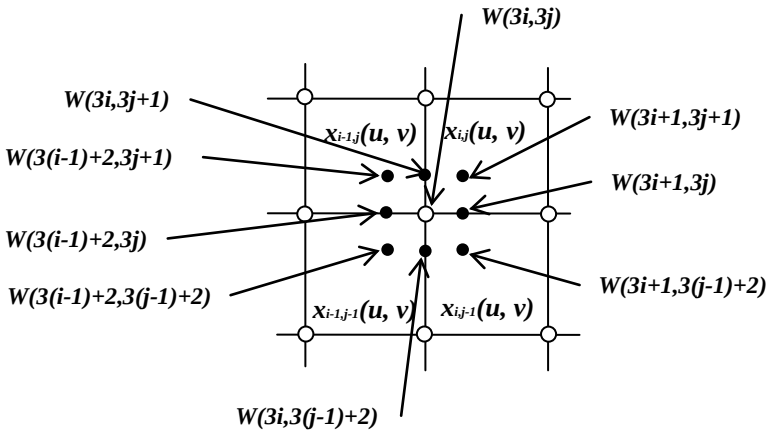


Figure A.9 Control Points Related to Twists at Four Patch Corners.

The C^1 patches have the same twist along the common boundary on both sides, especially the same four twists at the corner (Farin 1993). Therefore, it is restrictive to have the same twist at the corner for four adjoining patches and for two adjoining patches along common boundary curves. For this reason, the PAMA does not exert the condition of the same twists at the corner of four adjoining patches.

In general, control points along both *u* and *v* directions have effects on corner points and should be considered at the same time. Referring to Equation A.53c, the PAMA blends the effects of control points along *u* and *v* directions into one equation with the linear interpolation, which is written as

$$W(3i,3j) =$$

$$\frac{\beta_{u1}(i,j)W(3(i-1)+2,3j) + W(3i+1,3j) + \beta_{v1}(i,j)W(3i,3(j-1)+2) + W(3i,3j+1)}{2+\beta_{u1}(i,j)+\beta_{v1}(i,j)}. \qquad \text{(A.59)}$$

Equation A.59 is also Equation 7.11 in Chapter 7.

In Section A.7 and Figure A.7, we have discussed the construction of the intersection

(*W(3i)*) of two Bézier line segments with the method of Frain, Boehm, Barsky and DeRose (Barsky and DeRose 1989, Boehm 1985, and Farin 1982). To see it clearly, we re-write Equation A.59 as follows,

$$W(3i,3j)=\frac{1+\beta_{u1}(i,j)}{2+\beta_{u1}(i,j)+\beta_{v1}(i,j)}\cdot\frac{\beta_{u1}(i,j)W(3(i-1)+2,3j)+W(3i+1,3j)}{1+\beta_{u1}(i,j)}+$$

$$\frac{1+\beta_{v1}(i,j)}{2+\beta_{u1}(i,j)+\beta_{v1}(i,j)}\cdot\frac{\beta_{v1}(i,j)W(3i,3(j-1)+2)+W(3i,3j+1)}{1+\beta_{v1}(i,j)}$$

$$=\frac{1+\beta_{u1}(i,j)}{2+\beta_{u1}(i,j)+\beta_{v1}(i,j)}\cdot\overline{W}(3i,3j)+\frac{1+\beta_{v1}(i,j)}{2+\beta_{u1}(i,j)+\beta_{v1}(i,j)}\cdot\tilde{W}(3i,3j)\,, \qquad \text{(A.60)}$$

where $\overline{W}(3i,3j)=\frac{\beta_{u1}(i,j)W(3(i-1)+2,3j)+W(3i+1,3j)}{1+\beta_{u1}(i,j)}$ and

$$\tilde{W}(3i,3j)=\frac{\beta_{v1}(i,j)W(3i,3(j-1)+2)+W(3i,3j+1)}{1+\beta_{v1}(i,j)}\,.$$

In Equation A.60, $\overline{W}(3i,3j)$ is constructed along the *u* direction in the sense of Equation A.53c since the common boundary curve along the *u* direction is not only a *v*-isoparametric curve but also a Bezier curve. $\tilde{W}(3i,3j)$ is constructed along the *v* direction in the sense of Equation A.53c because the common boundary curve along the *v* direction is both a *u*-isoparametric curve and Bezier curve. The geometric meaning of Equation A.60 is shown in Figure A.10. The ratio of lengths of two line segments, *[W(3(i-1)+2,3j),* $\overline{W}(3i,3j)$*]*, and *[*$\overline{W}(3i,3j)$*,W(3i+1,3j)]*, is $1:\beta_{u1}(i,j)$. The ratio of lengths of two line segments, *[W(3i,3(j-1)+2),* $\tilde{W}(3i,3j)$*]* and *[*$\tilde{W}(3i,3j)$*,W(3i,3j+1)]*, is $1:\beta_{v1}(i,j)$. The ratio of lengths of two line segments, *[*$\overline{W}(3i,3j)$*,W(3i,3j)]* and *[W(3i,3j),* $\tilde{W}(3i,3j)$*]*, is $(1+\beta_{v1}(i,j)):(1+\beta_{u1}(i,j))$.

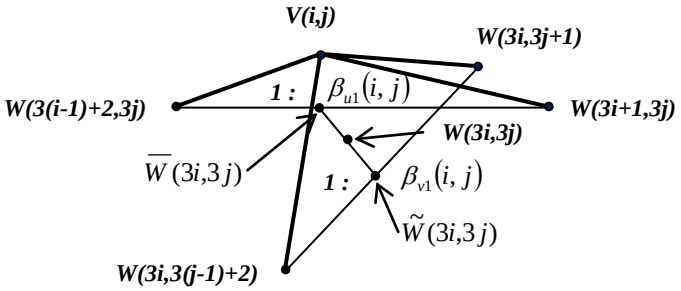


Figure A.10 Geometric Meaning of Equation A.60.

A.9 Constructions of Inside Points with PAMA

In Figure A.2, we can see four inside points in the patch $x_{i,j}(u,v)$ are *W(3i+1,3j+1), W(3i+2,3j+1), W(3i+1,3j+2),* and *W(3i+2,3j+2).* To follow the construction of the intersection (*W(3i*)) of two Bézier line segments with the method of Frain, Boehm, Barsky and DeRose (Barsky and DeRose 1989, Boehm 1985, and Farin 1982), we have to consider two issues. One issue is that each of these four points should meet the constraints of geometric continuities on *u* and *v* parameters. The other issue is that the shape parameters, $\beta_{u1}(i,j)$, $\beta_{v1}(i,j)$, $\beta_{u2}(i,j)$, and $\beta_{v2}(i,j)$, should vary along common boundary curves. Let us analyse these two issues, respectively.

If the first issue was met rigorously, the composite surface would be constructed with lower shaping freedom because it would be reduced to B-spline surfaces that hold higher parametric continuities but lower shaping freedom. We have observed that the higher shaping freedom comes from shape parameters. Thus, in the PAMA, the first issue is dealt with by blending the variations of points along both *u* and *v* directions with the method of Frain, Boehm, Barsky and DeRose (Barsky and DeRose 1989, Boehm 1985, and Farin 1982) and the bisection interpolation.

If the second issue was met seriously, we should interpolate shape parameters for points along common boundary curves with shape parameters of original points (*V(i,j)*), which would increase the multiplication computations. In the PAMA, shape parameters that are not available take approximately the values of shape parameters that are available and nearest to them in the position. For example, shape parameters of *W(3i,3j+1)* take the shape parameter values *of V(i,j),* and those of *W(3i+2,3j)* take the shape parameter values *of V(i+1,j).*

The construction equations of *W(3i+1,3j+1), W(3i+2,3j+1), W(3i+1,3j+2),* and *W(3i+2,3j+2)* with the PAMA are written as

$$W(3i+1,3j+1)=\frac{1}{2}\cdot\left(\frac{\left(1.0+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)\right)\cdot W(3i,3j+1)+\gamma_u(i,j)\cdot W(3(i+1),3j+1)}{1.0+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)}+\right.$$

$$\left.\frac{\left(1.0+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)\right)\cdot W(3i+1,3j)+\gamma_v(i,j)\cdot W(3i+1,3(j+1))}{1.0+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)}\right); \quad \text{(A.61)}$$

$$W(3i+1,3j+2)=$$

$$\frac{1}{2}\cdot\left(\frac{\left(1.0+\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)\right)\cdot W(3i,3j+2)+\gamma_u(i,j+1)\cdot W(3(i+1),3j+2)}{1.0+\gamma_u(i,j+1)+\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)}\right.$$

$$\left.+\frac{\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)\cdot W(3i+1,3j)+\left(1.0+\gamma_v(i,j)\right)\cdot W(3i+1,3(j+1))}{1.0+\gamma_v(i,j)+\beta_{v1}^2(i,j+1)\cdot\gamma_v(i,j+1)}\right); \quad \text{(A.62)}$$

$$W(3i+2,3j+1)=$$

$$\frac{1}{2}\cdot\Big(\frac{\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)\cdot W(3i,3j+1)+(1.0+\gamma_u(i,j))\cdot W(3(i+1),3j+1)}{1.0+\gamma_u(i,j)+\beta_{u1}^2(i+1,j)\cdot\gamma_u(i+1,j)}$$

$$+\frac{(1.0+\beta_{v1}^2(i+1,j+1)\cdot\gamma_v(i+1,j+1))\cdot W(3i+2,3j)+\gamma_v(i+1,j)\cdot W(3i+2,3(j+1))}{1.0+\gamma_v(i+1,j)+\beta_{v1}^2(i+1,j+1)\cdot\gamma_v(i+1,j+1)}\Big)\text{;}\quad\text{(A.63)}$$

and

$$W(3i+2,3j+2)=$$

$$\frac{1}{2}\cdot\Big(\frac{\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)\cdot W(3i,3j+2)+(1.0+\gamma_u(i,j+1))\cdot W(3(i+1),3j+2)}{1.0+\gamma_u(i,j+1)+\beta_{u1}^2(i+1,j+1)\cdot\gamma_u(i+1,j+1)}$$

$$+\frac{\beta_{v1}^2(i+1,j+1)\cdot\gamma_v(i+1,j+1)\cdot W(3i+2,3j)+(1.0+\gamma_v(i+1,j))\cdot W(3i+2,3(j+1))}{1.0+\gamma_v(i+1,j)+\beta_{v1}^2(i+1,j+1)\cdot\gamma_v(i+1,j+1)}\Big)\text{.}\quad\text{(A.64)}$$

Equations A.61, A.62, A.63, and A.64 are also Equations 7.12, 7.13, 7.14, and 7.15 in Chapter 7, respectively.

A.10 Summarising PAMA's Continuities

Inside a bi-cubic Bézier patch, $x_{i,j}(u,v)$, it is naturally C^2 continuous. Thus, we just focus on the continuities of points on boundaries, which join different patches. Figure A.4 illustrates these neighbouring patches and boundary curves.

A.10.1 For G^0

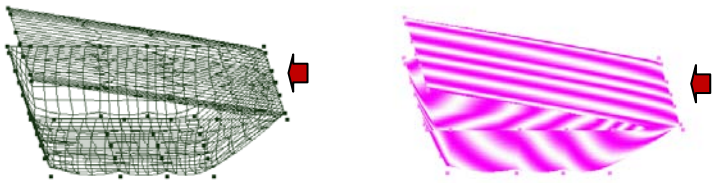
According to Equation A.3a, along a common boundary curve between two joined patches, these two patches agree with each other. Thus, the condition of C^0 continuity (and also G^0) is met for the global composite surface stitched with the PAMA.

A.10.2 For G^1

According to the analysis in Section A.6, the PAMA does not exert the sufficient condition of the G^1 at all the points along common boundary curves. In this way, it brings the following advantages,

- Four more degrees of freedom are added to the shape variation of surfaces for design purposes by shape parameters, $\beta_{u1}(i,j)$, $\beta_{v1}(i,j)$, $\beta_{u2}(i,j)$, and $\beta_{v2}(i,j)$.
- More types of shapes remain, besides the shapes that the continuous tangent planes along common boundary curves can provide. For example, the sharp change across two patches is retained, which can form folds along common boundary curves. In Figure A.11, the connection side of the clamshell box (marked with a red arrow) is constructed in this way. Figures A.11a and A.11b show the

wire-frame and filled-area views of the same box, respectively. The sharp folds cannot be provided by the G^1 surfaces.



(a) (b)

Figure A.11 The Sharp Fold along the Connection Side of (a) Wire-frame View and (b) Filled-area View of a Clamshell Box, Marked with Red Arrows.

A.10.3 For G^2

Following the discussion of Sections A.7-A.9, with the scrupulous construction scheme of the PAMA, the composite surfaces are maintained in a sense of the approximate G^2 along the *u*- and *v*-isoparametric curves. The benefits are that the effects of changing four shape parameters, $\beta_{u1}(i,j)$, $\beta_{v1}(i,j)$, $\beta_{u2}(i,j)$, and $\beta_{v2}(i,j)$, independently are distinguishable, which have been introduced in Section 7.4 of Chapter 7. Even better, they are analogous to their corresponding ones in curve cases. That is, $\beta_{u1}(i,j)$ and $\beta_{v1}(i,j)$ have the skewing effect (that is called 'bias' in Barsky 1984), and $\beta_{u2}(i,j)$, and $\beta_{v2}(i,j)$ have the tenseness effect (that is called 'tension' in Barsky 1984). In addition, these effects are orientated towards *u* or *v* direction, respectively. These are useful for the design purpose.

A.10.4 For C^1

We have known in Section A.1 that for C^1, the first partial derivatives agree along and across the common boundary curve between two neighbouring Bézier patches (Hoschek and Laser 1993).

With Properties 3, 4, and 5, it is straightforward to prove that the first partial derivatives agree naturally along a common boundary curve between two adjoining Bézier patches because the common boundary curve is a Bézier curve. They are also met by the PAMA because the PAMA is used to construct the control net for Bézier patches.

If expressed with formulas, the condition of the first partial derivatives of two adjoining patches agree across the common curves are written as the following equations. Referring to Figure A.4, along the *u*-isoparametric curve, $x_{i,j}(0,v)$, it is

$$\frac{\partial}{\partial u} x_{i-1,j}(1,v) = \frac{\partial}{\partial u} x_{i,j}(0,v); \tag{A.65a}$$

along the *v*-isoparametric curve, $x_{i,j}(u,0)$, it is

$$\frac{\partial}{\partial v} x_{i,j-1}(u,1) = \frac{\partial}{\partial v} x_{i,j}(u,0). \tag{A.65b}$$

The PAMA does not place the constraints of the first partial derivatives agreeing across common boundary curves on control points.

These are the summary for the PAMA's continuities.

References

Abbasinejad, F., Joshi, P., Grimm, C., Amenta, N., and Simons, L. (2013) Surface patches for 3D sketching. *Proceedings of the 2013 International Symposium on Sketch-Based Interfaces and Modeling, SBIM'13*, pp 53-60.

Altera Corporation (2008a) *User Guide for Altera Embedded Systems Development Kit, Cyclone III Edition*. http://www.altera.com. Site accessed on 16 Feb 2013.

Altera Corporation (2008b) *AN 527: Implementing an LCD Controller*. http://www.altera.com. Site accessed on 16 Feb 2013.

Altera Corporation (2012) *Cyclone III Device Handbook Volume 1-2*. http://www.altera.com. Site accessed on 16 Feb 2013.

Altera Corporation (2013) *ACEX 1K and FLEX 10K Data sheets*. http://www.altera.com. Site accessed on 16 Feb 2013.

Amdahl, G. (1967) Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, *AFIPS'67, ACM*, pp 483–485.

Angle, E. and Shreiner, D. (2008) An interactive introduction to OpenGL and OpenGL ES programming. *ACM SIGGRAPH Asia 2008 Courses*, Article No 2.

Arden, W.M. (2002) The international technology roadmap for semiconductors – perspectives and challenges for the next 15 years. *Current Opinion in Solid State and Materials Science*, 6 (5) 371-377.

Awad, M. (2009) FPGA supercomputing platforms: a survey. *Proceedings of International Conference on the 2009 Field Programmable Logic and Applications, FPL2009*, pp 564-568.

Baladron, J., Fasoli, D., and Faugeras, O.D. (2012) Three applications of GPU computing in neuroscience. *Computing in Science & Engineering*, 14 (3) 40-47.

Barr, A.H. (1984) Global and local deformations of solid primitives. *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'84,* 18 (3) 21-30.

Barsky, B.A. (1984) A description and evaluation of various 3-D models. *IEEE Computer Graphics and Applications*, 4 (1) 38-52.

Barsky, B.A. and DeRose, T.D. (1989) Geometric continuity of parametric curves: Three equivalent characterizations. *IEEE Computer Graphics and Applications*, 9 (6) 60-68.

Barsky, B.A. and DeRose, T.D. (1990) Geometric continuity of parametric curves: Constructions of geometrically continuous splines. *IEEE Computer Graphics and Applications*, 10 (1) 60-68.

Berekovic, M., Pirsch, P., Selinger, T., Wels, K.-I.I., Miro, C., Lafage, A., Heer C., and Ghigo, G. (2000) Co-processor architecture for MPEG-4 main profile visual compositing. *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems, ISCAS 2000*, 2, pp 180-183.

Bernstein, A. J. (1966) Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15 (5) 757–763.

Bliss, F.W. (1980) Interactive computer graphics at Fort Motor Company. *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'80*, pp 218-224.

Boehm, W. (1985) Curvature continuous curves and surfaces. *Computer Aided Geometric Design*, 2 (4) 313-323.

Bolz, J. and Schroder, P. (2002) Rapid evaluation of Catmull-Clark subdivision surfaces. *Proceedings of the Seventh International Conference on 3D Web Technology, Web3D'02*, pp 11-17.

Bonnel, N. and Marteau, P.-F. (2012) LNA: fast protein structural comparison using a Laplacian characterization of Tertiary structure. *IEEE/ACM Transaction on Computational Biology and Bioinformatics,* 9 (5) 1451-1458.

Botsch, M. and Sorkine, O. (2008) On linear variational surface deformation methods. *IEEE Transactions on Visualization and Computer Graphics*, 14 (1) 213-230.

Carpinelli, J. D (2002) *Computer systems organization and architecture*. People's Posts and Telecommunications Publishing House, Beijing, China.

Catmull, E. and Clark, J. (1978) Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10 (6) 350-355.

Cevik, U. (2004) Design and implementation of an FPGA-based parallel graphics renderer for displaying CSG surfaces and volumes. *Computers and Electrical Engineering*, 20 97-117.

Chelton, W.N. and Benaissa, M. (2008) Fast elliptic curve cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16 (2) 198-205.

Chen, T.-H., Lu, S.-Y., Lin, C.-M., Hsu, W.-K., and Fang, W. (2008) Carbon nanotube arrays on flexible substrate and their field emission characteristics. *Proceedings of the 2008 IEEE 21st International Conference on Micro Electro Mechanical Systems, MEMS 2008*, pp 697-700.

Cheng, F. and Goshtasby, A. (1989) A parallel B-spline surface fitting algorithm. *ACM Transactions on Graphics*, 8 (1) 41-50.

Choe, J.W., Nikoozadeh, A., Oralkan, O., and Khuri-Yakub, B.T. (2013) GPU-based real-time volumetric ultrasound image reconstruction for a ring array. *IEEE Transactions on Medical Imaging,* 32(7) 1258-1264.

Chrysos, G., Sotiriades, E., Rousopoulos, C., Dollas, A., Papadopoulos, A., Kirmitzoglou, I., Promponas, V., Theocharides, T., Petihakis, G., Lagnel, J., Vavylis, P., and Kotoulas, G. (2012) Opportunities from the use of FPGAs as platforms for bioinformatics algorithms. *Proceedings of the 2012 IEEE 12th International Conference on Bioinformatics & Bioengineering, BIBE 2012*, pp 559-565.

Chua, C. and Neumann, U. (2000) Hardware-accelerated free-form deformation. *Proceedings of the 2000 ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, HWWS 2000*, pp 33-39.

Ciletti, M.D. (2004) *Advanced Digital Design with the Verilog HDL*. Publishing House of Electronics Industry, Beijing, China.

Clack, B. and Keyser, J. (2013) Physical simulation of an embedded surface mesh involving deformation and fracture. *Proceedings of the 2013 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D'13*, pp 189.

Constantinides, G.A. and Nicolici, N. (2011) Guest editors' introduction: surveying the landscape of FPGA accelerator research. *IEEE Design & Test of Computers*, 28 (4) 6-7.

Cunningham, S. (2008) *Computer Graphics Programming in OpenGL for Visual Communication*. China Machine Press, Beijing.

Daga, M., Feng, W., and Scogland, T. (2011) Towards accelerating molecular modeling via multi-scale approximation on a GPU. *Proceedings of the 2011 IEEE 1st International Conference on Computational Advances in Bio and Medical Sciences, ICCABS*, pp 75-80.

DeGroat, J.E., Reehal, G., and Nagarjuna, S. (2008) Synthesizing FPGA digital modules for software defined radio. *Proceedings of the 2008 IEEE National Aerospace and Electronics Conference, NAECON 2008*, pp 358-362.

Deng, C. and Ma W. (2013) A unified interpolatory subdivision scheme for quadrilateral meshes. *ACM Transactions on Graphics*, 32 (3) Article No. 23.

Dewitt, D. and Gray, J. (1992) Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35 (6) 85-98.

Doo, D.W.H. and Sabin, M.A. (1978) Behaviour of recursive subdivision surfaces near extraordinary points. *Computer-Aided Design*, 10 (6) 356-360.

Dubois, J. and Mattavelli, M. (2003) Embedded co-processor architecture for CMOS based image acquisition. *Proceedings of the 2003 IEEE International Conference on Image Processing, ICIP 2003*, 3, pp II-591-594.

Dubuluowen, B.A., Roweikefu, C.N., and Fumingke, A.T. (2006) *Modern Geometry – Methods and Applications, Part I: The Geometry of Surfaces, Transformation Groups and Fields*. High Education Press, Beijing, China.

Duchamp, T. and Stuetzle, W. (2003) Spline smoothing on surfaces. *Journal of Computational and Graphical Statistics*, 12 (2) 354-381.

Efremov, A., Havran, V., and Seidel H. (2005) Robust and numerically stable Bézier clipping method for ray tracing NURBS surfaces. *Proceedings of the 21st Spring Conference on Computer Graphics, SCCG'05*, pp 127-135.

Egashira, A., Satoh, S., Irie, H., and Yoshinaga, T. (2012) Parallel numerical simulation of visual neurons for analysis of optical illusion. *Proceedings of 2012 Third International Conference on Networking and Computing, ICNC*, pp 130-136.

El-Ghazawi, T.A., El-Araby, E., Huang, M., Gaj, K., Kindratenko, V., and Buell, D. (2008) The promise of high-performance reconfigurable computing. *IEEE Computer*, 41 (2) 78-85.

Farin, G. (1982) Visually C2 cubic splines. *Computer Aided Design,* 14 (3) 137-139.

Farin, G. (1993) *Curves and Surfaces for Computer Aided Geometric Design, a Practical Guide*, *3rd Ed*. Academic Press, Inc. New York.

Ferguson, R.S. (2001) *Practical algorithms for 3D computer graphics*. A K Peters, Ltd. Natick, Massachusetts.

Forsey, D.R. and Bartels, R.H. (1988) Hierarchical B-spline refinement. *ACM SIGGRAPH Computer Graphics*, 22 (4) 205-212.

Franchini, S., Gentile, A., Sorbello, F., Vassallo, G., and Vitabile, S. (2008) An FPGA Implementation of a quadruple-based multiplier for 4D Clifford algebra. *Proceedings of the 11th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pp 743-751.

Frisvad, J.R., Christensen, N.J., and Falster, P. (2007) The Aristotelian rainbow: from philosophy to computer graphics. *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, GRAPHITE'07*, pp 119-128.

Gao, L. and Long, T. (2008) Spaceborne digital signal processing system design based on FPGA. *Proceedings of the 2008 Congress on Image and Signal Processing, CISP'08*, pp 577-581.

Ghasemzadeh, H., Ostadabbas, S., Guenterberg, E., and Pantelopoulos, A. (2013) Wireless Medical-embedded systems: a review of signal-processing techniques for classification. *IEEE Sensors Journal,* 13 (2) 423-437.

Gorski, P., Golatowski, F., Behnke, R., Fabian, C., Thurow, K., and Timmermann, D. (2010) Wireless sensor networks in life science applications. *Proceedings of 2010 3rd Conference on Human System Interactions, HSI*, pp 594-598.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003) *Introduction to Parallel Computing, 2nd Ed*. China Machine Press, Beijing.

Green, P.N. and Edwards, M.D. (2000) Object oriented development method for reconfigurable embedded systems. *IEE Proceedings – Computers and Digital Techniques*, 147 (3) 153-158.

Gustafson, J.L. (1988). Reevaluating Amdahl's Law. *Communications of the ACM*, 31 (5) 532–533.

Guthe, M., Balazs, A., and Klein, R. (2005) GPU-based trimming and tessellation of NURBS and T-spline surfaces. *Proceedings of the 32nd International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2005*, pp 1016-1023.

Hagen, H. (1986) Bézier-curves with curvature and torsion continuity. *Journal of Mathematics,* 16 (3) 629-638.

Hartman, N.W., Connolly, P.E., Gilger, J.W., Bertoline, G.R., and Heisler, J. (2006) Virtual reality-based spatial skills assessment and its role in computer graphics education. *Proceedings of SIGGRAPH'06 Educators Program*, Article No. 46.

Hearn, D., Baker, M.P., and Carithers W.R. (2011) *Computer Graphics with OpenGL, 4th Ed*. Pearson Education, Upper Saddle River, New Jersey.

Henzinger, T.A. and Sifakis, J. (2007) The discipline of embedded systems design. *Computer*, 40 (10) 32-40.

Hohmeyer, M.E. and Barsky, B.A. (1989) Rational continuity: parametric, geometric, and Frenet Frame continuity of rational curves. *ACM Transactions on Graphics*, 8 (4) 335-359.

Hoppe, H., DeRose, T., Duchamp, T., Halstead, M., Jin, H., Mcdonald, J., Schweitzer, J., and Stuetzle, W. (1994) Piecewise smooth surface reconstruction. *Proceedings of the 21nd International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH' 94,* pp 295-302.

Hoschek, J. and Lasser, D. (1993) *Fundamentals of Computer Aided Geometric Design*. A K Peters, Ltd. Wellesley, Massachusetts.

House, D.H. (1996) Overview of three-dimensional computer graphics. *ACM Computing Surveys*, *CSUR*, 28 (1) 145-148.

Huang, J., Shi, X., Liu, X., Zhou, K., Wei, L., Teng, S., Bao, H., Guo, B., and Shum, H. (2006) Subspace gradient domain mesh deformation. *ACM Transactions on Graphics,* 25 (3) 1126-1134.

Huang, M., Serres, O., El-Ghazawi, T.A., and Newby, G.B. (2009) Parameterized hardware design on reconfigurable computers: an image registration case study. *Proceedings of the 2009 5th Southern Conference on Programmable Logic, SPL'09*, pp 71-76.

Jiang, G., Chen, H., and Yoshihira, K. (2006) Discovering likely invariants of distributed transaction systems for autonomic system management. *Proceedings of IEEE International Conference on Autonomic Computing, ICAC'06*, pp 199-208.

Joe, B. (1990) Knot insertion for Beta-spline curves and surfaces. *ACM Transactions on Graphics*, 9 (1) 41-45.

Jonker, P. and Vogelbruch, J. (1997) The CC/IPP, and MIMD-SIMD architecture for image processing and pattern recognition. *Proceedings of 1997 Fourth IEEE International Workshop of Computer Architecture for Machine Perception, CAMP97*, pp 33-39.

Kahng, A.B. (2013) Product futures. *IEEE Design & Test of Computers*, 28 (6) 88-89.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D.J. (2008) H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDA Endowment*, 1 (2) 1496-1499.

Kalomiros, J.A. and Lygouras, J. (2007) A host/co-processor FPGA-based architecture for fast image processing. *Proceedings of IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Dortmund, Germany, September, 2007*, pp 373-378.

Karol, G., Aleksandra, F., Agata, J., and Andrzej, R. (2011) Parallel simulation of stochastic denritic neurons using NVidia GPUs with CUDA C. *Proceedings of the 2011 18th International Conference of Mixed Design of Integrated Circuits and Systems, MIXDES*, pp 614-617.

Kasik, V. and Chvostkova, Z. (2013) FPGA in technical resources of medical imaging. *Proceedings of 2013 IEEE 11th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2013*, pp 193-196.

Kazakov, M. (2007) Catmull-Clark subdivision for geometry shaders. *Proceedings of the 2007 ACM 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, AFRIGRAPH'07,* pp 77-84.

Kelly, M. and Hsu, K. W. (1998) A flexible pipelined image processor. *Proceedings of the 1998 Eleventh Annual IEEE International ASIC Conference*, pp 325-332.

KHRONOS Group, Connecting Software to Silicon (2013) *OpenGL ES Standard*. http://www.khronos.org/opengles/. Site accessed on 16 Feb 2013.

Kilgard, M. (1997) Realizing OpenGL: two implementations of one architecture. *Proceedings of 1997 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware,* pp 45-55.

Kilgard, M. J. and Akeley, K. (2008) Modern OpenGL: its design and evolution. *The 1st ACM SIGGRAPH Conference and Exhibition in Asia, SIGGROPH Asia 2008 Courses*, Article No. 13.

Kim, H., Park, J., Yoon, J., Kim, S., and Kim, L. (2012) A 1mJ/frame unified media application processor with a 179.7pJ mixed-mode feature extraction engine for embedded 3D-media contents processing. *Proceedings in 2012 IEEE Custom Integrated Circuits Conference, CICC*, pp 1-4.

Kim, J.K., Fessler, J.A. and Zhang, Z. (2012) Forward-projection architecture for fast iterative image reconstruction in X-ray CT. *IEEE Transactions on Signal Processing*, 60 (10) 5508-5518.

Konrad, S., Cheng, B.H.C., and Campbell, L.A. (2004) Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30 (12) 970-992.

Krishnakumar, Y., Prasad, T.D., Kumar, K.V.S., Raju, P., and Kiranmai, B. (2011) Realization of a parallel operating SIMD-MIMD architecture for image processing application. *Proceedings of IEEE 2011 International Conference on Computer, Communication and Electrical Technology*, *ICCCET.2011*, pp 98-102.

Krone, M., Bidmon, K., and Ertl, T. (2009) Interactive visualization of molecular surface dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 15 (6) 1391-1398.

Kuehne, B., True, T., Commike, A., and Shreiner, D. (2005) Performance OpenGL: platform independent techniques or "a bunch of good habits that will help the performance of any OpenGL program". *The 32nd International Conference on Computer Graphics and Interactive Techniques, SIGGROPH 2005 Courses*, Article No. 1.

Lampe, O.D., Viola, I., Reuter, N., and Hauser, H. (2007) Two-level approach to efficient visualization of protein dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 13 (6) 1616-1623.

Lawrence, J. and Funkhouser, T. (2003) A painting interface for interactive surface deformations. *Proceedings of 11th Pacific Conference on Computer Graphics and Applications*, pp 141-150.

Leon, S.J. (2007) *Linear Algebra with Applications*. China Machine Press, Beijing, China.

Lin, S., You, F., Luo, X., and Li, Z. (2008) Deducing interpolation subdivision schemes from approximating subdivision schemes. *SIGGRAPH Asia 2008, ACM Transactions on Graphics*, 27 (5) Article 146.

Li, X., Liu, B., and Wu, E. (2006) Double projective cylindrical texture mapping on FPGA. *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications, VRCIA'06,* pp 91-97.

Li, Z., Ma, L., and Tan, W. (2006) Three-dimensional object reconstruction from contour lines. *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications, VRCIA'06,* pp 319-322.

Liu, J.-H., Chen, J., Tsai, Y.-C., Tai, Y.-C., and Shih, C.-H. (2012) Supporting audio streaming in application cloud for embedded systems. *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCC-ICESS,* pp 1800-1805.

Liu, X., Xu, W., Guan, Y., and Shang, Y. (2009) Trigonometic polynomial uniform B-spline surface with shape parameter. *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, ICIS'09,* pp 1357-1363.

Liu, Y.-K., Yue, Y., Maple, C., Tang, H., and Guo, M. (2009) Comparison between two solutions on fast Fourier Transform Algorithm: software and hardware. *Proceedings of the 15th International Conference on Automation and Computing, ICAC'09, Luton, the U.K.*, pp 98-103.

Loop, C.T. and DeRose, T.D. (1989) A multisided generalization of Bézier surfaces. *ACM Transactions on Graphics*, 8 (3) 204-234.

Loop, C. and DeRose, T. (1990) Generalized B-spline surfaces of arbitrary topology. *ACM SIGGRAPH Computer Graphics*, 24 (4) 347-356.

Loop, C. (1994) Smooth spline surfaces over irregular meshes. *Proceedings of the 21nd International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH' 94,* pp 303-310.

Lopez-Ongil, C., Garcia-Valderas, M., Portela-Garcia, M., and Entrena-Arrontes, L. (2005) An autonomous FPGA-based emulation system for fast fault tolerant evaluation. *Proceedings of 2005 International Conference on Field Programmable Logic and Applications*, pp 379-402.

Losh, E. (2006) Making things public: democracy and government-funded videogames and virtual reality simulations. *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, pp 123-132.

Luebke, D. and Humphreys, G. (2007) How GPUs work. *Computer*, 40 (2) 96-100.

MacLean, W.J. (2005) An evaluation of the suitability of FPGAs for embedded vision systems. *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR'05*, pp 131-137.

Martin, T., Cohen E., and Kirby M. (2008) Volumetric parameterization and trivariate b-spline fitting using harmonic functions. *Proceedings of the 2008 ACM Symposium on Solid and Physical Modelling, SPM'08,* pp 269-280.

Martinez, P. and Chalmers, A. (2004) Using computer graphics in archaeology: a struggle for educative science or to educate science? *Proceedings of SIGGRAPH'04 Educators Program*, pp 35.

Massey, T., Dabiri, F., Jafari, R., Noshadi, H., Brisk, P., Kaiser, W., and Sarrafzadeh, M. (2007) Towards reconfigurable embedded medical systems. *Proceedings of 2007 Joint Workshop on High Confidence Medical Devices software, and Systems and Medical Device Plug-and-Play Interoperability, HCMDSS-MDPnP 2007*, pp 178-180.

Melnikova, O., Hahanova, I., and Mostovaya, K. (2009) Using multi-FPGA systems for ASIC prototyping. *Proceedings of the 2009 10th International Conference – The Experience of Designing and Applications of CAD Systems in Microelectronics, CADSM 2009,* pp 237-239.

Mohan, C., Lindsay, B., and Obermarck, R. (1986) Transaction management in the R* distributed database management system. *Transactions on Database Systems*, 11 (4) 378-396.

Monmasson, E. and Cirstea, M.N. (2007) FPGA design methodology for industrial control systems – a review. *IEEE Transactions on Industrial Electronics*, 54 (4) 1824-1842.

Müller, K., Fünfzig, C., Reusche, L., Hansford, D., Farin, G., and Hagen H. (2010) Dinus: double insertion, nonuniform, stationary subdivision surfaces. *ACM Transactions on Graphics*, 29 (3) Article 25.

Müller, K., Reusche, L., and Fellner, D. (2006) Extended subdivision surfaces: building a bridge between NURBS and Catmull-Clark surfaces. *ACM Transactions on Graphics*, 25 (2) 268-292.

Nagendra, C., Owens, R.M., and Irwin, M.J. (1993) Digit systolic algorithms for fine-grain architectures. *Proceedings of the 1993 International Conference on Application-Specific Array Processors,* pp 466-477.

Nahum, D.G. (1996) How can SIGGRAPH be more effective in promoting computer graphics? (panel). *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'96,* pp 497-498.

Nasri, A.H. (1987) Polyhedral subdivision methods for free-form surfaces. *ACM Transactions on Graphics*, 6 (1) 29-73.

Nasri, A.H. and Abbas, A.M. (2002) Lofted Catmull-Clark subdivision surfaces. *Proceedings of the 2002 IEEE Geometric Modelling and Processing – Theory and Applications*, pp 83-93.

National Instuments Corporation (2007) *LabVIEW System Identification Toolkit*. http://www.ni.com/white-paper/3584/en. Site accessed on 16 Feb 2013.

Nickolls, J. and Dally, W.J. (2010) The GPU computing era. *Micro, IEEE,* 30 (2) 56-69.

Nomoto, S., Kyo, S., and Okazaki, S. (2011) A dynamic SIMD/MIMD mode switching processor for embedded real-time image recognition systems. *Proceedings of the 2011 IEEE Asian Solid State Circuits Conference, A-SSCC*, pp 17-20.

Owens, J.D., Huston, M., Luebke, D., Green, S., Stone, J.E., and Phillips, J.C. (2008) GPU computing. *Proceedings of the IEEE*, 96 (5) 879-899.

Pan, Z., Heng, P., and Lau, R.W.H. (2000) Computer graphics around the world: computer graphics in Hong Kong, *ACM SIGGRAPH Computer Graphics*, 34 (1) 15-19.

Patashev, T., Abla, G., and Govind, N. (2000) Simulation of hardware support for OpenGL graphics architecture. *Proceedings of the 2000 International Conference on Information Technology: Coding and Computing*, pp 295.

Patney, A., Ebeida, M.S., and Owens, J.D. (2009) Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. *Proceedings of the 2009 Conference on High Performance Graphics, HPG'09*, pp 99-108.

Paul, B. (2013), *The Mesa 3D Graphics Library*. http://www.mesa3d.org/intro.html; ftp://ftp.freedesktop.org/pub/mesa/. Site accessed on 24 Feb 2013.

Perez, P., Gangnet, M., and Blake, A. (2003) Poisson image editing. *Proceedings of the 30th International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2003*, pp 313-318.

Pitas, I. (1993) *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*. Wiley, Chichester, West Sussex, England.

Qasim, S.M., Abbasi, S.A, and Almashary, B. (2009) A review of FPGA-based design methodology and optimization techniques for efficient hardware realization of computation intensive algorithms. *Proceedings of International Conference of Multimedia, Signal Processing and Communication Technologies, 2009, IMPACT'09*, pp 313-316.

Quarteroni, A., Sacco, R., and Saleri, F. (2006) *Numerical Mathematics*. Science Press, Beijing, China.

Ramasubramanian, N., Subramanian, R., and Pande, S. (2002) Automatic compilation of loops to exploit operator parallelism on configurable arithmetic logic units. *IEEE Transactions on Parallel and Distributed Systems*, 13 (1) 45-66.

Repplinger, M., Löffler, A., Schug, B., and Slusallek, P. (2009) Extending X3D for distributed multimedia processing and control. *Proceedings of the 14th International Conference on 3D Web Technology, Web3D'09,* pp 61-69.

Rhyne, T.-M. (2008) Visualization and the larger world of computer graphics. *Proceedings of the 35th International Conference and Exhibition on Computer Graphics and Interactive Techniques, SIGGRAPH 2008 Classes*, Article No. 97.

Saddem, R., Toguyéni, A.K.A., and Tagina, M.M. (2011) Diagnosis of critical embedded systems: application to the control card of a railway vehicle braking systems. *Proceedings of the 2011 IEEE Conference on Automation Science and Engineering, CASE,* pp 163-168.

Sathyanarayana, K. and Kumar, G.V.V. (2008) Evolution of computer graphics and its impact on engineering product development. *Proceedings of the Fifth International Conference on Computer Graphics, Imaging and Visualisation, CGIV'08*, pp 32-37.

Schröder, S., Peterson, J.A., Obermaier, H., Kellogg, L.H., Joy, K.I., and Hagen, H. (2012) Visualization of flow behaviour in earth mantle convection. *IEEE Transactions on Visualization and Computer Graphics*, 18 (12) 2198-2207.

Sederberg, T.W. and Parry, S.R. (1986) Free-form deformation of solid geometric models. *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH'86*, 20 (4) pp 151-160.

Sederberg, T.W., Zheng, J., Sewell, D., and Sabin, M. (1998) Non-uniform recursive subdivision surfaces. *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH' 98,* pp 387-394.

Sederberg, T.W., Zheng, J., Bakenov, A., and Nasri, A. (2003) T-splines and T-NURCCs. *ACM Transactions on Graphics*, 22 (3) 477-484.

Si, W. and Guenter B. (2010) Linear-time dynamics for multibody systems with general joint models. *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation SCA'10*, pp 31-37, pp 228.

Siegel, L. J., Kemmerer, F. C., Mueller, P. T. Jr., Smalley, H. E. Jr., and Smith, S. D. (1981) PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, 30 (12) 934-947.

Smith, G.L. and De La Torre, L. (2006) Techniques to enable FPGA based reconfigurable fault tolerant space computing. *Proceedings of the 2006 IEEE Aerospace Conference*, 10.1109/AERO.2006.1655958.

Sorkine, O., Cohen-Or, D., Lipman, Y., Alexa, M., Rossl, C., and Seidel, H.-P. (2004) Laplacian surface editing. *Proceedings of the 2004 Eurographics Symposium on Geometry Processing*, pp 175-184.

Sridharan, K. and Priya, T. K. (2004) A parallel algorithm for constructing reduced visibility graph and its FPGA implementation. *Journal of Systems Architecture*, 50, 635-644.

Sumner, R.W., Schmid, J., and Pauly, M. (2007) Embedded deformation for shape manipulation. *ACM Transactions on Graphics*, 26 (3) Article 80.

Surducan, V., Surducan, E., Ciupa, R.V., and Roman, M.N. (2010) Embedded system controlling microwave generators in hyperthermia and diathermy medical devices. *Proceedings of 2010 IEEE International Conference on Automation Quality and Testing Robotics, AQTR 2010*, pp 1-6.

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., ans Abadi, D.J. (2012) Calvin: fast distributed transactions for partitioned database systems. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD'12*, pp 1-12.

True, T., Grantham, B., Kuehne, B., and Shreiner, D. (2004) Performance OpenGL: platform independent techniques. *Proceedings of the 29th International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2004*, Course 16.

Turqueti, M.A., Saniie, J., and Oruklu, E. (2010) MEMS acoustic array embedded in an FPGA based data acquisition and signal processing system. *Proceedings of the 2010 53rd IEEE International Midwest Symposium on Circuits and Systems, MWSCAS 2010,* pp 1161-1164.

Underwood, K. (2004) FPGAs vs. CPUs: trends in peak floating-point performance*. Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp 171-180.

Van den Bout, D.E., Morris, J.N., Thomae, D.A., Labrozzi, S., Wingo, S., and Hallman, D. (1992) AnyBoard: an FPGA-based, reconfigurable system. *IEEE Design & Test of Computers*, 9 (3) 21-30.

Varghese, R.R. and Tharayil, C.G. (2001) *Design, Simulation and Synthesis at an FFT Processor Using VHDL*. Project report of Mar Athanasius College of Engineering, Mahatma Gandhi University, Kottayam, Kerala.

Vuduc, R.W. and Czechowski, K. (2011) What GPU computing means for high-end systems. *Micro, IEEE*, 31 (4) 74-78.

Wang, W., Pottmann, H., and Liu, Y. (2006) Fitting B-spline curves to point clouds by curvature-based squared distance minimization. *ACM Transactions on Graphics*, 25 (2) 214–238.

Wang, X. and Ziavras, S.G. (2006) Exploiting mixed-mode parallelism for matrix operations n the HERA architecture through reconfiguration. *Computers and Digital Techniques, IEE Proceedings*, 153 (4) 249-260.

Welch, W. and Witkin, A. (1994) Free-form shape design using triangulated surfaces. *Proceedings of the 21rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'94,* pp 247-256.

Weyrich, T., Heinzle, S., Aila, T., Fasnacht, D.B., Oetiker, S., and Botsch, M. (2007) A hardware architecture for surface splatting. *ACM Transactions on Graphics*, 26 (3) Article 90.

Xiao, S., Lin, H., and Feng, W. (2011) Accelerating protein sequence search in a heterogeneous computing system. *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS,* pp 1212-1222.

Yoo, T.S., Bliss, D., Lowekamp, B.C., Chen, D.T., Murphy, G.E., Narayan, K., Hartnell, L.M., Do, T., Subramaniam, S. (2012) Visualizing cells and humans in 3D: biomedical image analysis at nanometre and meter scales. *IEEE Computer Graphics and Applications,* 32 (5) 39-49.

Yu, Y., Zhou, K., Xu, D., Shi, X., Bao, H., Guo, B., and Shum, H. (2004) Mesh editing with Poisson-based gradient field manipulation. *Proceedings of the 29th International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2004*, pp 644-651.

Yudanov, D. and Reznik, L. (2012) Scalable multi-precision simulation of spiking neural networks on GPU with OpenCL. *Proceedings of 2012 International Joint Conference on Neural Networks*, *IJCNN*, pp 1-8.

Zave, P. (1982) An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, SE-8 (3) 250-269.

List of Publications (2009 to 2013)

Papers published in proceedings of international conferences:

Liu, Y.-K. (2013) A surface model for interactive shape editing. *Proceedings of the 2013 International Conference on Computer Science and Artificial Intelligence, ICCSAI2013*, pp 287-291.

Liu, Y.-K., Guo, L.-W., Liu, J.-M., Yue, Y., Maple, C., and Crabbe M.J.C. (2013) An application-oriented top-down scheme for FPGA-based embedded system design with 3D graphics applications. *Proceedings of the 2013 International Conference on Computer Sciences and Applications, CSA2013*, pp 756-764.

Wang, C.-H., Liu, Y.-K., Guo, L.-W., Yue, Y., and Maple, C. (2011) An on-line distributed induction motor monitoring system based-on ARM and CAN bus. *Proceedings of the 6th International Symposium on Parallel Computing in Electrical Engineering, PARELEC 2011,* pp 185-188.

Zhang, Y.-Q., Liu, Y.-K., and Gao, H.-B. (2011) Study of a scenic spot monitoring system based on RFID and multi-sensor information fusion technology. *Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology, EMEIT 2011,* pp 1739-1742.

Gao, H.-B, Liu, Y.-K., and Guo, L.-W. (2011) Implementation of an embedded induction motor test and analysis system. *Proceedings of 2011 International Conference on Consumer Electronics, Communications and Network*s, *CECNet 2011*, pp 1328-1331.

Liu, Y.-K., Yue, Y., Maple, C., Tang, H.-W., and Guo, M. (2009) Comparison between two solutions on fast Fourier Transform Algorithm: software and hardware. *Proceedings of the 15th International Conference on Automation and Computing, ICAC'09,* pp 98-103.

Liu, Y.-K., Guo, L.-W., and Huang, C.-R. (2009) Implementation and verification of the Amplitude Recovery Method algorithm with the faults diagnostic system on induction motors. *Conference Record of 12th International Conference on Electrical Machines and Systems, ICEMS2009*, IEEE Catalogue Number: CFP09081-CDR, DS2G3-1.

Huang, C.-R., Gao, H.-B., Liu, Y.-K., and Pan, W.-Y. (2009) Large-Inertia Temperature Control based-On Information Fusion. *Proceedings of 2009 International Forum on Information Technology and Applications*, *IFITA2009,* v2, pp 420-423.

Journal Papers:

Liu, Y.-K., Guo, L.-W., Wang, Q.-X., An, G.-Q., Guo, M., and Lian, H. (2010) Application to induction motor faults diagnosis of the amplitude recovery method combined with FFT. *Mechanical Systems and Signal Processing*, Elsevier, 24(8), 2010, 2961-2971.

An, G.-Q., Liu, J.-M., Liu, Y.-K., and Liang, Y.-C. (2012) Diagnosis of broken rotor bar fault in squirrel-cage motor fed with variable frequency power based on correlation filtering method. *Electric Machines and Control,* 3, 2012, 47-50.

An, G.-Q., Liu, J.-M., Guo, L.-W., and Liu, Y.-K. (2011) Diagnosing rotor broken bar fault in motor by using correlation fundamental component filtering method. *Electric Machines and Control,* 3, 2011, 69-73.

Book:

Liu, Y.-K., Yue, Y., and Guo, L,-W. (2011) *UNIX Operating System: the development tutorial via UNIX kernel services.* High Education Press, Beijing, and Springer, Heidelberg, Dordrecht, London, New York.

Presentation:

Liu, Y.-K., (2013) A novel algorithm for surface modelling and morphing, The Fifth BCS Doctoral Consortium, London, the UK, 16th May 2013.

Papers under review or under preparation:

Liu, Y.-K., et al (2014) Continuities of Progressive and Mixing Algorithm for surface modelling and editing. *Proceedings of the 2014 International Conference on Progress in Informatics and Computing, PCI-2014, May 2014, Shanghai, China*.

Liu, Y.-K., et al (2014) Parallelism in embedded systems. *Microprocessors and Microsystems: Embedded Hardware Design*, Elsevier.

Liu, Y.-K., et al (2014) The improvement of Progressive and Mixing Algorithm with its construction scheme. *Computational Geometry, Theory and Applications,* Elsevier.