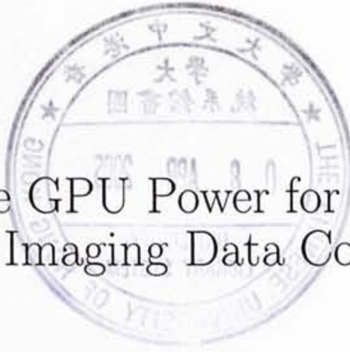

Exploiting the GPU Power for Intensive Geometric and Imaging Data Computation

Wang Jianqing

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

©The Chinese University of Hong Kong

Aug, 2004

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or the whole of the materials in this thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.

香港中文大學
大學
圖書館系統
THE CHINESE UNIVERSITY OF HONG KONG
0 8 APR 2005
UNIVERSITY
LIBRARY SYSTEM

Abstract

Computer graphics, imaging and visualization nowadays requires dealing with and performing various analysis on large amount of data. For example, deformation and skeletal animation of complex geometries in real-time, multiple-level discrete wavelet transform on high resolution imaging data, etc. These operations usually are computational intensive and impose a heavy burden on the CPU, which are also hard to achieve real-time performance. On the other hand, with the recent advances in consumer-level graphics hardware, the current new generation of graphics accelerator now consists of a graphics processing unit (GPU) which offers SIMD-based parallel processing power. It doesn't merely do the job of rendering texture-mapped polygons, but also provide us with high-precision rendering pipeline and high programmability nowadays. We can perform other general purpose computing on it when carefully designed. In this thesis, we have successfully exploited the power of GPU for computation and processing of both geometric data and imaging data in the 2 applications – real-time character facial and skeletal animation; and multiple levels of discrete wavelet transform. We develop parallel algorithms to map our solutions to the GPU SIMD architecture and leverage its power to give a great performance gain in both applications. These approaches also offload the intensive computing tasks from CPU and achieve load balancing between CPU and GPU. At the same time, for the virtual character project, we also propose a simple and efficient framework for both facial and skeletal animation, as well as the seamless integration of them for rendering. The solutions for supporting

multilingual lip synchronization and integration with text-to-speech system of multiple languages are also developed.

摘要

今天，在計算機圖形學，圖形處理與可視化的研究領域中，處理與分析大量數據是必不可少的。比如複雜三維幾何體的實時變形與骨骼動畫，或是對高分辨率的大尺寸圖像做多重小波變換。這些處理通常需要繁重與大量的計算，並且對電腦中央處理單元(CPU)的運作造成了大量的負擔，也很難實現實時的性能。在另一方面，隨著近幾年消費者級的圖形處理硬件的快速迅猛發展，新一代的圖形加速卡已經配備了一顆具有並行向量處理與多重流水綫的圖形處理單元(GPU)，由於提供了高精度的圖形流水綫與易于使用的圖形編程接口，它不僅僅只適用於圖形渲染的功能，只要經過適當的算法設計，我們可用它來實行其他更多非圖形或一般化的計算任務。我們成功探索了圖形處理單元對於大量幾何與圖像數據的潛在計算處理能力， 並提供了對於兩個此類問題的解決方案—實時虛擬角色的臉部與骨骼動畫，以及對高分辨率圖像進行快速多重小波變換。我們設計的並行算法成功將這些問題映射到 GPU 的並行架構之上並最大限度發掘它對這兩個問題的性能加速，減輕 CPU 的計算負荷與達到負載平衡。同時在虛擬角色的研究中，我們也推出了一個簡單而有效且可以同時支持臉部與骨骼動畫的系統架構，對於多語言進行唇形同步，以及與文本到語音合成系統的整合問題我們也提出了很好的解決方案。

Acknowledgments

I would like to thank my supervisors professor Pheng-Ann Heng and professor Tien-Tsin Wong, who have patiently guided me through the steps of my MPhil study. I thank them for being great advisors, teachers, and friends. They convinced me that my work of GPU computation for virtual character system and discrete wavelet transform was a good topic to devote my time for research. Without their encouragement and guidance I couldn't have achieved any of the awards or publications now. I want to express my gratitude to professor Helen Meng, her guidance in the virtual performer project, as well as her advices and encouragement are of great help to me. I would also like to thank my thesis committee, professor Hanqiu Sun and professor Kin-Hong Wong. Professor Hanqiu Sun has also been my marker for 2 times and given me many useful advices on my research work, many thanks for that.

I would like to express my gratitude to Mr. K. Yamato, the famous CG artist from Japan, for providing the beautiful character model to me for use in my project without any fee. I want to thank my great partner Simon Wong, we are a great team and had a good time working together for the virtual performer project. I am very grateful to Mark Harris and Tristan Lorach from NVIDIA as well as Cliff Wolley from University of Virginia for their great support and help in endless technical issues of GPU. I would also like to thank all the people that contribute to the GPU community at www.gpgpu.org and www.cgshaders.org, the passionate discussions and enlightening ideas are always my valuable resources and forces for research. I'm very thankful to

my colleagues here in our graphics research group in Department of Computer Science and Engineering of CUHK, we have spent 2 years of wonderful time together doing graphics research and all of you have helped me a lot. I would like to express my great gratitude to my parents and family and my girlfriend. Without your love and help, I could not have gone this far.

At last, I would like to thank all the graphics cards I've worked with, in my heart you are not just circuit boards with silicon chips, you are my friends at all times. Without you, none of my programs would run and give me beautiful graphics on the screen, which are always my greatest rewards.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Thesis 3

1.3 Contributions 4

1.4 Organization 6

2 Programmable Graphics Hardware **8**

2.1 Introduction 8

2.2 Why Use GPU? 9

2.3 Programmable Graphics Hardware Architecture 11

2.4 Previous Work on GPU Computation 15

3 Multilingual Virtual Performer **17**

3.1 Overview 17

3.2 Previous Work 18

3.3 System Overview 20

3.4 Facial Animation 22

3.4.1 Facial Animation using Face Space 23

3.4.2 Face Set Selection for Lip Synchronization 27

3.4.3 The Blending Weight Function Generation and Coarticulation 33

3.4.4 Expression Overlay 38

3.4.5 GPU Algorithm 39
3.5 Character Animation 44
3.5.1 Skeletal Animation Primer 44
3.5.2 Mathematics of Kinematics 46
3.5.3 Animating with Motion Capture Data 48
3.5.4 Skeletal Subspace Deformation 49
3.5.5 GPU Algorithm 50
3.6 Integration of Skeletal and Facial Animation 52
3.7 Result 53
3.7.1 Summary 58

4 Discrete Wavelet Transform On GPU **60**
4.1 Introduction 60
4.1.1 Previous Works 61
4.1.2 Our Solution 61
4.2 Multiresolution Analysis with Wavelets 62
4.3 Fragment Processor for Pixel Processing 64
4.4 DWT Pipeline 65
4.4.1 Convolution Versus Lifting 65
4.4.2 DWT Pipeline 67
4.5 Forward DWT 68
4.6 Inverse DWT 71
4.7 Results and Applications 73
4.7.1 Geometric Deformation in Wavelet Domain 73
4.7.2 Stylish Image Processing and Texture-illuminance Decoupling 73
4.7.3 Hardware-Accelerated JPEG2000 Encoding 75
4.8 Web Information 78

5 Conclusion **79**

Bibliography **81**

List of Figures

2.1 The graphics hardware pipeline 12
2.2 The vertex processor 13
2.3 The fragment processor 14

3.1 Virtual character system architecture 20
3.2 Face set in the high dimensional face space 24
3.3 Using deformation vector as basis to span a high dimensional face space 26
3.4 Sample input and output of CU Vocal 28
3.5 IPA visemes 31
3.6 Japanese characters with phonetic representation 32
3.7 The coarticulation between 2 adjacent phonemes 35
3.8 Variation of blending weights over time for animation 37
3.9 Live news report animation using weight function generated automatically by CU Vocal TTS system 38
3.10 Manual weight function generation using waveform analyze software 39
3.11 Facial expressions 40
3.12 Vertex shader structure for facial animation 41
3.13 Using video ram as stream data buffer 42
3.14 Skeleton hierarchy and the corresponding mesh 45
3.15 A simple 2 bone skeletal structure 47

3.16 Motion capture animation using BVH data 49
3.17 Skeletal subspace deformation 50
3.18 Pseudocode for skeletal subspace deformation on GPU 51
3.19 Skeletal subspace deformation on GPU 51
3.20 Integration of skeletal and facial animation 53
3.21 The color and alpha channel for hair 54
3.22 Cubemap for rendering the environment 55
3.23 Environment and effect rendering 56
3.24 Cartoon style rendering 57

4.1 One dimensional DWT: filtering and downsampling. 63
4.2 The 3D rendering pipeline 65
4.3 The lifting scheme 66
4.4 Separable 2D DWT 68
4.5 The indirect address table for boundary extension. 70
4.6 Mapping to the base positions and decomposition 70
4.7 Virtual upsampling and interleaving for precomputing indirect addresses. 72
4.8 Reconstruction filtering in inverse DWT. 72
4.9 Three different wavelet-based geometric designs. 73
4.10 Fast image stylishing by combining coefficients in wavelet domain. 74
4.11 Fast texture-illuminance decoupling. 75
4.12 JPEG2000 encoding flow 76
4.13 Timing Comparison: software versus GPU DWT 77

List of Tables

2.1 Features and performance of selected NVIDIA GPUs 11
2.2 Vertex engine features 12
2.3 Fragment engine features 15

3.1 Cantonese onsets mapping 30
3.2 Cantonese nuclei mappings 30
3.3 Cantonese codas mapping 31
3.4 Cantonese nasals mapping 31
3.5 Japanese mapping table 33
3.6 Mandarin mapping 34
3.7 Results of the user perception experiments. 56
3.8 Geometric and texture data statistics 57
3.9 Performance on different systems 58

4.1 Breakdown of computational time (sec) 78
4.2 Encoding quality comparison for lossy coding 78

Chapter 1

Introduction

1.1 Overview

Nowadays, various applications in computer graphics, visualization and image processing require dealing with large amount of data. The data can be generally categorized into 2 types: geometric data and imaging data. Geometric data is usually in the form of polygonal meshes or spline patches; and imaging data is usually 2D grid data with several channels. High resolution polygonal mesh and large texture images are now frequently used for rendering in game, movie industry and TV commercials. Volume Data in medical imaging like CT or MRI images are handled and processed for visualization in many applications. All the facts show that in the area of computer graphics and imaging, we are facing an increasing need for processing more and more large data sets with the computing power of our PC. In many applications that require high performance or quick response, the need for processing and performing intensive computation on large data sets as quick as possible or even in real-time continuously grows.

In applications of geometric computing, a typical example of intensive computation on large geometric data set is real-time 3D facial deformation and

kinematics calculation in virtual character animation, which requires fast deformation calculation of the 3D human face mesh and weighted skeletal transformation of all of the vertices. It remains a challenge for achieving real-time and high-quality rendering at the same time. In many previous applications, people trade performance with low quality model of less data, or have to wait for a long time to do off-line rendering of high quality animations.

When dealing with imaging data, one typical example of computational intensive task will be the discrete wavelet transform on large images or data set. The intensive computation of DWT due to multilevel filtering/downsampling does not cause a serious problem when the data scale is small. but this will become a significant bottleneck in real-time applications with large data set. Like using the new JPEG2000 standard to encode/decode high resolution images from the digital cameras today, the waiting time may be unendurably long from tens of seconds to minutes per image according to its size.

On the other hand, the recent rapid increase in the processing speed and programmability of consumer graphics hardware cast lights on our problems. With the powerful SIMD architecture and parallel pipelines of the graphics processing unit (GPU), it is possible to exploit the graphics rendering pipeline and design GPU algorithms for more general computational tasks besides just rendering.

In this thesis, we focus on exploiting the computational power of two engines inside GPU at different stages of the pipeline – the vertex processor and fragment processor for their potential applications in large geometric and imaging data processing. We achieve this goal by providing efficient solutions to the two typical problems described above – real-time high quality virtual character animation and fast discrete wavelet transform.

1.2 Thesis

Central to our research, the goal in this thesis is to perform intensive geometric and imaging data computation on the GPU using the vertex and fragment shaders, which are programs being executed by the vertex and fragment processor, respectively. The rendering pipeline is exploited to perform geometric and imaging computation tasks that are offloaded from CPU, and at the same time, to increase performance with the SIMD and parallel processing power of GPU.

We illustrate how to perform large amount of geometric computation on GPU by providing solutions for the virtual character animation problem. This is a joint project between Department of Computer Science and Engineering (CSE) and Department of System Engineering and Engineering Management (SEEM). We cooperate with Dr. Helen Meng from Human Computer Communication Laboratory (HCCL). An efficient framework for virtual character animation suitable for GPU computation is developed. This enables us to perform facial and skeletal animation computation totally on the GPU, thus achieve high quality rendering as well as maintain real-time performance with the help of mapping the large amount of geometric data computation onto the vertex processor of GPU. Besides the performance issue, the language capability of the virtual character for lip synchronization is also explored. In order for the character to be multilingual and speak or sing in different languages, an International Phonetic Alphabet (IPA)-based mapping technique is developed and could be used to extend the language capability of the virtual character to many different languages like Mandarin, Cantonese, English, Japanese, and much more. The home-brew cantonese text-to-speech system – CU Vocal developed by the HCCL of SEEM department is also integrated into our system, that enables us to generate cantonese lip-synced cantonese speech animation in real-time just from Chinese texts.

Fast processing of large imaging data set on GPU is also possible. We illustrate this by providing solution to accelerate the slow multi-level DWT process. We successfully map the whole transform onto the GPU using the fragment shader engine. Indirect addressing / dependent texture read technique is employed to effectively implement the DWT on GPU. Although the forward and inverse wavelet transforms are mathematically different, our proposed algorithm unifies them to an almost identical process that can be efficiently implemented on GPU. Different wavelets kernels and boundary extension schemes can be easily incorporated by simply modifying input parameters. To demonstrate its applicability and performance, we apply it to wavelet-based geometric design, stylish image processing, texture-illuminance decoupling, and JPEG2000 image coding.

1.3 Contributions

The major contribution of this thesis is the techniques and approaches developed by us to map intensive geometric and imaging data computation to the vertex and fragment engines of the GPU, and how to make use of its SIMD and parallel pipeline power to achieve high system performance.

The virtual character system provides a good solution for performing geometric deformation for animation computing on GPU. At the same time, it also presents a simple and efficient framework for both facial and skeletal animation. The multilingual lip synchronization capabilities, as well as the support for real-time generation of lip-synchronized animation with text-to-speech systems are also developed.

And for imaging data processing on the GPU, we accelerate the discrete wavelet transform with the fragment processor using indirect addressing or dependent texture techniques. Our approach not only unifies both forward and inverse transform into an almost same process, different wavelets kernels

and boundary extension schemes can be also be easily incorporated.

Our research work on geometric and imaging computation on GPU have also been highly evaluated by experts from different research areas. We achieved publications in international conferences, as well as various awards and publicity. These have been the forces to drive us for better research in the GPU area.

Publications

- Jianqing Wang, Ka-Ho Wong, Pheng-Ann Heng, Helen M. Meng and Tien-Tsin Wong, *A Real-Time Cantonese Text-To-Audiovisual Speech Synthesizer*, in Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004), Vol. I, Montreal, Quebec, Canada, May 2004, pp. 653-656.
- Jianqing Wang, Tien-Tsin Wong, Pheng-Ann Heng and Chi-Sing Leung, *Discrete Wavelet Transform on GPU*, in Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, USA, August 2004, to appear
- Jianqing Wang, Tien-Tsin Wong, Pheng-Ann Heng and Chi-Sing Leung, *Discrete Wavelet Transform on Consumer Level Graphics Hardware*, submitted to IEEE Transaction of Circuit and Systems for Video Technology.

Awards

- Multilingual virtual performer won the second prize in the 8th China Challenge Cup Contest in Guangzhou, November, 2003.

Publicity

- Multilingual virtual performer is reported in E-Zone magazine, No. 268, 30 September 2003.

- The research on "Discrete Wavelet Transform on GPU" is reported by www.gpgpu.org (general purpose computation on graphics processing unit), a popular GPU research information web site.
- Our open source implementation of DWTGPU module and extension of JPEG2000 codec–JasPer recieved over 300 downloads within 4 months (http://www.cse.cuhk.edu.hk/~ttwong/software/dwtgpu/dwtgpu.html).
- The virtual character system is used in Hong Kong national I.T gala dinner 18 September, 2003

It is also hoped that by studying these 2 research topics, more and more techniques for geometric and imaging computation on GPU will be explored and shared in the research community, and the vertex and fragment processers can be leveraged for more parallel processing power in different graphics, scientific, or engineering computational intensive tasks. As said above, besides the geometric and imaging computation on GPU, our research work also explore the animation framework for virtual character system, as well as the multilingual language capabilities for it. We hope this GPU-based virtual character animation framework will provide a good solution to be used in different multimedia areas.

1.4 Organization

The thesis is organized as follows. The next chapter provides a brief introduction to the evolution history and the architecture of the programmable graphics hardware, as well as how it can be used for other general purpose computation. Chapter 3 describes geometric computing on GPU – the virtual character framework on vertex processor and the detail of multilingual lip synchronization as well as skeletal animation with motion capture data. In chapter 4, we presents our techniques of imaging computing on GPU, with

our unified algorithm for execute both forward and inverse DWT totally on fragment processor. In the last chapter, we concludes and discuss the future directions for GPU computing.

Chapter 2

Programmable Graphics Hardware

2.1 Introduction

Computer graphics hardware is advancing at incredible rates.When IBM introduced Video Graphics Array (VGA) hardware in 1987, the VGA controller was what we now call a "dumb" frame buffer. This meant that the CPU was responsible for updating all the pixels.

Prior to the introduction of GPUs, companies such as Silicon Graphics (SGI) and Evans & Sutherland designed specialized and expensive graphics hardware. The graphics systems developed by these companies introduced many of the concepts, such as vertex transformation and texture mapping, that we take for granted today. These systems were very important to the historical development of computer graphics, but because they were so expensive, and they are not used by many developers.

NVIDIA introduced the term "GPU" in the late of 1990s when the legacy term "VGA controller" was no longer an accurate description of the graphics hardware in a PC. Today the CPU rarely manipulates pixels directly. Instead, graphics hardware designers build the "smarts" of pixel updates into the GPU. Basically we can categorize the evolution of GPU as 4 Generations.

The first generations of GPUs (up to 1998) includes NVIDIA's TNT2, ATI's Rage, and 3dfx's Voodoo3. These GPUs are capable of rasterizing pre-transformed triangles and applying one or two texture. This generation GPUs relieve the CPU from updating individual pixels. However, they suffer from 2 limitations. First, vertex transformations happen totally in the CPU, 1st generation GPU do not have the ability to do vertex transformations. Second, only a small limited set of mathematical operations for combining textures are supported to compute the color of pixels.

The second generation of GPUs (1999-2000) includes NVIDIA's Geforce 256 and Geforce2. ATI's Radeon 7500, and Savage 3D. These GPUs are capable of doing vertexing transformation and lighting in hardware. While before this generation, fast vertex transformations are only available in high-end workstations. But for the set of mathematical operations supported in the texture blending stage is still limited. It's more configurable, but still not programmable.

The third generation of GPUs includes NVIDIA's Geforce3 and Geforce4 Ti, Microsoft's Xbox, and ATI's Radeon 8500. This generation provides good vertex programmability than the second generation, and more pixel-level configurability is available. But these modes are not powerful enough to be considered truly programmable.

The fourth and current generation of GPUs includes NVIDIA's GeforceFX family with the CineFX architecture and ATI's Radeon 9700. These GPUs provide both vertex-level and pixel level programmability. This is the generation that's really powerful enough to offload complex tasks from the CPU.

2.2 Why Use GPU?

GPUs are designed to be efficient coprocessors for rendering and shading. But the programmability now available in GPUs such as the NVIDIA GeForce FX

and the ATI Radeon 9800 makes them useful coprocessors for more applications. We can see the fact that the time between new generations of GPUs is currently much less than for CPUs, which mean faster coprocessors are available more often than faster central processors. GPU performance tracks rapid improvements in semiconductor technology more closely than CPU performance. This is because CPUs are designed for low latency computations, while GPUs are optimized for high throughput of vertices and fragments [1]. Low latency on memory-intensive applications typically requires large caches, which use a lot of silicon area. Additional transistors are used to greater effect in GPU architectures because they are applied to additional processors and functional units that increase throughput. In addition, programmable GPUs are inexpensive and compatible with many operating systems and hardware architectures.

More importantly, in many graphics and visualization applications great processing power is needed for real-time performance. Since in most applications the ultimate result is rendered on the screen, so moving more simulation computation onto the GPU that renders the result not only reduces computational load on the main CPU, but also avoids the substantial bus traffic required to transmit the results of a CPU simulation to the GPU for rendering. In this way, GPU computation provide an additional tool for load balancing in complex interactive applications.

The general problems with the graphics hardware were the difficulty of programming and the lack of high precision fragment operations and storage. But these issues were mostly resolved by the latest GPUs and software. Following a research trend in the use of high-level shading language to program graphics hardware [2] [3], NVIDIA released its Cg shading language [4]. Also the new generations of GPUs, the NVIDIA Geforce FX series and the ATI Radeon 9700/9800, provide us IEEE 32-bit single precision floating point precision throughout the graphics pipeline. This enables GPU computation to apply to

Generation	Year	Product Name	Transisters	Antialiasing Fill Rate	Polygon Rate
1	1998	Riva TNT	7M	50M	6M
1	1999	Riva TNT2	9M	75M	9M
2	1999	Geforce 256	23M	120M	15M
2	2000	Geforce 2	25M	200M	25M
3	2001	Geforce 3	57M	800M	30M
3	2002	Geforce 4	63M	1200M	60M
4	2003	GeforceFX	125M	2000M	200M

Table 2.1: Features and performance of selected NVIDIA GPUs

those applications that require high dynamic range data. Further generations of GPUs will likely continue to improve in precision and performance.

2.3 Programmable Graphics Hardware Architecture

CPUs normally have only one programmable processor. In contrast, GPUs have at least two programmable processors, the vertex processor and the fragment processor, plus other non-programmable hardware units. The processors, the non-programmable parts of the graphics hardware, and the application are all linked through data flows. The graphics rendering pipeline is illustrated in Figure 2.1.

When we write programs for the vertex processor and the fragment processor, we refer the programs as vertex shaders and fragment shaders (also known as pixel shaders), respectively.

Vertex Shaders

Vertex shaders get executed for each vertex that passes through the pipeline. A vertex shader is a program that has exactly one vertex as input and one vertex as output. A vertex in this context is a structure composed a number of vertex attributes, of which one must be the vertex position. Other vertex attributes

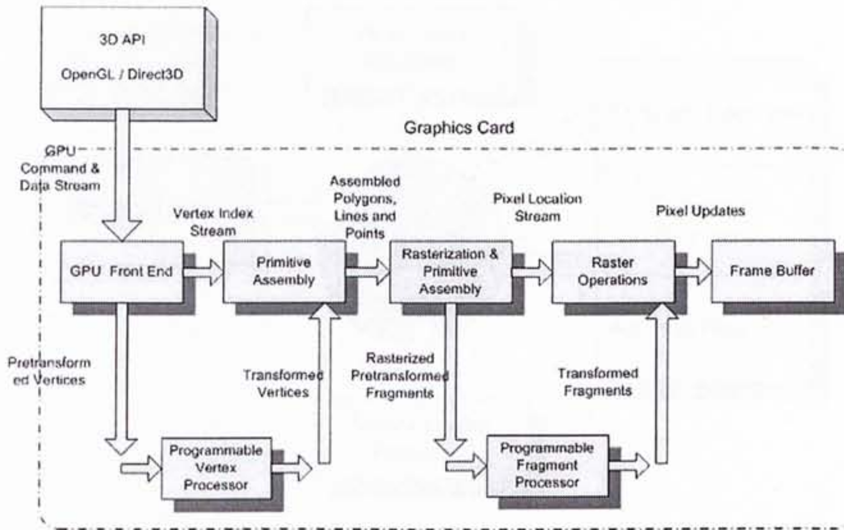

Figure 2.1: The graphics hardware pipeline

Parts of fixed-function pipeline replaced	Parts that are not replaced
Transformation from world space to clipping space	Primitive assembly
Normalization	Frustum culling
Lighting and materials	Perspective Division
Texture coordinates generation	Viewport mapping
	Backface culling

Table 2.2: Vertex engine features

include normal vectors, colors, texture coordinates, or any other user-defined value that is required for the pre-vertex computations in the vertex shader. Vertex shaders can never operate on several vertices at a time.

When a vertex shader is used, some parts of the vertex processing fixed-pipeline is replaced and some are not, which is shown in Table 2.2

As shown in Figure 2.2,vertex shader can read the vertex attributes from a relatively small number of read-only input registers. Using a usually large number of read-only constant registers and a small number of temporary registers the shader then performs its computations. The constant registers contain values that do not change per vertex, but only change once every frame or once every couple of frames. Example for values that are usually stored in the

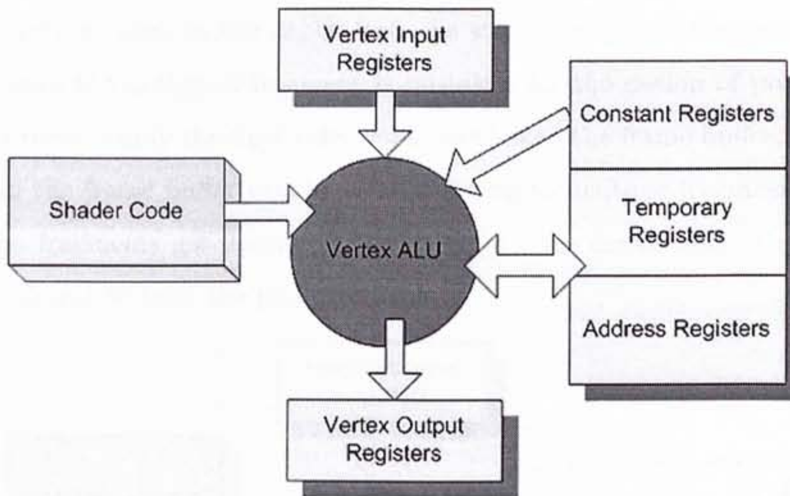

Figure 2.2: The vertex processor

constant registers are the combined world-view-projection matrix and other matrices of use, light direction, etc. A small number of address registers can also be used by the shader to perform indexed relative addressing into the array of constant registers.

Finally, the shader writes its results to a number of write-only output registers. These output registers have a pre-defined semantic meaning such as the transformed, homogenous vertex position, texture coordinates, and vertex colors. These results are then passed on to the next stage of the fixed-function pipeline, and might eventually be used by a possibly activated fragment shader at a later stage in the pipeline.

Fragment Shaders

Fragment shaders get executed per fragment during the rasterization phase in the graphics pipeline. A fragment is a point in window coordinates produced by the rasterizer with associated attributes, such as interpolated color values, a depth value, and possibly one or more texture coordinates. A fragment modifies the pixel in the frame buffer at the same window space location based on a number of parameters and conditions defined by the pipeline stages following

the rasterizer, such as the depth test, the stencil test, or a fragment shader. Sometimes the notion of fragment is mistaken for the notion of pixel. However, a pixel is only the final color value written to the frame buffer, and each pixel in the frame buffer usually corresponding to multiple fragments. Some of these fragments get discarded because of e.g the depth test; others might get combined to form the final pixel color.

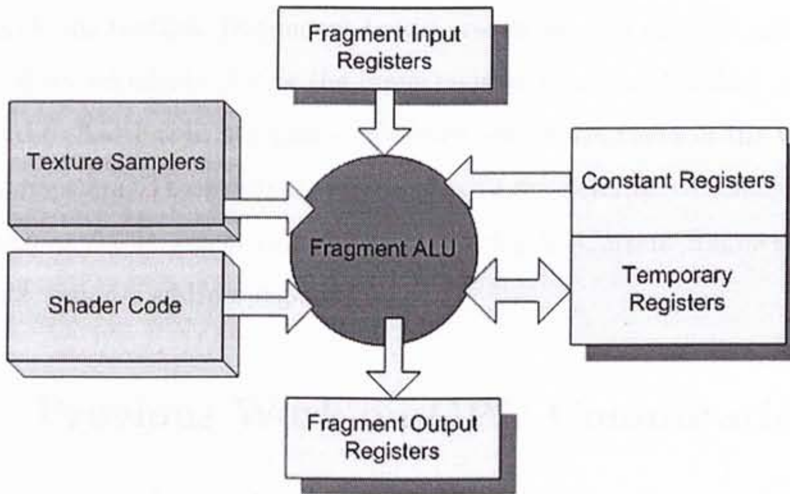

Figure 2.3: The fragment processor

Fragment shaders operate on fragments just before they reach the final stages of the rendering pipeline such as the alpha, depth and stencil tests. The fragment shader receives the vertex shader outputs interpolated across a primitive as input and delivers a single color value and a depth value that gets passed on to the final stages of the pipeline as output.

When a fragment shader is used, some parts of the fixed-function pipeline is replaced, while some are not:

Just like vertex shaders, fragment shaders have access to a number of register files, the input registers contain the interpolated vertex shader results, such as the fragment's color values or texture coordinates. Additionally the fragment shader can look up filtered texture values using texture sampler stages as in Figure.2.3. The fragment shader can either use the interpolated texture

Parts of fixed-function pipeline replaced	Parts that are not replaced
Texture access Texture application and blending Fog and color sum	Alpha test Depth test Stencil test Frame buffer blending

Table 2.3: Fragment engine features

coordinates passed in or texture coordinates computed directly in the shader to sample the texture. Dependent texture reads are also possible, allowing for more advanced effects. Using the input register values and looked up texture values the shader then computes its results and stores them in the write-only output registers. These output registers have a pre-determined semantic meaning, such as the fragment color or fragment depth. Current fragment shaders does not support address registers.

2.4 Previous Work on GPU Computation

The use of computer graphics hardware for general–purpose computation has been an active research for some years. The wide deployment of GPUs in the last several years has resulted in an increase in research with graphics hardware. Trendall and Steward [5] give a detailed summary of the types of computation on GPUs.

For graphics application, programmable graphics hardware has been used for procedural texturing and shading [6] [7] [2] [3]. Graphics hardware has also been used for volume visualization [8] [9], global illumination like ray-tracing [10] [11], photon mapping [12], and radiosity [13].

Other general purpose use of the GPU includes level set segmentation of images and volume data [14] [15], collision detection [16] [17], and computational geometry [18] [19].

The wide variety of applications demonstrated that the GPU has become

an extremely powerful co–processor on our PC. The vector and matrix processing ability as well as the SIMD architecture and parallel pipelines makes it especially useful for various graphics and imaging data processing or other scientific and engineering computations.

Chapter 3

Multilingual Virtual Performer

3.1 Overview

The vertex processor in today's GPU is an ideal platform for performing geometric computations. Provided its matrix and vector processing power and multiple vertex pipelines, when carefully designed we can make use of its parallel processing power and map intensive geometric computation to the GPU and achieve load balancing with CPU at the same time.

A typical example of intensive geometric computation is the facial deformation and skeletal animation for virtual character system.Nowadays 3D virtual characters are widely used in computer games, movies, as well as various multimedia productions. However the development of a real-time 3D virtual character with cinematic high quality rendering and real-time lip-synchronization to speech waveforms remains a challenge in the multimedia industry. This is mainly due to the large data involved, the intensive computation for high quality animation as well as the cost for synchronization with the speech. Even now the computational power of CPU continuously grows, this still puts a heavy burden on the CPU and is very difficult to achieve real-time performance if high quality animation is required.

Also on the speech and lip synchronization aspect of the virtual character, currently most of them are usually designed to be tailored for one specific

language and is hard to be extended to be multilingual, which imposed a limitation for international usage for lip synchronized animation of the virtual character.

Here we propose an approach to develop a real-time multilingual 3D virtual character framework with the animation computation totally on the GPU. We have developed GPU algorithms to effectively explore the geometric data computation power of GPU and offloaded the facial animation and skeletal motion animation computation from CPU. The parallel algorithm greatly release CPU resources for other tasks like speech synthesis and audio-visual synchronization.

Multiple languages is supported through the use of International Phonetic Alphabet(IPA) for lip synchronization visemes. Phonemes, syllables or subsyllables of a language is mapped to the IPA through a mapping table. This approach not only provides an open architecture for adding new languages and make the virtual character capable for multinational languages, but also simplifies the manual geometric design work for the character.

It is also hoped that by studying such a problem, usefulness of such an virtual character framework will be shown to both the graphics and speech community for a lot of potential applications, and the graphics processing unit could be leveraged for more useful features for geometric computing.

3.2 Previous Work

The pioneering work on facial animation was done by Frederic I. Parke in the 1970s [20], and many researchers were interested in this topic in the mid-1980s included the muscle model approach to facial expression of Keith Waters [21].

Development of virtual character with lip synchronization feature with speech has been ongoing for a long time. Such a system offers a multimedia/multimodal presentation of dynamic information, e.g. for news and

weather information reporting, for applications in entertainment, for personified dialog systems, or as an aid for the hearing-impaired [22] where the simulated lip movements can help the user decipher the spoken message. The talking face can also convey non-verbal communicative signals, such as emotions. The late 1980s also saw the first attempts at this, including the work of Dominic Massaro and Michael Cohen, and number of other researchers. Previous approaches include an image-based synthesizer as in [23] that concatenates viseme images. A viseme is a facial image treated as a unit in video that corresponds to a phoneme unit in speech. An alternative approach involves parameterized lip shapes, such as the facial animation parameters (FAPs) in the MPEG-4 standard. This has been applied to three-dimensional facial animation. Additionally, there has been previous work in lip-synced virtual character for languages aside from English, such as the Italian talking head described in [24].

Significant work has also occurred in graphics for deforming articulated characters using geometric methods [25] [26] [27] and physically-based methods [28] [29] [30]. Despite this, most character animation in interactive applications, such as video games, is based on a geometric skeletal deformation technique commonly refered to as skeletal subspace deformation, in which vertex locations are weighted averages of points in several coordinate frames [25] [31].

Our Contribution

Our approach mainly focus on attacking and give efficient solution to the following problems:

- Develop an efficient and easy framework for facial animation, lip synchronization, and seamlessly integrate the facial and skeletal character animation.
- High quality rendering but in real-time performance and consumes as

little CPU resources as possible, by means of performing intensive geometric computation on GPU.

- Multilingual support for lip-synchronized facial animation

3.3 System Overview

Our achievement in this project is the development of a simple and efficient virtual character framework, with GPU acceleration on animation for real-time performance, and multilingual support for lip synchronization with speech. An multilingual Text-To-Speech system interface module for real-time generation of speech wave form and timing curves is also designed, which enable us to incorporate more text-to-speech system in different languages in the future.

The overall structure of our multilingual virtual performer system is shown in Figure 3.1. Basically it can be divided into 3 big modules, the animation system , the audio system and the multilingual texture-to-speech system.

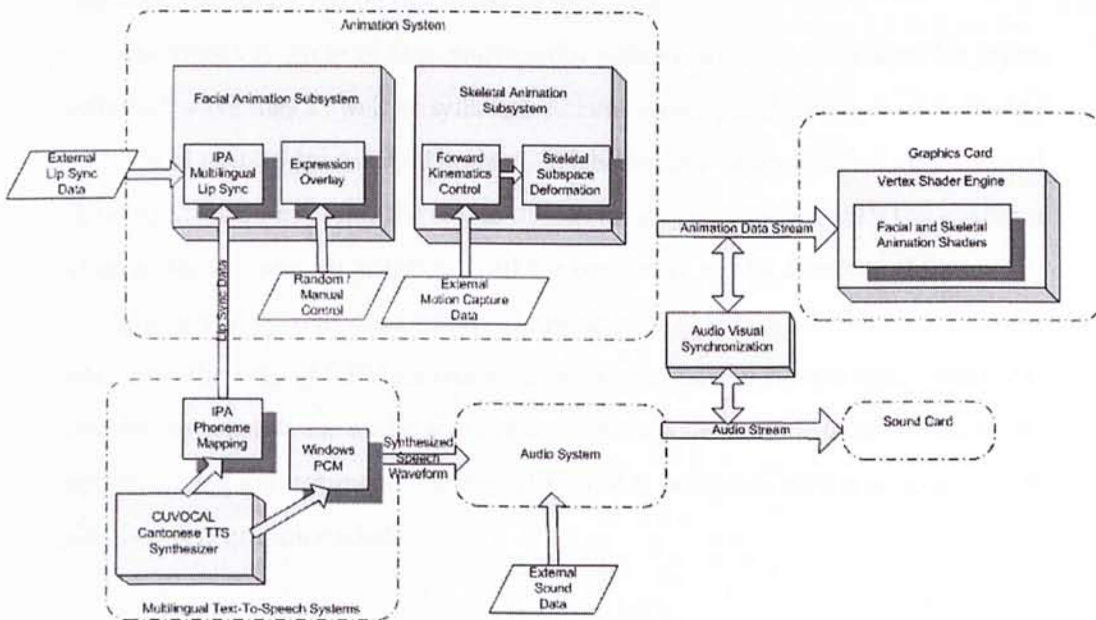

Figure 3.1: Virtual character system architecture

The animation system is composed of facial animation control subsystem and skeletal character animation control subsystem. Inside the facial animation system, the multilingual lip-sync and expression overlay submodules are responsible for providing a multilingual lip synchronization interface and facial expression control, respectively. Lip synchronization data (to be discussed in section 3.4) can be input in the form of external weigh function sample points data generated manually, or from our real-time cantonese text-to-speech synthesis system – CU Vocal directly. The facial expression overlay provides an interface as a real-time user control as well as random generation of expression parameters for increased realism.

We have implemented the forward kinematics control for the skeletal character animation system totally on the GPU, with the interface for input motion capture data. The skeletons or bones of the character is animated by the motion capture data and at the same time a skeletal subspace deformation algorithm is used to further deform the polygonal mesh that is attached to the skeleton hierarchy (to be discussed in section 3.5).

The audio module of this multimedia system provides interfaces for input external wave files as well as synthesized cantonese speech audio wave from the CU Vocal text-to-speech synthesizer. With the help of the central audio-visual synchronization module, the sound or speech is synchronized with the graphics stream that is sent to graphics card for rendering to the frame buffer.

The whole system does not need any special graphics workstation to operate, with the help of GPU acceleration on the animation computation, the CPU can be responsible for audio and synchronization task and produce smooth lip-synchronized animation on a normal PC that is equipped with a programmable GPU-based graphics card.

3.4 Facial Animation

There have been many approaches developed for facial animation, mainly they can be characterized into 2 categories – physically correct muscle-skin simulation and free form deformation, which only gives the approximation.

The muscle-skin simulation methods give a more accurate result for facial deformation. They usually need further processing on the polygonal mesh to make it in a multi-layered tissue structure for simulation instead of the polygon surface. Finite element methods are commonly used to simulate the movements of discrete nodes in the tissue layers, which are connected by virtual springs. So generally it is a discrete deformable model (DDM) with node-spring-node structure. Although different spring model can be used in simulation, such as Voight viscoelastic model or Hill's Model, it is usually very computational intensive to solve the whole system.

The free form deformation methods do not simulate the physical process of skin–muscle deformation, but only deform the face freely using different approaches like feature points, control points, or morphing. Usually the motion of a set of predefined points are recorded first, they are used to smoothly influence different regions of the face to create facial animation. The morphing technique is to gradually displace and move all the vertices of 1 face model to the corresponding coordinates of another. The free form deformation methods only give us an approximation of the facial animation, but generally are more efficient and simple.

Facial animation can be synchronized to speech in several ways. The employed method depends mainly on the kind of speech data which is available for synchronization, e.g. whether the audio signal, the phonemes and timing of an utterance is available or only a text representation.

The text-driven approach received a text as input which is transcribed into its phonetic representation. This information is used to generate both

synthetic audible speech and synchronized facial animation.

The speech-driven method takes pre-recorded speech as input. The audio file is analyzed for phonemes and timing information. This data is used to create the facial animation which is performed synchronously to the audio file play–back.

Although there are currently various facial animation methods for different facial models, they are either complex, computational intensive or on the other hand difficult to be accelerated using GPU computing. At the same time, most methods did not address the problem of multilingual support for lip synchronization. So we take an approach to extend the face space theory from Steve DiPaola [32], apply the high dimensional face space concept to the polygonal face models, while at the same combine the simplicity of morphing in free form deformation. On the other hand, the multilingual lip synchronization feature is also carefully considered and handled when designing the facial animation algorithm. For the lip synchronization part, we make use of both text and speech driven approaches to provide interface for the multilingual text-to-speech system as well as external sound data.

3.4.1 Facial Animation using Face Space

Imagine an n-dimensional space describing every conceivable humanoid face, where each dimension represents a different facial characteristic. Within this continuous space, it would be possible to traverse a path from any face to any other face, morphing through locally similar faces along that path. A multi-dimensional space is a convenient way to generate a universe of faces and for making facial animations.

Here we further extend the concept to polygonal face models, which can be represented as a point in the high dimensional face space using all of the vertices. Denote a face f in the vector form

$$f = [x_0, y_0, z_0, x_1, y_1......x_n, y_n, z_n] \tag{3.1}$$

If we can predefined a set of faces with the same topology in this high dimensional space. Denote this as a face set Ψ

$$\Psi = \{f_i | f_i \subset R^{3n} \; i = 1..k\} \tag{3.2}$$

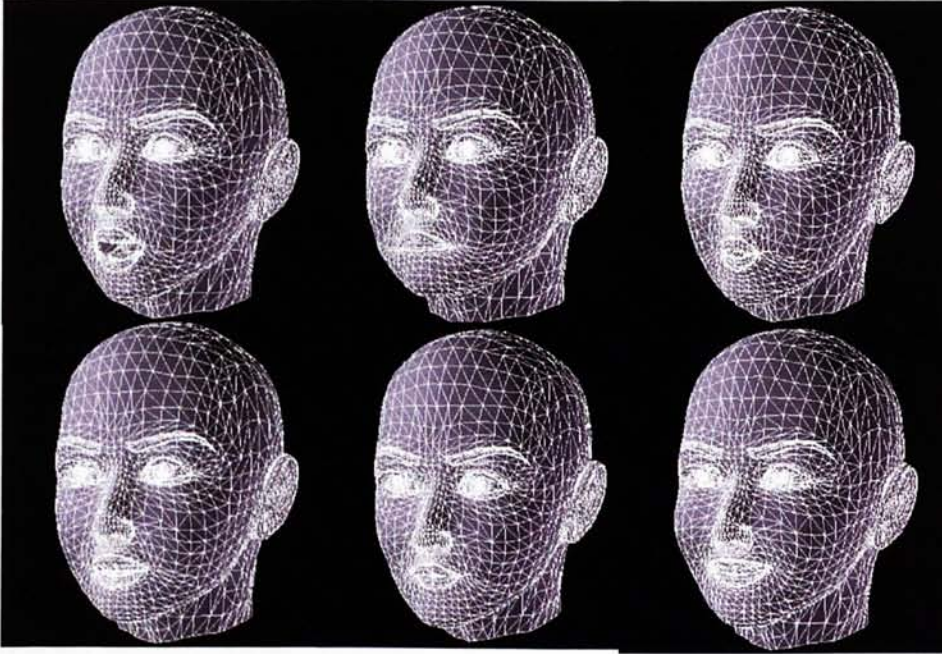

Figure 3.2: Face set in the high dimensional face space

Similar to vector space in mathematics, a new face in the face space can be generated by linearly combining the faces in the face set.

$$f' = \sum_{j=1}^{k} \alpha_j f_j \tag{3.3}$$

The advantage of this representation is that, now each new face can be represented only by the set of weight coefficients $\{\alpha_j\}$ for the basis faces, which reduce the dimension of the space to the number of basis faces k. In order to

create facial animation, we can simply animate the weights by making them a set of functions that change with respect to time, in the form of $\{\alpha_j(t)\}$.

And now facial animation can be viewed as a timed path from one point in the face subspace to another, this process can also be represented as a timed weighted blending of the faces in the face space.

$$f'(t) = \sum_{j=1}^{k} \alpha_j(t) f_j \tag{3.4}$$

Here the size of the face set will affect the degree of freedom in the facial animation, and the face set should belong to the same person in order to animate one particular face.

However, this approach is not so intuitive to be used for animation purpose, it is rather difficult to determine the function of blending weights with respect to time for every face in the face set. Therefore, in our approach, we incorporate some modifications of the above method. Animation is achieved by the use of deformation vectors (DV) derived from the target viseme/emotion models. We define a neutral face $f_{neutral}$ in the face set which represents a closed mouth model with no facial expression, which can also be considered as the origin of our defined face space. Then the DV for smile, for example, is defined as

$$DV_{smile} = f_{smile} - f_{neutral} \tag{3.5}$$

Different DVs can be then linearly combined with the neutral face model ($f_{neutral}$) to form a new face model (f_{new}), then further form facial animation by varying the blending weights as illustrated in the equation.

$$f'(t) = f_{neutral} + \sum_{i} \alpha_i(t) DV_i \tag{3.6}$$

where DV_i is the DV for the ith face model in the face set and α_i is the blending weight for the ith DV.

Although the concept of DV is simple, it provides us with a clear meaning for the weights for each DV. They serve as the basis for spanning a face subspace, and the neutral face is the origin of this subspace. A weight with value 0 means no deformation/action for this DV, and a weight value 1 means deforming the face with the deform meaning of the DV to its maximum state. Like a weight 1 for a smile DV means to deform the face to smile expression to the largest extent.

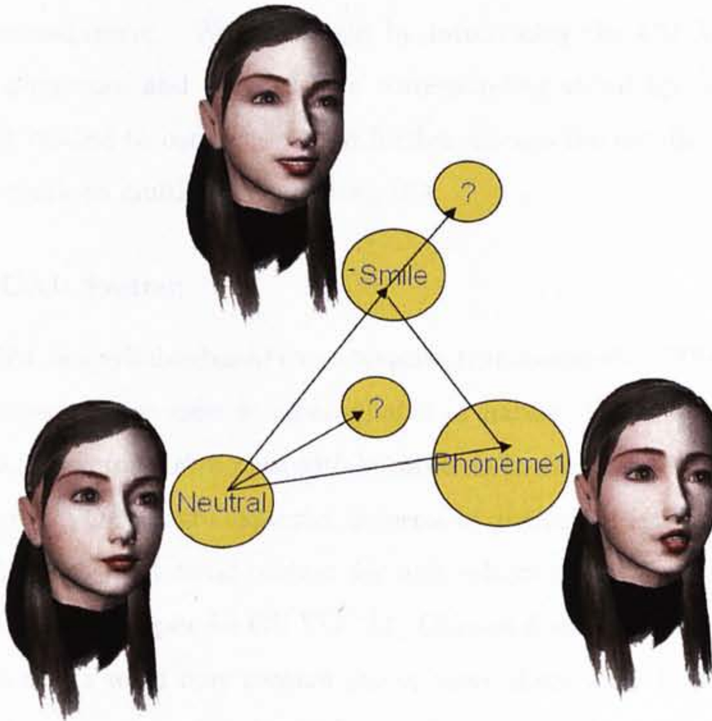

Figure 3.3: Using deformation vector as basis to span a high dimensional face space

And now the facial animation problem can be seen as to

1. determine an appropriate set of predefined faces/DVs Ψ in the face space, which are adequate for performing both speech and facial expressions.
2. determine a set of blending functions $\alpha_j(t), j = 1..k$ that actually performs the blending animation that changes with time

3.4.2 Face Set Selection for Lip Synchronization

In this section we mainly focus on how to solve the problem of selecting an appropriate set of predefined faces/DVs that is suitable for multi-language lip-synchronization and facial expression animation.

Since this system is mainly for being used with a real-time cantonese text-to-speech synthesizer – CU Vocal, which is developed by the human-computer communications laboratory of department of systems engineering and engineering management. We will begin by introducing the CU Vocal system and the phonemes and visemes (the corresponding visual lip shape for the phoneme) related to cantonese, then further discuss the details of extending the framework to multi-language using IPA.

CU VOCAL System

CU VOCAL is a syllable-based concatenative text-to-speech (TTS) synthesizer for Cantonese. Cantonese is monosyllabic in nature (like Chinese) and the dialect has a rich tonal structure with between six to nine tones. Coarticulatory effects in CU VOCAL are captured in terms of distinctive features. The TTS engine also uses right tonal context for unit selection. Figure 3.4 illustrates typical input and output for CU VOCAL. Chinese does not have explicit word delimiters and a word may contain one or more characters. Hence the input Chinese character string is tokenized into Chinese words by a greedy algorithm with reference to a lexicon and the word pronunciations are looked up from a dictionary. For example, in Figure 3.4, the first character in the input text string (meaning: you), is pronounced as /nei5/ (i.e. the syllable is /nei/ with tone 5. The syllable inventory adopted in CU VOCAL follows the LSHK convention. CU VOCAL generates the synthetic speech output in windows PCM waveform format. The TTS engine has also been extended to explicitly generate the syllable sequence with timing information, which are also very

important for blending weights generation (to be discussed in section 3.4.3), e.g. the first syllable unit /nei5/ has a duration of 0.39 second, the fourth unit LP indicates a pause (silence) for 0.504 second and the last two syllables are /lam4/ of duration 0.32 second each. The syllable unit can be further subdivided into an optional onset (i.e. the consonant that starts the syllable), a nucleus (i.e. the core vowel/diphthong) and an optional coda (i.e. the consonant that ends the syllable). The Chinese syllable unit is often subdivided into an initial (i.e. the onset) and the final (i.e. the nucleus and coda). For example, the syllable /nei/ has initial /n/ and final /ei/ (or onset /n/ and nucleus /ei/). The syllable /lam/ has initial /l/ and final /am/ (or onset /l/, nucleus /a/ and final /m/).

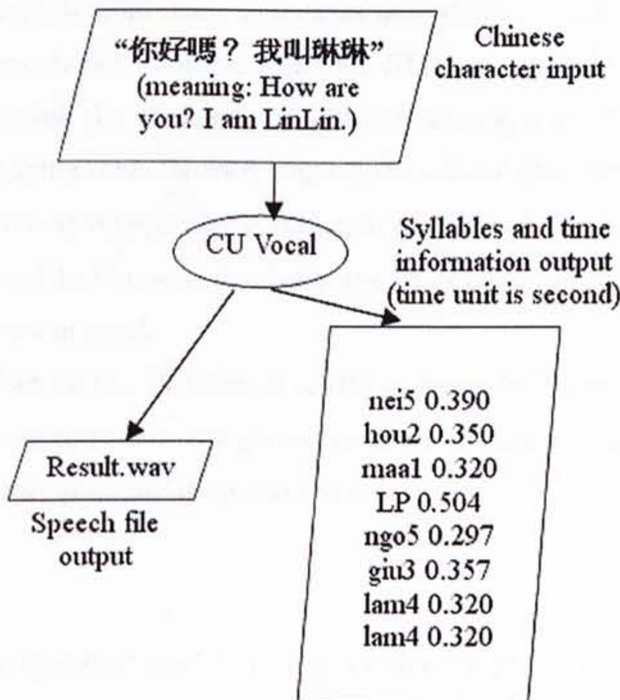

Figure 3.4: Sample input and output of CU Vocal

Multilingual Support Using IPA

Since much previous work defined visemes in relation to phonemes, our approach involves decomposing a syllable into its onset, nucleus and coda and mapping these to their closest phonetic symbol.

Up to this point, it might be natural to come up with a solution for the cantonese lip-synchronization problem that we choose the face set that corresponds to every LSHK cantonese symbol. However, this solution can not support for future extension of the system. Whenever we want to add in a new language, we have to make another face set for the particular phonemes of that language. This is a tremendous waste of time and system resources. However, from the speech point of view, an obvious fact we can see is that the phonetics symbols from many languages may share the same pronunciation, and the International Phonetic Alphabet (IPA) can provide a good solution for us to combine the phonemes of different languages into 1 set of symbols. We use a mapping table for each language to achieve this purpose and a total of 28 IPA symbols is used. Since in human speech, different phonetic symbols may correspond to the same lip shape, the 28 symbols are further reduced to only 15 visemes in total.

The face set for the 15 Visemes of IPA is shown in Figure 3.5

We will now show how the phonemes of each language can be mapped to the IPA to share pronunciation and visemes.

Cantonese

The mapping table from LSHK to IPA syllable is proved below in 4 different parts, onsets, nuclei, codas and nasals which are shown in Tables 3.1, 3.2, 3.3 and 3.4.

Japanese

Phoneme ID	Phoneme Symbol	IPA ID	IPA Symbol
0	closed	0	closed
1	b	8	M
2	p	8	M
3	m	8	M
4	f	11	F
5	d	7	L
6	t	7	L
7	n	9	N
8	l	7	L
9	g	15	K
10	k	15	K
11	ng	10	NG
12	h	2	EH
13	gw	5	UH
14	kw	5	UH
15	w	5	UH
16	z	13	S
17	c	13	S
18	s	13	S
19	j	14	SH

Table 3.1: Cantonese onsets mapping

Phoneme ID	Phoneme Symbol	IPA ID	IPA Symbol
20	aa	3	AA
21	i	1	IY
22	u	4	O
23	e	2	EH
24	o	4	O
25	yu	14	SH
26	oe	2	EH
27	a	3	AA
28	eo	2	EH

Table 3.2: Cantonese nuclei mappings

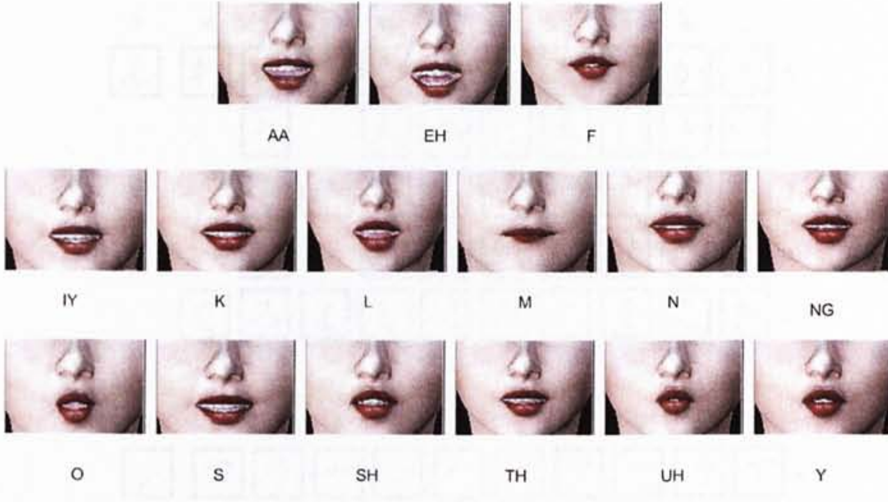

Figure 3.5: IPA visemes

Phoneme ID	Phoneme Symbol	IPA ID	IPA Symbol
29	p	8	M
30	t	7	L
31	k	15	K
32	m	8	M
33	n	9	N
34	ng	10	NG
35	i	1	IY
36	u	5	UH

Table 3.3: Cantonese codas mapping

As we all know that the basic phonetic symbols for Japanese language is called the Hiragana and Katakana with a total of 54 symbols (Figure 3.6), and we further decompose them into phoneme elements and mapped to the IPA symbols. For example, "ka" maybe further decomposed to phonemes "k" and "a", and then mapped to the most similar IPA symbol.

Mandarin

Phoneme ID	Phoneme Symbol	IPA ID	IPA Symbol
37	m	8	M
38	ng	10	NG

Table 3.4: Cantonese nasals mapping

N	w	r	y	m	h	n	t	s	k	_	
ん	わ	ら	や	ま	は	な	た	さ	か	あ	a
		り		み	ひ	に	ち	し	き	い	i
		る	ゆ	む	ふ	ぬ	つ	す	く	う	u
		れ		め	へ	ね	て	せ	け	え	e
	を	ろ	よ	も	ほ	の	と	そ	こ	お	o

N	w	r	y	m	h	n	t	s	k	_	
ン	ワ	ラ	ヤ	マ	ハ	ナ	タ	サ	カ	ア	a
		リ		ミ	ヒ	ニ	チ	シ	キ	イ	i
		ル	ユ	ム	フ	ヌ	ツ	ス	ク	ウ	u
		レ		メ	ヘ	ネ	テ	セ	ケ	エ	e
	ヲ	ロ	ヨ	モ	ホ	ノ	ト	ソ	コ	オ	o

Figure 3.6: Japanese characters with phonetic representation

For the Mandarin "Pin Yin" phonetic symbols, it may be further decomposed into "sheng mu" and "yun mu", each of them could be successfully mapped to IPA.

English

Since the international phonetic alphabet is derived and extended from English phonetic symbols, so the IPA symbols has a natural corresponding with those in English. Thus, the mapping table is not listed here.

Overall, no matter for any specific language, using our approach can always simplify the lip-sync animation problem to:

$$f'(t) = f_{neutral} + \sum_i \alpha_i(t) DV_{i,ipa} \tag{3.7}$$

Phoneme ID	Phoneme Symbol	IPA ID	IPA Symbol
0	closed	0	closed
1	a	3	AA
2	i	1	IY
3	u	5	UH
4	e	2	EH
5	o	4	O
6	k	15	K
7	s	13	S
8	t	7	L
9	n	9	N
10	h	2	EH
11	m	8	M
12	y	6	Y
13	r	7	L
14	w	5	UH
15	g	15	K
16	p	8	M
17	b	8	M

Table 3.5: Japanese mapping table

All we need to do now is to define the blending weights functions $\{\alpha_i(t)\}$.

3.4.3 The Blending Weight Function Generation and Coarticulation

Based on our IPA lip-sync approach, the lip-sync animation can be generated when the blending weights function for the DVs are determined by the speech. A critical aspect for realistic lip synchronization is the simulation for the coarticulation effect. Coarticulation can be defined as the smooth blending between adjacent phonemes. Because adjacent phonemes can influence each other, the dominance of a phoneme does not automatically cease at the phoneme boundary but can well reach into other phonemes. Thus the weight function of neighboring phonemes may overlap.

In our system, the coarticulation effect is achieved in the generation of blending weight function. First, the time and duration that each phoneme

Phoneme ID	Phoneme Symbol	IPA ID	IPA Symbol
0	closed	0	closed
1	a	3	AA
2	o	4	O
3	e	2	EH
4	i	1	IY
5	u	5	UH
6	v	5	UH
7	b	8	M
8	p	8	M
9	m	8	M
10	f	11	F
11	d	7	L
12	t	7	L
13	n	9	N
14	l	7	L
15	g	15	K
16	k	15	K
17	h	2	EH
18	j	2	EH
19	q	2	EH
20	x	2	EH
21	z	13	S
22	c	13	S
23	s	13	S
24	r	14	SH
25	zh	14	SH
26	ch	14	SH
27	sh	14	SH
28	y	6	Y
29	w	5	UH
30	ng	10	NG

Table 3.6: Mandarin mapping

appeared in the speech are determined, so that outside this time range the weighting for the corresponding DV will be set to 0, then, in order to generate the transition between the 2 phonemes for smooth animation, smoothing and interpolation is applied and the duration of each phoneme DV weighting that is nonzero can be generally categorized into 2 phases, a increasing and decreasing phase. This can be explained using a simple example that transits from one phoneme to another (Figure 3.7). The transition time is actually the increasing phase for the second phoneme but the decreasing phase for the first phoneme, we gradually decrease the weight of the first phoneme DV from 1 to 0, while for the second one, we gradually increase its weighting from 0 to 1.

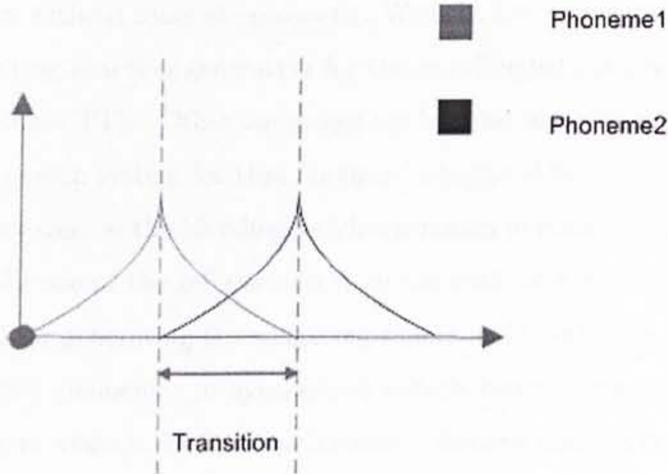

Figure 3.7: The coarticulation between 2 adjacent phonemes

For smooth coarticulation, we can use different interpolation technique for the increasing or decreasing phase. Like the negative exponential function (Figure 3.7):

$$\alpha(t) = \lambda e^{-|t-(t_{start}+\frac{t_{dur}}{2})|} \tag{3.8}$$

where t_{start} is the starting time of the phoneme and t_{dur} is the duration of it. Due to the reason that each phoneme usually takes very little time so that accurate interpolation that based on physical simulation is not needed

here. In our system, we choose to use linear interpolation for its simplicity and efficiency. The difference between 2 interpolation techniques is negligible through the results of experiment.

There are 2 methods to generate the blending weights function – by using extra information from TTS systems and also by manual processing.

Automatic Blending Weights Function Generation Using TTS system

By using a text-to-speech system and making good use of its output timing information for phonemes, we can achieve automatic blending weights function generation without manual processing. Without loss of generality, we describe our weighting function generation for the multilingual lip-sync interface with the Cantonese TTS . Other languages are handled in a similar way if we have a text-to-speech system for that language integrated in.

Here we discuss the blending weights function in more detail for Cantonese as we make use of the information from the cantonese text-to-speech system CU Vocal for generating the weighting functions in real-time. The transition between two phonemes in synthesized speech corresponds to the transition between two visemes in facial animation. Smooth transition is achieved by controlling the weights in the blending technique. We will elaborate on this point by means of an example.

Consider for a Chinese word meaning "center" pronounced as /zung/ /gan/ in LSHK syllables. For a given syllable, we reference the CU VOCAL syllable corpus to get the average duration among the occurring instances. For example, the syllable /zung/ averages 0.33 second in duration. We also reference the corpus to get the average fraction of the syllable's duration that is occupied by its initial and final respectively. For example, the syllable /zung/ has the initial /z/ and final /ung/. The initial /z/ takes up about a quarter of the syllable's duration on average, while the remaining three quarters is taken

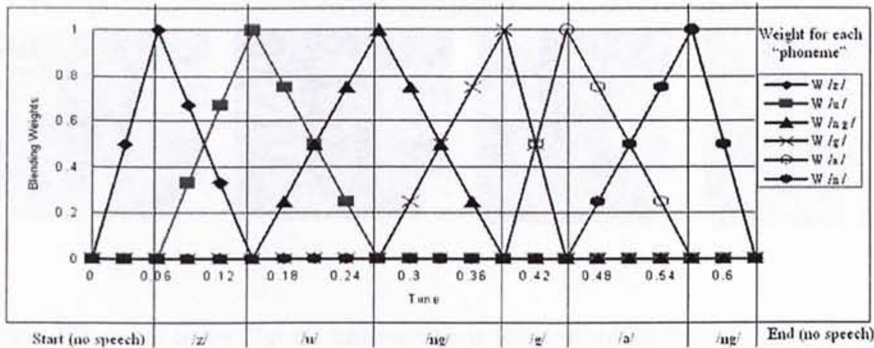

Figure 3.8: Variation of blending weights over time for animation

up by the final /ung/. The final can be further subdivided into the nucleus /u/ and coda /ng/. For the sake of simplicity, we assume the nucleus and coda for the final /ung/ have equal average durations. Hence about 0.25 of the average duration of /zung/ is occupied by the syllable onset /z/, about 0.375 by the syllable nucleus /u/ and the remaining fraction of 0.375 by the syllable coda /ng/. In order to use this information for facial animation, we locate the visemes that correspond to the IPA symbols /z/, /u/ and /ng/ respectively. Since these are static viseme models, we need to determine the blending weights that correspond to these visemes for 3D animation A linear interpolation is used as shown in Figure 3.8. Each viseme starts with a unity weight at its start instant, and linearly decreases to zero weight at its end point. This defines the variation of the blending weights over time and our system demonstrates that this achieves a realistic and smooth facial animation effect.

By using this weight function generation technique through the CU Vocal text-to-speech system, real-time lip synchronized animation can be generated quickly from texts. An example of rendering of a live news report using this approach is shown in Figure 3.9.

Figure 3.9: Live news report animation using weight function generated automatically by CU Vocal TTS system

Manual Weight Function Generation

The blending weight function can also be generated manually with the help of some software that can analyze the waveforms of a sound file, for example, GoldWave (Figure 3.10). It is used as a tool for examining the sound file and marking the starting time and duration of each phoneme manually. Thus, the scattered data points for weighting function or we can also call them the "keyframes" of the phonemes can be obtained this way and a system module for loading the lip-sync data and do further processing for smooth coarticulation is also implemented.

3.4.4 Expression Overlay

The expression is generally overlayed onto the lip synchronized face with the same approach, i.e, using the DV to further deform the lip synchronized face into emotional state, this involves emotion DVs like smile, worry and so on.

Now, together with the lip synchronized animation, we can produce lively facial animation using the following equation:

$$f'(t) = f_{neutral} + \sum_i \alpha_i(t) DV_{i,ipa} + \sum_j \beta_j(t) DV_{j,exp} \tag{3.9}$$

In our system, the variation of blending weights for emotion face models (Figure 3.11) can be defined manually by the user by means of a slider rule

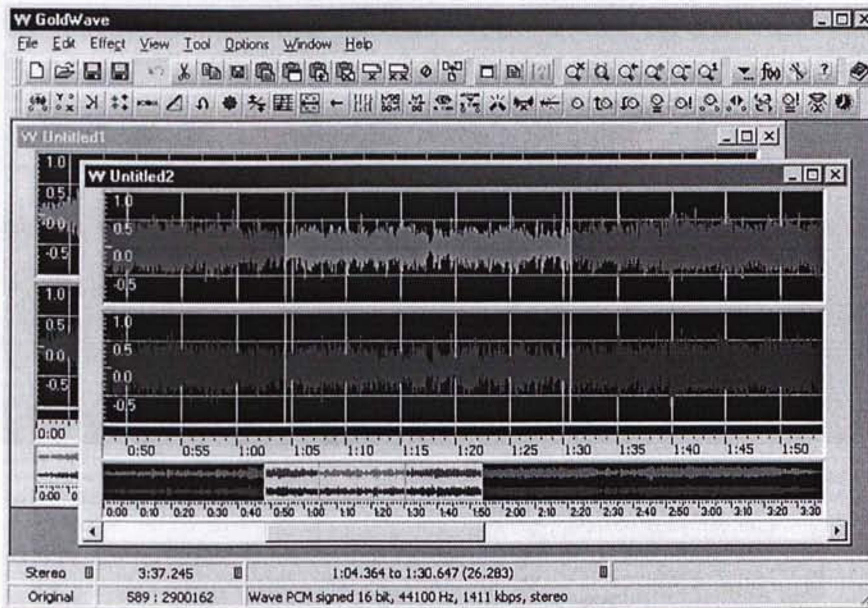

Figure 3.10: Manual weight function generation using waveform analyze software

in our system's interface. These weights are used in a similar way for 3D face rendering weights. Our system is capable of change and blending more than 1 facial expressions simultaneously.

3.4.5 GPU Algorithm

Due to the large computation burden involved for blending and animating our virtual character in high resolution, we explore the powerful SIMD processing feature of the GPU and employ the vertex shader engine to perform the facial animation calculation.

Face Deformation

Since basically the vertex engine can be viewed as a stream processor, the first step to map the equation

Figure 3.11: Facial expressions

$$f'(t) = f_{neutral} + \sum_i \alpha_i(t) DV_{i,ipa} + \sum_j \beta_j(t) DV_{j,exp} \tag{3.10}$$

to the GPU is to streamlize our data. Taken into account that normally there can not be more than 2 phonemes and 2 expressions for the human face simultaneously. In order to add in more realism, we add 1 more channel for eye action. We categorized the input data for blending into 6 channels of streams – neutral face, IPA phoneme DV1, IPA phoneme DV2, expression DV1, expression DV2, eye action. These streams are also called varying data parameters because they vary with different vertex in the vertex shader. The blending weights for the IPA and expression DV remains constant for all the vertices in every frame rendered, thus we store them into the constant memory or registers. The structure for the vertex shader is illustrated in Figure 3.12.

As shown above, the stream data are in 6 channels. Besides the vertex, normal, and texture coordinate data for the neutral face, which are stored in the corresponding registers. We put other stream data like the IPA phoneme DVs and the expression DVs into other registers like the high precision texture coordinate registers and color registers. The constant memory also contains 4–floats vector registers that we can put in the blending weights and other

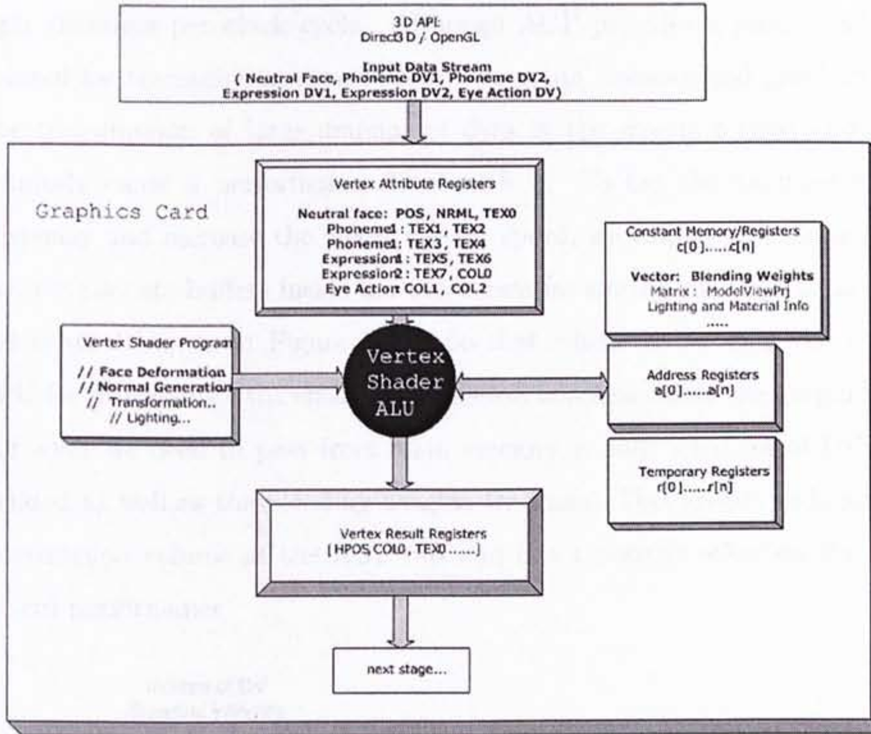

Figure 3.12: Vertex shader structure for facial animation

standard matrixes for transformation like the model–view–projection matrix and as well as the relevant lighting data. Then the vertex shader program for computing the face deformation equation and perform normal generation (to be discussed in next section) is executed. After this, the deformed vertex is transformed and lighting is calculated, that are passed to the output registers for the next stage in the pipeline.

We also explored the data transmission efficiency in our system. Most graphics card today use the AGP bus as the major channel for transmitting data with main memory. The AGP bus is 32 bits wide, just the same as PCI is, but instead of running at half of the system (memory) bus speed the way PCI does, it runs at full bus speed. AGP in its lowest speed mode has a bandwidth of 254.3 MB/s. In addition to doubling the speed of the bus, AGP has also defined a 2X ,4X and 8X mode, which perform two, four,

eight transfers per clock cycle. Although AGP provides a large bandwidth channel for transmitting data between the main memory and graphics card. The transmission of large amount of data in the stream format above will definitely cause a performance down with it. To use the bandwidth more efficiently and increase the whole system speed, we employed a method that directly allocate buffers inside the video ram for storing the neutral face data and other DVs, as in Figure 3.13. So that whenever we pass the data to GPU for processing , the entire transmission happens inside the graphics card and what we need to pass from main memory is only 5 indices of DV to be blended as well as the blending weights for them. This greatly reduced data transmission volume at the AGP bus and has a positive effect on the whole system performance.

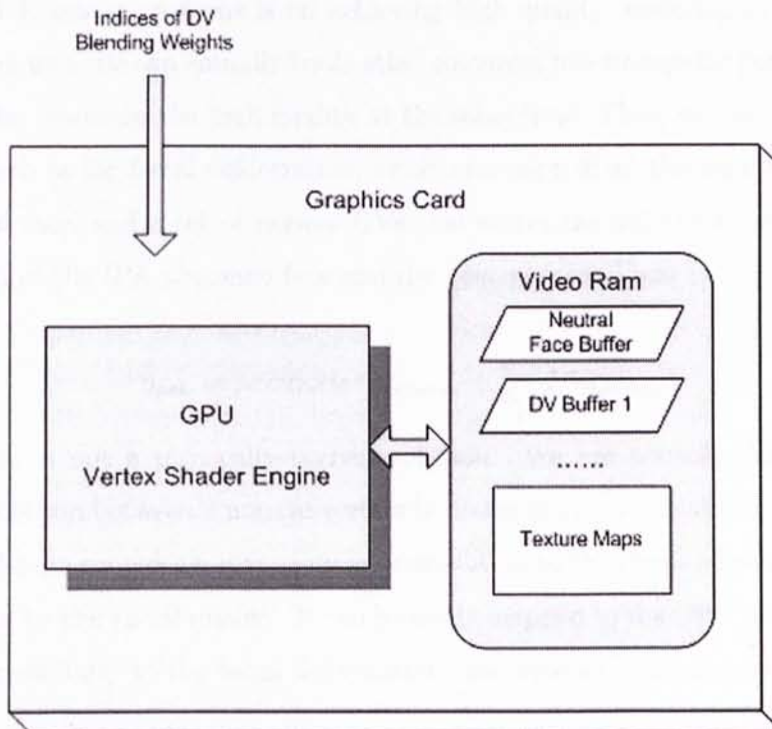

Figure 3.13: Using video ram as stream data buffer

Normal Vector Blending

In our facial animation, when we deform the neutral face using a set of DVs, the normal vectors on the face actually changes with the geometry as well. For realistic rendering of the facial animation, the normal vectors are an important part for lighting calculation. So it is needed for us to obtain the new normals after deformation for rendering.

However, recalculation of the face normal based on the new geometry is tedious and will consume even more computing resources than the face deformation process. In a traditional way, for a polygonal face mesh in our system, it will involve computing the plane normal for each triangle and then averaging the triangle plane normals at the vertex that is shared by more than 1 triangle polygons, in order to get the new vertex normal.

While our main focus is on achieving high quality rendering in real-time performance, we can actually trade other resources like storage for performance and also maintain the high quality at the same time. Thus, we take the same approach as for facial deformation, we store a copy of all the normals for the neutral face, and a set of normal DVs that stores the difference between the normal of the IPA phoneme face and the neutral face. Thus

$$n_{new} = normalize(n_{neutral} + \sum_i DV_{normal,i}) \tag{3.11}$$

This is not a physically–correct solution. We are actually doing linear interpolation between 2 normal vectors by using the equation above. However, our solution provides a very close approximation to the physically–correct one judging by the visual quality. It can be easily mapped to the GPU using vertex shader similarly to the facial deformation case without increasing complexity.

3.5 Character Animation

Many approaches have been devised for character animation through the years. Generally, we will be able to choose between explicit and implicit methods.

Explicit methods store the sequence of animated vertices from our geometry every few frames, like snapshots from a movie. They are easy to code and involve simple math. But on the other hand, storing animated vertices is memory intensive, so they are also called memory-hungry methods.

Implicit methods do not store the animation data, but instead store a higher level description of the motion. Skeletal animation system, for example, store the configuration (in terms of rotation angles) for each joint, like the elbow, knee and so on in our virtual character. Then, in real time, this description is mapped to an unanimated character mesh, so the animation is computed. This computation usually involves complex math with trigonometry and matrices. Thus these methods are all fairly intensive for the CPU, but they only need small data structures to convey the description of the motion.

In our project, we choose to use the implicit method – the skeletal animation system for character animation. It is more powerful and offer sophisticated controls with very low memory consumption. We free the CPU from the intensive computation by making use of the parallel vertex processing capability of GPU. By mapping the skeletal animation algorithm to the GPU, the animation efficiency is improved and CPU computing resources can be used for other tasks like sound processing, audio-visual synchronization, etc.

3.5.1 Skeletal Animation Primer

Skeletal animation is a implicit technique used to pose character models. A skeleton is embedded in, and attached to, a character model. Once the skeleton is attached, the character model becomes the skin. Posing the skeleton causes the skin to be deformed to match the position of the underlying bones (Figure

3.14).

The skeletal structure, as you can imagine, is a series of connected bones that form a bone hierarchy. The bones are connected through joints. One bone, called the root bone, forms the pivotal point for the entire skeletal structure. All other bones are attached to the root bone, either as child or sibling bones.

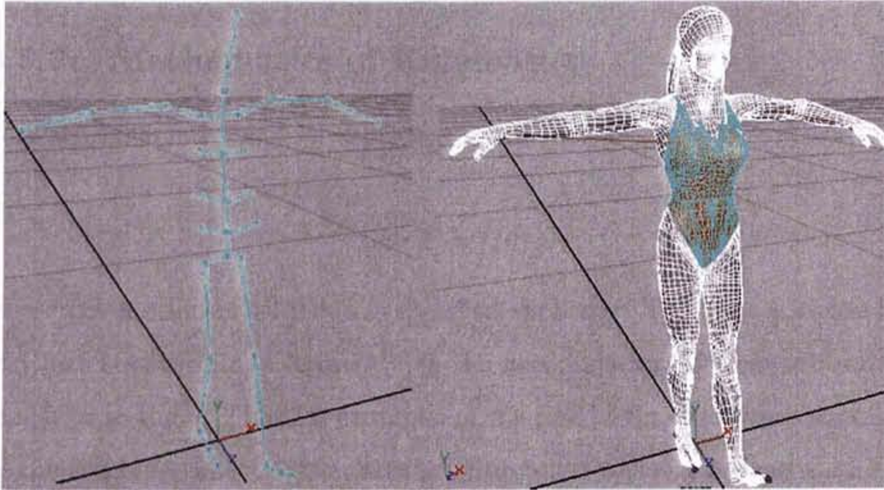

Figure 3.14: Skeleton hierarchy and the corresponding mesh

In order to animate the skeletal structure, two general approaches exist: forward kinematics (FK) and inverse kinematics (IK).

In forward kinematics, we will start from the a root node and propagate the skeleton downward, inheriting motions as we advance. The upper arm should inherit the chest, the lower arm both the upper arm and chest, and finally the hand. Forward kinematics is coded by stacking series of transformation matrices as we enter each body part. It is the method of choice for motion-capture data. Inverse kinematics works the other way around: It starts at the terminal element and computes the joints as it moves higher in the animation skeleton. Thus, inverse kinematics allows us to locate an element in space, and then calculate the needed configuration of its ancestor joints. Inverse kinematics is useful for adaptive animation.

After animating the skeleton structure, for the character model to move

and deform with it, we need to perform skeletal subspace deformation (SSD) to the mesh vertices. For this we need to assign each vertex of the mesh to 1 or more bones in the hierarchy with appropriate weights. As the bones moves, so do the vertices that are attached to it, and how much the vertex is influenced by each bone is determined by the corresponding weight.

3.5.2 Mathematics of Kinematics

Forward kinematics can be expressed in the form of

$$X = f(\theta) \tag{3.12}$$

where the motion of all joints is specified explicitly. The motion of the hands and feet is determined indirectly as the accumulation of all transformations that lead to them. This, for example, in the case of the character's foot, would be the combined effect of the transformations at the hip, knee and ankle. That is, given θ, derive X.

Inverse kinematics is usually called 'goal-directed motion' and can be defined in the form:

$$\theta = f^{-1}(X) \tag{3.13}$$

The animator defines the position of the end effectors like hand or foot only. Inverse kinematics solves for the position and the orientation of all joints in the link hierarchy that lead to the end effector. Given X,θ is derived.

We can illustrate the difference between the two approaches using a simple two-link structure shown in Figure 3.15. One end is fixed and both links move in the plane of the paper. The forward kinematics solution $X = (x, y)$ is given by:

$$X = (l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2), l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)) \tag{3.14}$$

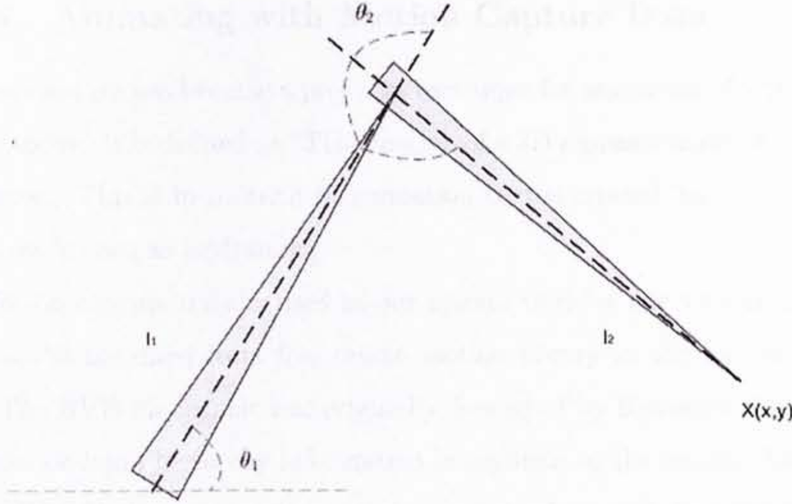

Figure 3.15: A simple 2 bone skeletal structure

The inverse kinematics solution can be obtained by applying some elementary trigonometry:

$$\theta_1 = \frac{-(l_2 \sin\theta_2)x + (l_1 + l_2\cos\theta_2)y}{(l_2\sin\theta_2)y + (l_1 + l_2\cos\theta_2 x)} \tag{3.15}$$

$$\theta_2 = \cos^{-1}\frac{(x^2 + y^2 - l_1^2 - l_2^2)}{2l_1l_2} \tag{3.16}$$

We can see that both techniques become harder to use as the complexity of the articulation increases. But the inverse kinematics will be much more complicated than forward kinematics in a large hierarchical system, and for the computation, we can find that the computation forward kinematics can be easily formed into matrix-vector multiplication, which is especially good for GPU implementation. In this project we aimed to adapting motion capture data to our virtual character, and forward kinematics technique is used as the motion data is also in the FK format.

3.5.3 Animating with Motion Capture Data

Motion capture has become a premiere technique for animation of virtual characters today. It is defined as "The creation of a 3D representation of a live performance." This is in contrast to animation that is created 'by hand' through a process known as keyframing.

Motion capture data is used in our system to drive our virtual character, the data is obtained from free online motion library in the format of BVH files. The BVH file format was originally developed by Biovision. This format provides skeleton hierarchy information in addition to the motion data.

A BVH file has two parts, a header section which describes the hierarchy and initial pose of the skeleton; and a data section which contains the motion data.

To calculate the position of a joint we first create a transformation matrix from the local translation and rotation information for that joint. For any joint the translation information will simply be the offset as defined in the hierarchy section. The rotation data comes from the motion section. Adding the offset information is simple, just put the X,Y and Z translation data into the proper locations of the matrix. Once the local transformation is created then concatenate it with the local transformation of its parent, then its grand parent, and so on. A typical example is

$$\begin{aligned}
M_{neck} &= M_{neck} \\
M_{shoulder} &= M_{shoulder} M_{neck} \\
M_{elbow} &= M_{elbow} M_{shoulder} M_{neck} \\
M_{hand} &= M_{hand} M_{elbow} M_{shoulder} M_{neck}
\end{aligned}$$

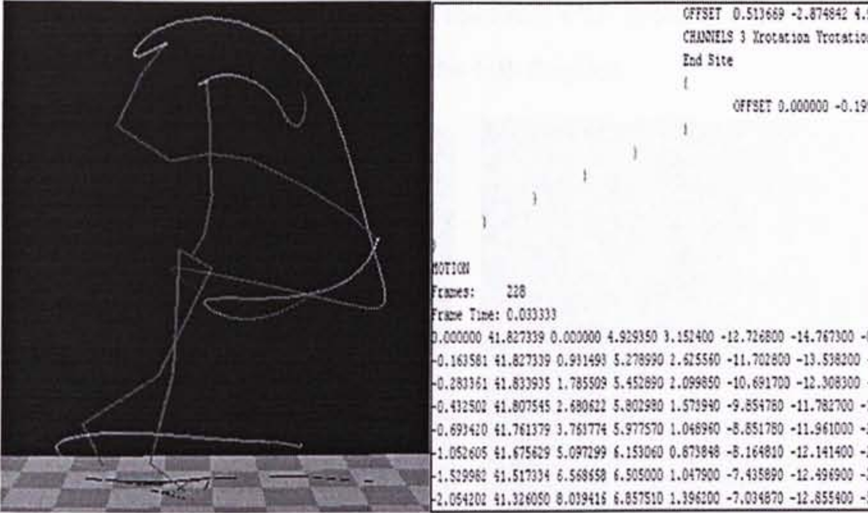

Figure 3.16: Motion capture animation using BVH data

3.5.4 Skeletal Subspace Deformation

Let β be the set of indices for all the bones, and denote the bones affecting vertex i by the subset of indices $B_i \subset \beta$. For a given skeletal configuration, with bone transformation $\{T_b\}_{b\in\beta}$, the position of the ith vertex after the skeletal subspace deformation is

$$\widetilde{v_i} = (\sum_{b\in B_i} w_{ib} T_b) v_i \tag{3.17}$$

where v_i is the position of vertex i in the neutral pose, and $w_i b$ is the weight which indicates how much the vertex is affected by this bone, it gives us the affine combination of bone transformations for this vertex. In the character's neutral pose we assume that $T_b = I, \forall b \in \beta$.

Starting with a reasonable set of bone weights is important. We compute our SSD bone weights that is inverse proportional to the vertex bone distances in the neutral pose. The following function is used for calculating the weights.

$$w_{ib} = \frac{k}{\sqrt{(v_i - J_b)^n}} \tag{3.18}$$

where J_b is the position of the joint of the bone with index b, k determines the scale of the function, and n controls the falloff speed.

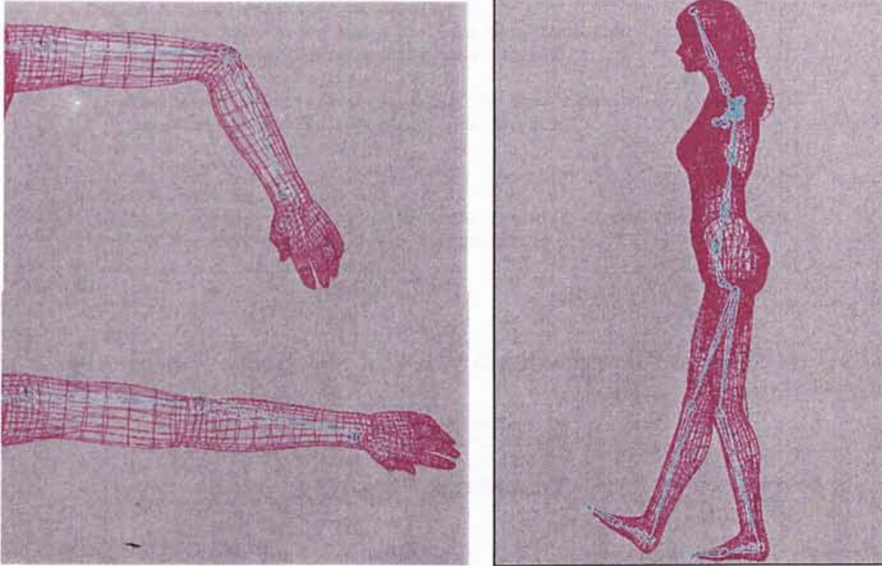

Figure 3.17: Skeletal subspace deformation

The normals are transformed almost the same way as the vertices, by the transformation matrix and using the same weights. The only different difference is that the we only use the 3×3 rotational part of the matrices.

$$\widetilde{n}_i = (\sum_{b \in B_i} w_{ib} T_{3x3,b}) n_i \tag{3.19}$$

where n_i is the normal vector to be transformed and $T_{3,b}$ are the upper left rotational part of the bone transformation matrix.

3.5.5 GPU Algorithm

Since the nature of skeletal animation algorithm is the computation for each vertex, we can also map the skeletal subspace deformation to the GPU using the vertex shader engine. The pseudocode of the algorithm is as follows:

```
Skeletal Subspace Deformation

 for each vertex in the mesh

   for each bone affecting the vertex
      transform the vertex using the bone transformation matrices
      (weighted by the corresponding bone weight)

      transform the normal vector using the bone rotation matrices
      (weighted by the corresponding bone weight)
    end

 calculate lighting
 perform the world, view,projection transformation to the vertex
 output to the next pipeline stage for rendering

 end
```

Figure 3.18: Pseudocode for skeletal subspace deformation on GPU

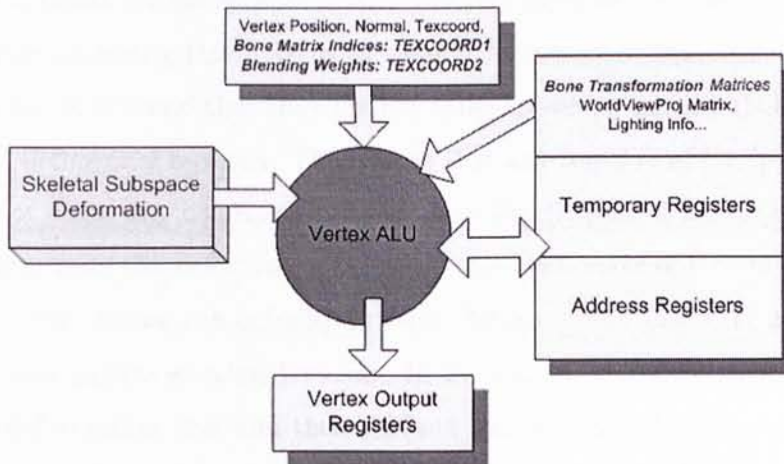

Figure 3.19: Skeletal subspace deformation on GPU

As shown in Figure 3.19, using the GPU for performing the skeletal subspace deformation is fairly efficient since the algorithm can be naturally arranged as a per vertex computation. The vertex, normal, texture coordinate data can be passed to the engine in a vertex stream format. By making use of the unused texture coordinate registers, we can pass bone index and bone weights as 2 additional texture coordinates for each vertex. For the invariant parameters that are shared among vertices like the "bone transformation matrix array", the model view projection matrix, lighting parameters can be

easily stored in the constant memory of the GPU for access by vertex shaders.

3.6 Integration of Skeletal and Facial Animation

Because of our different algorithms for the facial animation and skeletal animation computation, a solution to seamlessly integrate them must be provided. In general, our virtual character should be able to talk, sing, which are merely deformation happened to her face, and rotate its head from side to side or around, which are the skeletal motion that are controlled by the bone inside.

After analyzing this 2 kinds of motion happened to the head of virtual character, it is found that they are not coherent, or we can say that they are almost orthogonal motions. This means that any rotation of the head nearly does not cause any deformation to the face (except part of the neck), and at the same time the deformation operation does not make any rotation of the head either. So we can actually perform the operations one after another in order, and achieve seamless blending. In our system, we choose to perform the facial deformation first and then perform the skeletal subspace deformation afterwards, this is because that the DV vectors stored are in untransformed neutral directions. So the integrated facial animation algorithm is in the form of:

$$v_{face,k} = (\sum_{b \in B} w_b T_b)(v_{neutral,k} + \sum_i \alpha_i(t) DV_{i,ipa,k} + \sum_j \beta_j(t) DV_{j,exp,k}) \quad (3.20)$$

where $v_{face,k}$ is the position of vertex k in the new face, similarly, $v_{neutral,k}$ is the kth vertex in the neutral face .w_b is the weight which indicates how much the vertex is affected by this particular bone. The set of bone indexes that affect the vertex is denoted by B and the corresponding transformation matrices T_b.

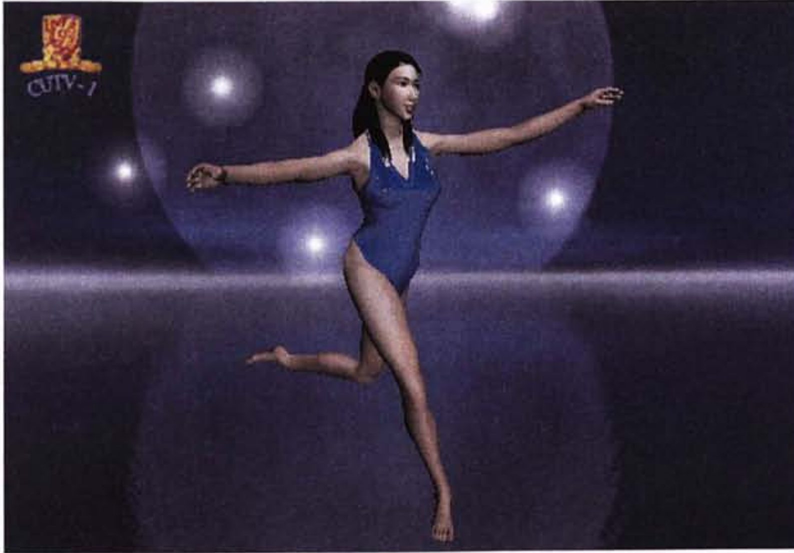

Figure 3.20: Integration of skeletal and facial animation

3.7 Result

The final rendering is composed of several layers or components – the character, the environment, and the effects.

Since the facial animation and skeletal animation vertex data stream is passed down the pipeline after the animation calculation in the vertex processor, it is easy for us to further process them in the fragment processor. The fragment shader will involve per-pixel lighting calculation or texturing or both of them, then the final fragment color is passed down for further tests before put into the frame buffer. To render the transparent objects like the hair, one more alpha channel for the texture is added for alpha blending (Figure 3.21).

The environment is simply rendered with a polygonal sphere. A fragment shader that calculate and retrieve the color from a cubemap based on the world space viewing vector is used here for environment rendering. This is a per-pixel process that performed on every rasterized pixel so as to maintain high quality rendering.

The cubemap texture is composed of 6 prerendered images, which make

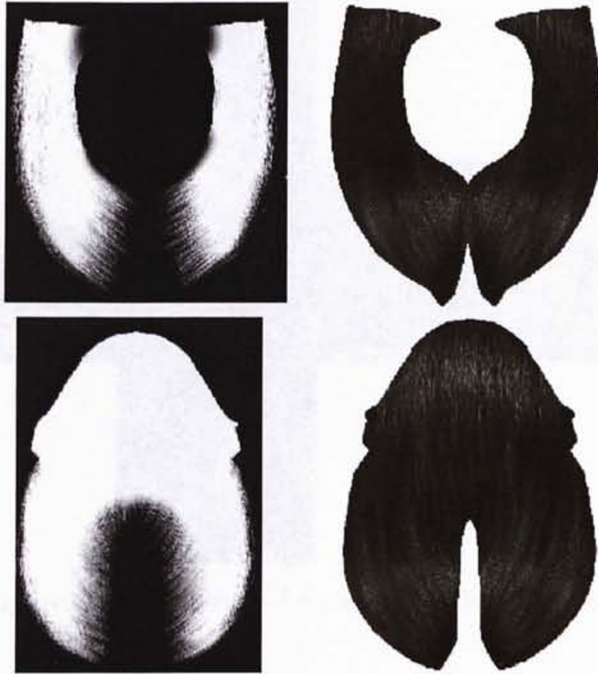

Figure 3.21: The color and alpha channel for hair

up of 6 faces of a cube (Figure 3.22). Thus it can be created by designing a scene and then render 6 images with the camera positioned at the scene center and pointing to 6 directions of the face normals of the cube, or it can be from 6 real photographs. The cubemap gives a good representation of the environment information in colors. It can be denoted as a function which returns color whenever you give the viewing direction from the scene center.

In order to render the environment correctly, we designed a vertex shader that calculated the view vector based on its world coordinate and eye coordinate. Then for per-pixel rendering using the view vector, we passed it down to the pipeline using the texture coordinate register that is not only in 32-bit high precision format, but will also enable the value to be interpolated between every pixel of the sphere that is rasterized. This allows us to render the environment correctly.

Figure 3.22: Cubemap for rendering the environment

Snow effect in the system is rendered using particle system simulation and billboard technique. Physical simulation of the particles using newtonian physics is performed first, with the position and velocity of every particle recalculated every frame, which is also known as the euler integration. The particles are then rendered using the billboard, that are polygons that always face the viewer so as to fool the eyes to treat it as a 3D object instead of a 2D plane. As the viewpoint changes, a rotation matrix is generated to rotate the particle billboards so that they face the new viewpoint. The result rendering can be seen in Figure 3.23.

We can also perform different style renderings like cartoon rendering by using the vertex and fragment shader engine.

In order to test the lip-synchronization feature of our system, We ran user perception experiments with 12 randomly generated seven-digit strings. For each digit string we generate either an audio recording of the synthesized speech in a noisy (cafeteria) environment; or a video file that augments the noisy synthesized audio with a talking face. Our tests involve 16 Cantonese-speaking

Figure 3.23: Environment and effect rendering

	Substitution	Deletion	Insertion	Accuracy
Speech Only	3.1%	14.7%	1.6%	80.6%
Speech With Animation	4.8%	3.4%	2.8%	89.0%

Table 3.7: Results of the user perception experiments.

subjects. Each subject is presented with the 12 audio/video files and asked to write down the digit string that was spoken. Subjects have no prior knowledge of the lengths of the digit strings. Table 3 shows the experimental results in terms of substitution (S), deletion (D) and insertion (I) errors, Accuracy(%)=1 - total error rates(%).

The digits '5' and '2' are pronounced in Cantonese as /ng3/ and /yi6/ respectively. These are often misrecognized due to their low energies. Furthermore, their visemes look similar - both have a slightly open lip shape. When the synthetic face is included, we observe a slight increase in substitution errors. This is caused by substitutions between '2' and '5'. The significant decrease in deletion errors is predominantly due to better perception of '5' when the viseme is included. The slight increase in substitution errors is due to the insertion of '2' at the end of the digit string - a slight smile in the talking face at the end of the utterance misled the subjects to believe that the viseme

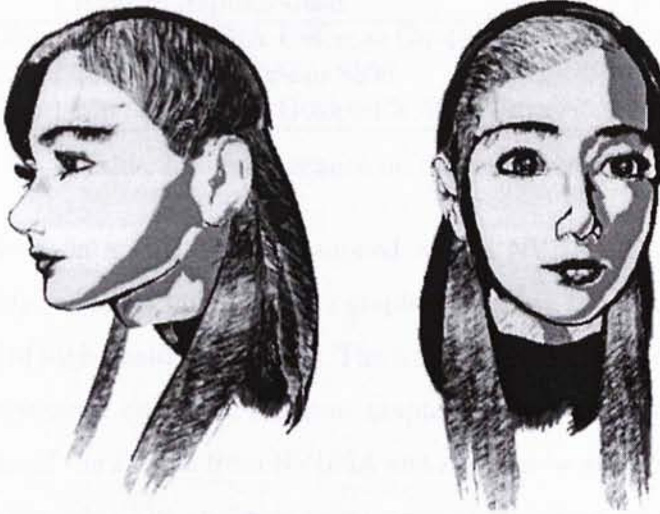

Figure 3.24: Cartoon style rendering

	Vertex	Normals	Texture Coordinates	Polygons	Texture Map
Head	4098	4098	4098	8120	2000x2000
Body	6442	6442	6442	12732	2000x2000
Legs	4850	4850	4850	9580	2000x2000
Environment	1986	(no need)	(generated in shader)	2048	512x512x6
Particles	400	(no need)	400	100	128x128

Table 3.8: Geometric and texture data statistics

for '2' was realized.

The model we used is a high resolution model with high quality textures. The data amount is shown below in Table 3.8. The textures for the virtual character are all in 2000x2000 high resolution, and the total vertices and polygons for the character model is 15390 and 30432, respectively. The texture coordinates for environment rendering is generated in the fragment shader, which will be used to fetch color form a cubemap composed 6 images of 512x512 in resolution.

To test the performance of our system, we have tested it on computer systems with different configurations and graphics cards, as shown in Table 3.9.

We can see that on the 3 computer systems we have tested, all of them achieve real-time performance, i.e, the frame rate is higher than 30 frames per

	CPU	Ram	Graphics Card	Video Ram	Frame Rate
Laptop	PIII 1.1Ghz	128	NVIDIA Geforce4 Go 440 (Mobile)	32MB	33
Desktop	PIV 2.0Ghz	256	ATI Raedeon 8500	64MB	61
Desktop	PIV 2.0Ghz	256	NVIDIA GeforceFX 5900 Ultra	256MB	98

Table 3.9: Performance on different systems

second. Even on a PIII laptop equipped with a NVIDIA Geforce4 440 Go graphics chip, which is only a mobile graphics chip, we can achieve 33 frames per second of high quality rendering. The 2 desktop PC tests are performed on the same computer only with different graphics card. The difference in GPU performance of the 2 cards from NVIDIA and ATI can be seen in the difference of 37 in frame rate. The available video memory also plays an important role for the system performance, since we try to put as many data into the video ram for fast access by the GPU, like the neutral face vertex, normal ,texture coordinates, phoneme and expression DVs, etc, if the video ram is small that we are unable to allocate enough buffers for all the data, the rest of them will still reside in the main memory and consume considerable amount of AGP bandwidth for rendering. That can also explain part of the reason why graphics card with small video ram will suffer in performance. But due to the nature of GPU programs and shared usage of the graphics card with the operating system and other applications, it is generally hard to measure how much effect the GPU clock and the video ram has on the system performance.

3.7.1 Summary

The virtual character system we developed not only provides an integrated framework for multilingual lip synchronized facial animation and skeletal animation, but also presents a good solution for performing intensive geometric data computation on GPU using vertex processor. The facial deformation can be integrated with skeletal animation seamlessly in the vertex shader program, and text-to-speech systems of different languages can also be integrated

by means of the IPA-based mapping.

For the geometric computation on GPU, the organization of input data into several streams is an important aspect. This ensures that we can map different data streams to the input vertex registers, and put other data or parameters that are constant for all the different stream elements within one frame to the constant memory or registers. Take the facial animation as example, Equation 3.9 is a good formulation of the problem in a format that can be streamlized, the IPA and expression DVs as well as the neutral face data can be easily separated to streams, and the value of the blending weight functions is stored into constant memory since they do not vary between different stream elements. This principle can be applied to other applications that require intensive geometric computation but can be parallelized.

Besides geometric data, many graphics and scientific applications also deal with large amount of imaging data set, i.e, data that is organized into 2D grid structure with several channels. Thus our next step in GPU computation is the exploration of imaging computing capabilities of fragment processor, which can perform operations on each rasterized pixel, and also have access to the large amount of texture memory. This is a nice feature of the fragment processor as the imaging data can be easily stored into the texture memory for fetch and use in pixel computation. One typical intensive pixel computation task is the multi-level discrete wavelet transform on large data or images. The succuss of applying intensive geometric computation on GPU motivates us to further explore the area of imaging computing using fragment processor. Our research on discrete wavelet transform on GPU will be described in detail in the next chapter.

Chapter 4

Discrete Wavelet Transform On GPU

4.1 Introduction

In this chapter we focus on how to efficiently design the imaging computation algorithm on GPU for multi-level discrete wavelet transform. Among those mathematical tools for multiresolution analysis, *discrete wavelet transform* (DWT) has been proved to be elegant and efficient. With the DWT, we can represent data by a set of coarse and detail values in different scales. Its locality nature facilitates the representation of high-frequency signals. With its coarse-to-fine nature, signals can also be synthesized in a progressive manner. Several wavelets applications in computer graphics have been proposed in recent years, including BRDF representation [33], environment map [34], global illumination [35], shadow representation [36], wavelet environment matting [37], progressive mesh [38], and even multiresolution video [39], etc. Besides, wavelets have been adopted as the core engine in JPEG2000 [40], the second generation of popular JPEG still image encoding.

The intensive computation of DWT due to multilevel filtering/downsampling does not cause a serious problem when the data scale is small. However, this will become a significant bottleneck in real-time applications with large data

set. Sweldens proposed an efficient implementation of DWT, known as the lifting scheme [41]. By reusing the intermediate values from previous steps, lifting achieves a high performance. Unfortunately, pure software DWT on large-scale data still cannot achieve real-time performance. This is evidenced by the software JPEG2000 implementations. The need of real-time performance has already driven several hardware implementations of DWT [42, 43].

4.1.1 Previous Works

Although hardware implementation (such as FPGA) offers real-time DWT solution, extra cost is needed for installing extra hardware. As these specialized hardware are preliminary, they are still expensive and not cost-effective. On the other hand, current generation of consumer-level graphics hardware, GPU, has already evolved to a stage that supports parallel processing, high programmability, and high-precision computation [4]. It performs not just rendering of texture-mapped polygons, but also general computations, such as sparse matrix solving [44], linear algebra operations [45], fast Fourier transform [46], and also non-linear optimization for image-based modelling [47]. Hopf and Ertl proposed a method that utilizes the specific OpenGL extensions to perform convolution and downsampling in DWT [48]. However, these extensions may vary with different graphics hardware. Hence it may be difficult to extend to different signal boundary extension schemes. Moreover, the control of "exact pixel selection" in the downsampling process is tedious.

4.1.2 Our Solution

In this paper, we propose a real-time DWT shader that runs on GPU, hence it reduces the computational burden of CPU. No tailor-made hardware nor extension is needed. Providing the shader programmability, our generic DWT solution can be trivially adapted to different wavelet transform and different

boundary extension schemes. The exact pixel selection during downsampling also comes with no extra cost. Moreover, our approach unifies both the forward and inverse DWT to an identical and simple process.

We demonstrate our real-time GPU-based DWT by first applying it to manipulate the wavelet subbands for real-time geometric deformation. Next, we show that designer can rapidly perform stylish image processing and texture-illuminance decoupling [49]. Lastly, we have also integrated our DWT engine into a well-known JPEG2000 codec, JasPer [40], and significantly improved the encoding performance, especially for high-resolution images obtained from normal digital cameras.

4.2 Multiresolution Analysis with Wavelets

Consider the Hilbert space $L^2(\Re)$ of measurable, square-integrable functions defined on real line $\Re$. A multiresolution analysis [50, 51, 52] consists of a sequence of closed subspaces $\{V^j | j \geq 0\}$, $V^j \subset L^2(\Re)$, where j denotes the resolution level. Those subspaces are in a nested manner: $V^0 \subset V^1 \subset \cdots V^j \cdots$. At each resolution level j, there exists a set of scaling functions ϕ_i^j with $i \in K(j)$, where $K(j)$ is an index set at level j such that $K(j) \subset K(j+1)$. Those scaling functions ϕ_i^j should be a Riesz basis of V^j.

Let f^j be a function in the resolution level j, *i.e.* $f^j \in V^j$. It can be expressed as a weighted sum of scaling functions ϕ_i^j:

$$f^j = \sum_{i \in K(j)} \lambda_{j,i} \phi_i^j \ , \tag{4.1}$$

where the parameters $\lambda_{j,i}$ are defined as the scaling coefficients of the function f^j. In other words, a function f^j in V^j can be represented by the scaling coefficients $\lambda_{j,i}$.

Let the subspace W^{j-1} be the orthogonal complement of V^{j-1}, such that

$V^{j-1} \oplus W^{j-1} = V^j$. There exists a set of functions $\{\psi_m^{j-1} | j \geq 0, m \in M(j-1)\}$, where $M(j-1) \subset K(j)$. Those functions should be a Riesz basis of W^{j-1}. In this case, the functions ψ_m^{j-1} define a wavelet basis. The function f^j can also expressed as:

$$f^j = \sum_{k \in K(j-1)} \lambda_{j-1,k} \phi_k^{j-1} + \sum_{m \in M(j-1)} \gamma_{j-1,m} \psi_m^{j-1} \quad (4.2)$$

because $V^{j-1} \oplus W^{j-1} = V^j$. The parameters $\lambda_{j-1,k}$ are the coarse approximations and the parameters $\gamma_{j-1,m}$ corresponds to detail subspace.

The one-step wavelet transform computes the coefficients (scaling $\lambda_{j-1,k}$ and detail $\gamma_{j-1,m}$) at level $j-1$ from the scaling coefficients $\lambda_{j,i}$ at level j:

$$\lambda_{j-1,k} = \sum_i h_{j-1,k,i} \lambda_{j,i} \quad (4.3)$$

$$\gamma_{j-1,m} = \sum_i g_{j-1,m,i} \lambda_{j,i} \quad (4.4)$$

where the parameters $h_{j-1,k,i}$ and $g_{j-1,k,i}$ are the decomposition low-pass filter parameters and decomposition high-pass filter parameters, respectively. Those filter parameters depend on the choice of the scaling functions (ϕ_i^j and ϕ_k^j) and wavelet basis ψ_m^{j-1}. In digital signal processing representation (Figure 4.1), the input sequence $x(n)$ are the scaling coefficients $\lambda_{j,i}$; and the output sequences L and H are the scaling coefficients $\lambda_{j-1,k}$ and the coarse coefficients $\gamma_{j-1,m}$, respectively.

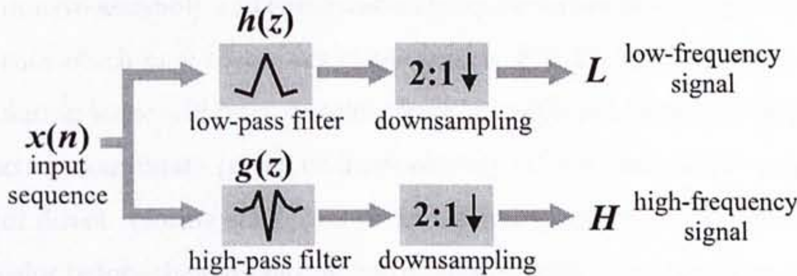

Figure 4.1: One dimensional DWT: filtering and downsampling.

The downsampling process can be illustrated by Figure 4.1, where low-pass h and high-pass g filter kernels are convolved with the 1D signal $x(n)$ to produce low-frequency and high-frequency subbands. These subbands are then downsampled. The signal $x(n)$ is decomposed into multiple frequency subbands of different scales by successively applying this filtering-and-downsampling process to the low-frequency subbands. For 2D signal, 2D separable wavelet transform can be used, *i.e.* 2D DWT can be achieved by first applying 1D DWT on the rows and then on the columns.

The one-step inverse wavelet transform computes the scaling coefficients $(\lambda_{j,i})$ at level j from the coefficients $\lambda_{j,i}$ at level $j-1$:

$$\lambda_{j,i} = \sum_{k} h'_{j-1,k,i}\lambda_{j-1,i} + \sum_{m} g'_{j-1,m,i}\gamma_{j-1,m}\,, \tag{4.5}$$

where the parameters $h'_{j-1,k,i}$ and $g'_{j-1,m,i}$ are the reconstruction low-pass filter and high-pass filter parameters, respectively.

4.3 Fragment Processor for Pixel Processing

A basic texture-mapped polygon rendering process can be illustrated with a simple example. First the user will issue 3D rendering command through 3D API in the form of polygon vertices and texture map images, then the vertices undergo a 3D transformation stage that positions it at the desired location. The vertex processor is mainly responsible for this transformation. After that, the primitive assembly and rasterization stage rasterizes the polygons into pixel fragments which may cover part of the screen. Finally, the fragment texturing and coloring stage will retrieve colors from the texture images according to the 2D texture coordinate (given or interpolated). The fetched color value can be used for direct coloring each pixel or for further computation to determine the pixel color before they are output to the frame buffer. The fragment processor is responsible for this texturing and coloring stage.

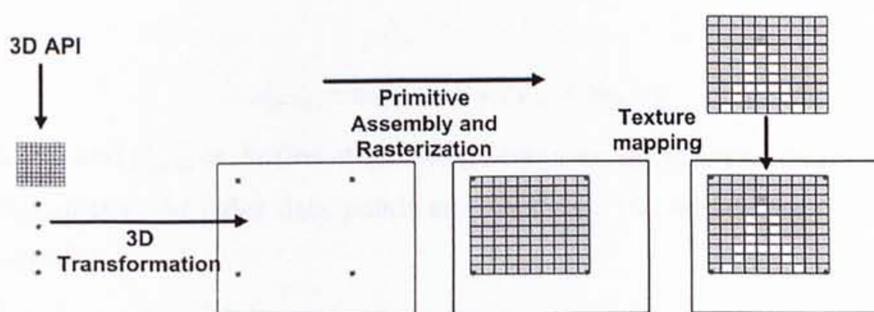

Figure 4.2: The 3D rendering pipeline

We can see that the fragment processor is responsible for texturing and coloring. With the programmability, the user can develop applications for other purposes, like here for image processing in our application, the fragment shader can be utilized.

The GPU is SIMD in nature, which means we have a bunch of vertex pipelines and fragment pipelines that run in parallel but on different vertices/fragments. The current generation GPU already supports IEEE 32-bit floating point computation that facilitates high-precision discrete wavelet transform in our application.

4.4 DWT Pipeline

4.4.1 Convolution Versus Lifting

DWT can be achieved by either the straightforward convolutional approach or the lifting scheme. The convolutional approach directly implements the filtering operation. It consumes more memory and requires more computation. On the other hand, the lifting scheme [41] implements a waveform transform by a successive simple filtering operations.

Denote the original data as $\{x_n\}, n = 1..k$, to perform the wavelet transform using lifting, we first apply the first stage lifting as follows:

$$x'_{2n+1} = x_{2n+1} + a \times (x_{2n} + x_{2n+2}) \tag{4.6}$$

where a and x'_{2n+1} is the first stage lifting parameter and outcome, respectively. After all the odd index data points are calculated, the second stage lifting is performed:

$$x''_{2n} = x_{2n} + b \times (x'_{2n-1} + x'_{2n+1}) \tag{4.7}$$

where we refer the second stage lifting parameter and outcome as b and x''_{2n}, respectively. The third and fourth stage lifting can be performed in similar ways:

$$\begin{aligned} H_n &= x'_{2n+1} + c \times (x''_{2n} + x''_{2n+2}) \\ L_n &= x''_{2n} + d \times (H_{n-1} + H_n) \end{aligned}$$

where H_n and L_n are the resultant high and low pass coefficients. This process can also be shown with Figure 4.3

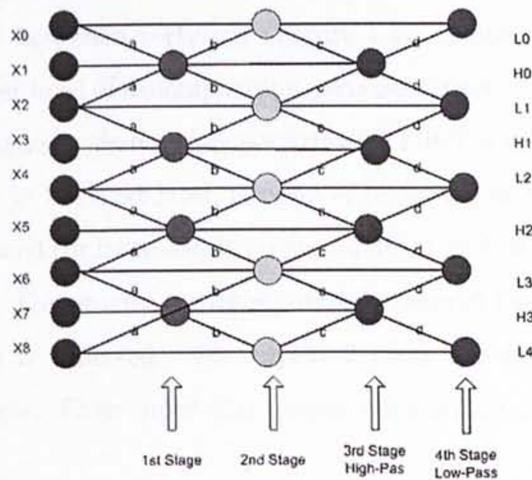

Figure 4.3: The lifting scheme

The lifting scheme consumes small memory and less computation. For software implementations, it is obvious the lifting scheme is preferred. However,

our goal is to develop a DWT engine that executes on GPU whose major advantage is its SIMD-based parallel processing. No intermediate values sharing among pixels is allowed. Lifting implicitly imposes an order of execution which is not fully parallelizable. As the intermediate value sharing is the key factor of lifting to reduce computation, it will cause too many passes and hence the switching of rendering context in GPU implementation. Note that the rendering context switching in GPU introduces large overhead on current GPU design. All these suggest that the parallelizable convolutional approach is favorable. The large memory consumption issue may not be a serious problem as large and extremely fast video memory is available on consumer-level GPU.

4.4.2 DWT Pipeline

We design the DWT engine as a set of fragment shaders which perform DWT on IEEE 32-bit floating-point image. The whole multi-level transform consists of rendering carefully aligned quadrilaterals that cover the whole or part of the image. At each level, the 2D DWT is achieved by performing 1D DWT first horizontally and then vertically (Figure 4.4). Hence the quadrilateral is rendered twice per level of decomposition/reconstruction. At each output pixel (fragment), a fragment shader that performs 1D DWT is executed to compute the convolution. In the next level, number of pixels requiring fragment shader execution is reduced (or increased in reconstruction) by 4 so as to cover the low-passed subband. The process continues until the desired level of decomposition / reconstruction is achieved. We employ 2 pixel buffers to hold the input and output images. Their input and output roles are interchanged after each rendering pass.

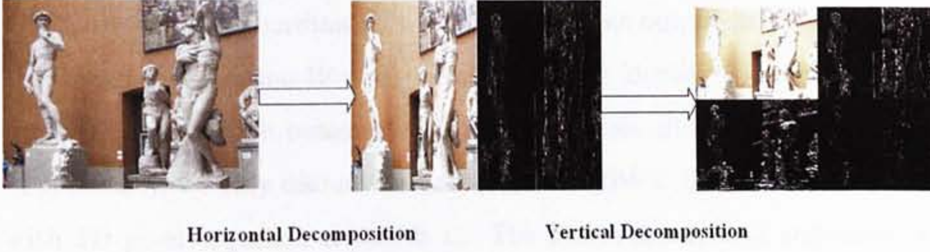

Figure 4.4: Separable 2D DWT

4.5 Forward DWT

Let $\{\lambda'_j(n)\}$ be the boundary-extended input signal at level j. After 1–D DWT and downsampling, the low– and high–pass sequences are given by

$$\lambda_{j-1}(n) = \sum_k h(k)\lambda'_j(2n-k) \quad (4.8)$$

$$\gamma_{j-1}(n) = \sum_k g(k)\lambda'_j(2n+1-k) \quad (4.9)$$

Let $\{z_{j-1}(n)\}$ be the concatenation of $\{\lambda_{j-1}(n)\}$ and $\{\gamma_{j-1}(n)\}$, We can rewrite (4.8) and (4.9) in a more generic way for efficient SIMD implementation on GPU

$$z_{j-1}(n) = \sum_k f_{d,j-1}(n,k) f_{\lambda,j}(n,k), \quad (4.10)$$

$f_{d,j-1}(n,k)$ is a position-dependent filter that selects the proper coefficient from $h(k)$ and $g(k)$ at decomposition level j-1. $f_{\lambda,j-1}(n,k)$ is a function that returns the corresponding data in the level j boundary-extended signal $\{\lambda'(n)\}$ for convolution. These can be easily implemented by indirect addressing / dependent texture fetching [11] in the GPU algorithm.

The $f_{d,j-1}(n,k)$ position dependent filter is achieved by a *filter selector* variable α calculated in the fragment shader to selects between $h(k)$ and $g(k)$, i.e.,at each output pixel, the fragment shader has to determine whether the current pixel belongs to the high-passed or low-passed regions after DWT. If it is high-passed pixel, the high-pass filter kernel is used for convolution, and vice versa. Care must be taken when handling images with odd dimension.

Given the texture coordinate (s, t) for the current output pixel (fragment) in the image of resolution $W \times H$, we can uniquely identify whether the current pixel belongs to high-passed or low-passed regions after DWT. Without loss of generality, we only discuss the horizontal 1D DWT. Suppose we are dealing with 1D pixel sequence of length L. The filter selector α is computed with value 1 means high-pass and value 0 means low-pass.

$$\alpha = \begin{cases} 1, & \text{if} \quad sW > L/2 \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

Then the corresponding kth filter coefficient is used for convolution.

Function $f_{\lambda,j-1}(n, k)$ is implemented by calculating the base position(filtering center) β in the fragment shader and then fetch its neighbor from input texture as shown in Figure 4.6. The base position β can be computed by the following equation.

$$\beta = 2(s - \alpha\lceil\frac{L}{2}\rceil) + \alpha + 0.5 \tag{4.12}$$

We add 0.5 to address the pixel center in texture fetching. Figure 4.6(a) links the computed base position in the input buffer with the corresponding output pixel in the output buffer. With this, downsampling is automatically achieved without wasting computation on unused samples.

The convolution takes place with the base pixel at the center and fetches $\pm\lceil\frac{k}{2}\rceil$ neighboring pixels. In general, low-pass and high-pass filter kernels usually have different lengths, hence different values of k. For implementation uniformity, k should be chosen as the larger one.

If the fetching of neighbors goes beyond the image boundary of the current level, we need to extend the boundary extension. Common extension schemes include periodic padding, symmetric padding and zero padding, etc. We have applied symmetrical periodic extension [53] that mirrors pixels across the boundary, with the boundary pixel not mirrored. Figure 4.6(b) shows two examples of convolution with fetching to extended neighbors.

Instead of computing α, β, and boundary extension within the fragment shader using arithmetic instructions, we use a more efficient way which pre-computes and stores all these values in a 2D texture. Most GPUs nowadays support highly optimized texture fetching operations which are much faster than arithmetic operations. This table-lookup approach also offers flexibility in implementing different boundary extension schemes by replacing the addresses in this indirect address table (texture).

2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	12	11
2	1	0	1	2	3	4	5	6	5	4							
2	1	0	1	2	3	2	1										

Figure 4.5: The indirect address table for boundary extension.

The texture is organized with each row holding boundary extension, α and β values for one particular level of DWT. Inside each texel, channel R stores the indirect address of pixel with boundary extended. Channels G and B store α and β respectively. Therefore the width of table for a data sequence with maximum length L is $L + k - 1$. Figure 4.5 shows three levels of indirect addresses stored in the texture with data sequence of length 14 and $k = 5$. Color dark grey indicates the boundary-extended elements while color light grey indicates elements within the level of data sequence. This texture is small in size as the number of rows equals to $\log_2(\max(W, H))$.

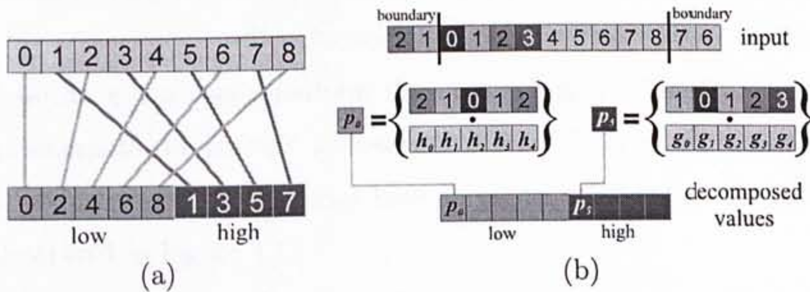

Figure 4.6: (a) Mapping to the base positions. (b) Decomposition with boundary extension.

4.6 Inverse DWT

The 2D inverse DWT can be achieved by applying 1D inverse DWT horizontally and then vertically. Although the inverse DWT is mathematically different from the forward one, we show that, by using the same indirect address table, the inverse DWT reduces to almost the same process as forward DWT. Both low-frequency and high-frequency coefficients contribute to reconstruction process.

Let $\{\lambda'_{j-1}(n)\}$ and $\{\gamma'_{j-1}(n)\}$ be the zero-padding upsampled and boundary extended low– and high–pass signal at level $j-1$. The reconstruction of $\{\lambda_j(n)\}$ is given by

$$\lambda_j(n) = \sum_k h'(k)\lambda'_{j-1}(n-k) + \sum_k g'(k)\gamma'_{j-1}(n-k)\,, \tag{4.13}$$

where $h'(k)$ and $g'(k)$ are low– and high–pass reconstruction filters, respectively.

Similar to the forward DWT, (4.13) can be rewritten as

$$\lambda_j(n) = \sum_k f_{r,j-1}(n,k) f_{z,j-1}(n,k), \tag{4.14}$$

$f_{z,j-1}(n,k)$ returns the corresponding data in the upper-sampled and boundary-extended signal of $\{z(n)\}$ at level $j-1$.This is efficiently implemented into fragment shader as shown in Figure 4.7. It shows that both low-frequency and high-frequency coefficients are upsampled and interleaved *virtually*. Note that we do not actually perform the upsampling nor interleaving. Instead, we precompute the indirect addresses and store them in the indirect address table. Note that the boundaries have to be extended before the interleaving as illustrated in Figure 4.7.

Once the indirect address table is ready, values in the next level can be reconstructed by convolution (Figure 4.8). Based on the odd/even status of position of the reconstructing pixel, we decide the reconstruction filter to convolve.

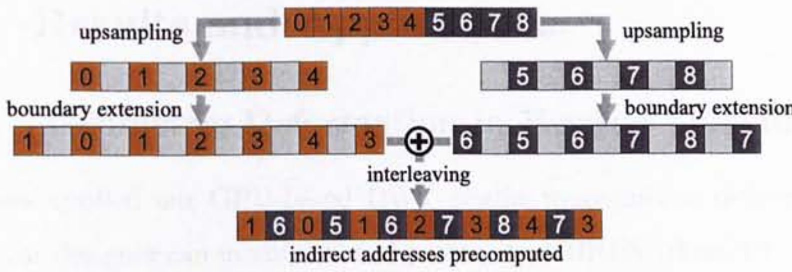

Figure 4.7: Virtual upsampling and interleaving for precomputing indirect addresses.

Note that low-frequency elements must be multiplied to the low-pass reconstruction filter, h', while high-frequency elements must be multiplied to high-pass reconstruction filter, g'. Here the position-dependent filter $f_{r,j-1}(n,k)$ is similar to the forward case which selects the proper coefficient from the reconstruction filter bank $h'(k)$ and $g'(k)$. For efficiency, we reorganize the filter kernels to form the interleaved kernels as illustrated in Figure 4.8. In general, h' and g', are different in length. With this indirect addressing, both the shaders for forward and inverse DWTs are basically performing the same operations, namely indirect pixel lookup, filter selection, and convolution.

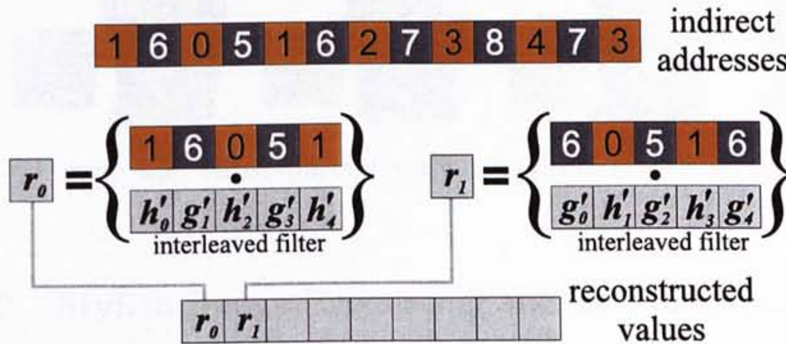

Figure 4.8: Reconstruction filtering in inverse DWT.

4.7 Results and Applications

4.7.1 Geometric Deformation in Wavelet Domain

We have applied our GPU-based DWT engine to geometric deformation in which the designer can modify control points of a NURBS 3D model in wavelet domain. The designer can arbitrarily scale the wavelet coefficients in different frequency subbands to achieve the desired effect. Note that the deformation is done in real-time.

Figure 4.9 shows three deformed heads along with the scaling configurations of wavelet subbands. The subband with no scaling is color-coded in grey. The subbands being scaled up and down are color-coded in red and blue respectively. As coefficients in different subband influence the geometry in different scales, the designer can focus on the semantic rather than the spatial position of control points.

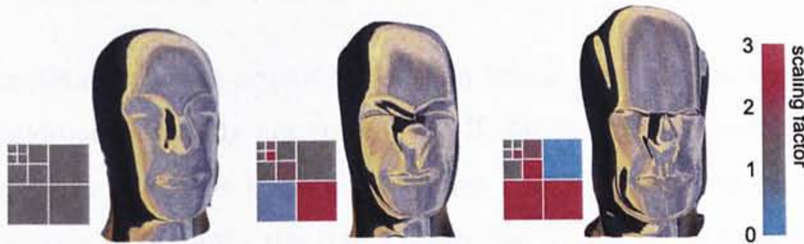

Figure 4.9: Three different wavelet-based geometric designs.

4.7.2 Stylish Image Processing and Texture-illuminance Decoupling

Our GPU-based DWT engine allows us to do real-time wavelet-based multiresolution image processing which offers various effects, even images are of high resolution. Hence, we combine, remove, or scale the wavelet coefficients in different subbands and in different color spaces (RGB or YUV) to achieve the desired effects. As illustrated in Figure 4.10, the bumpy feature of the bean

image (middle) is transferred to the Starry Night painting (left) by combining the high-frequency subbands in the Y channel of both images while removing the lowest frequency subband of the bean image. We take the maximum between two corresponding coefficients to maintain the details from both images. The real-time ability of our wavelet transform allows the designer to rapidly evaluate the visual results from wavelet domain processing.

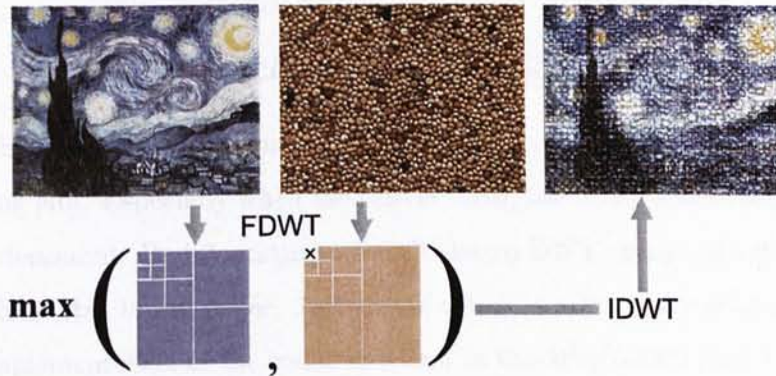

Figure 4.10: Fast image stylishing by combining coefficients in wavelet domain.

Sometimes when we acquire textures by taking photographs, the illumination condition is usually not controlled. If the illumination only introduces slow intensity change in the acquired image (*i.e.* low-frequency component), it is possible to decouple the contribution due to the uncontrolled illumination from the desired texture. Figure 4.11 illustrates such application. We first remove high-frequency subbands and generate the illuminance image using inverse DWT. By dividing the original image with this illuminance image, we obtain an "illumination-constant" decal map which is ready for texture mapping.

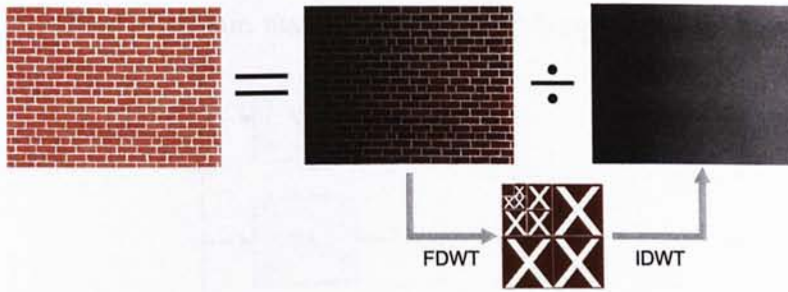

Figure 4.11: Fast texture-illuminance decoupling.

4.7.3 Hardware-Accelerated JPEG2000 Encoding

DWT has been identified as a time-consuming part of the lossy JPEG2000 encoding [40]. Especially when the rate is small, the DWT processing time becomes dominant. By integrating our GPU-based DWT engine into the popular JPEG2000 still image codec, JasPer [40],which is a free software-based reference implementation of the codec specified in the JPEG-2000 Part-1 standard (i.e., ISO/IEC 15444-1), the encoding speed has been greatly increased.

The flow of a JPEG2000 encoder can be shown in Figure 4.12. The first part is the component and tile separation, which is used to cut the image into chunks and to decorrelate the color components. For multi-component color images, a component transform is performed to decorrelate the componenets.Each tile of each component is then processed separately. The data are first transformed into the wavelet domain, and are then quantized. After that, the quantized coefficients are regrouped to facilitate localized and resolution access. Each subband of the quantized coefficients is divided into rectangular blocks. Three spatially co-located rectangles (one from each subband at a given resolution level) form a packet partition. Each packet partition is further divided into code-blocks, each of which is compressed into an embedded bitstream. They are then assembled into packets, each of which represents a quality increment of one resolution level at one spatial location. By combining packets from all the partitions of all resolution level of all the tiles and components, we form a

layer. The final bitstream may contain multiple layers.

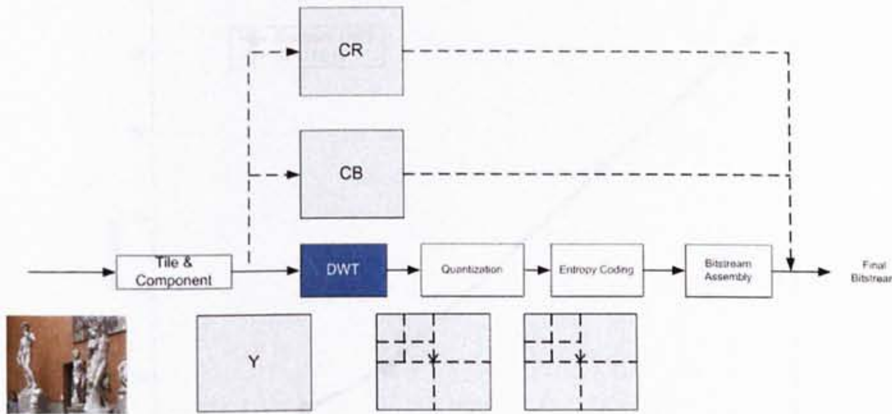

Figure 4.12: JPEG2000 encoding flow

The most important part of the codec is the DWT shown in Figure 4.12, in JPEG2000, both reversible integer to integer and non-reversible real-to-real wavelet transforms are employed, called the 5/3 and 9/7 wavelet kernel, respectively. Since the main performance issue is in the 9/7 real-to-real wavelet transform, and at the same time the GPU processes floating point values throughout the graphics pipeline, we have integrated our GPU-based DWT specially for the 9/7 wavelet into the JasPer codec.

We have evaluated the performance of the GPU-based DWT. The implementation adopts OpenGL and NVIDIA's Cg for shader development. The evaluation is conducted on a PC with Pentium IV 2.0 GHz CPU, 512MB memory, and GeforceFX 5900 Ultra with 256MB video memory.

We compare the execution time of GPU-based DWT with the software lifting-based Jasper. Seven test images ranging from 128×128 to 2048×2048 are encoded with 1 tile, 1 layer, 9/7 transform, 5 decomposition levels. Figure 4.13 shows the timing statistics that compares the original software codec with ours. Instead of measuring the image dimension, we measure the number of pixels in the unit of *million pixels* which is commonly used in digital camera terminology. For low-resolution images, the speed of the GPU DWT is a bit

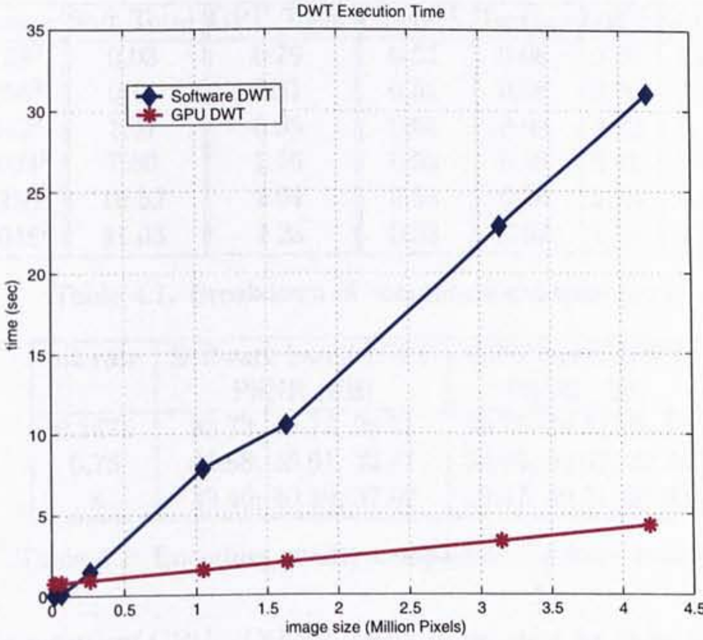

Figure 4.13: Timing Comparison: software versus GPU DWT

poorer than the software one because of the overhead of GPU initialization and data transfer. As the image size is raised to around 0.16 million pixels (about 400 × 400). Our codec outperforms the software one. The speedup is apparent for encoding high-resolution images. This shows the advantage of parallel processing in SIMD-based GPU.

For a fair comparison, we have accounted for all overheads of using GPU, these include data conversion, initialization and texture creation, etc. The breakdown of execution time is shown in Table 4.1. The computation time due to overheads is included in the column "GPU Total" of Table 4.1. The breakdown is shown and the timing for software lifting implementation of DWT with JasPer ("Software Total") is also presented for comparison. "OpenGL" shows the time for initializing OpenGL and Cg systems, which is more or less constant. "Texture" accounts for the time for texture and pixel buffer creation. Hence it increases as the data size increases. The computation time for DWT occupies only a small portion of the whole execution time due to the parallel

Image	Soft Total	GPU Total	OpenGL	Texture	DWT	Others
128^2	0.03	0.76	0.53	0.06	0.16	0.01
256^2	0.11	0.81	0.54	0.08	0.16	0.03
512^2	1.47	0.95	0.51	0.09	0.22	0.13
1024^2	7.80	1.56	0.53	0.19	0.41	0.43
1280^2	10.52	2.04	0.54	0.30	0.59	0.61
2048^2	31.03	4.28	0.53	0.67	1.38	1.70

Table 4.1: Breakdown of computational time (sec)

bit rate	Software based DWT PSNR (dB)	GPU based DWT PSNR (dB)
0.1875	30.73, 29.72, 28.71	30.75, 29.73, 28.69
0.75	35.68, 35.01, 32.47	35.65, 35.02, 32.46
3	39.46, 40.30, 37.02	39.47, 40.31, 37.03

Table 4.2: Encoding quality comparison for lossy coding

processing nature of GPU. "Others" refers to the time for other operations as well as data conversion and transfer, therefore it is also dependent on the data sizes.

Besides, we have also evaluated the image quality by experimenting on the standard test image, Lena. We encode it at different bit rates. The PSNR in the RGB channels are shown in Table 4.2. There is no significant difference between our GPU DWT and the software lifting-based Jasper, thank to the high-precision floating point computation in GPU.

4.8 Web Information

More information and demo programs are available at:

http://www.cse.cuhk.edu.hk/~ttwong/software/dwtgpu/dwtgpu.html

Chapter 5

Conclusion

In order to exploit the SIMD and parallel processing power of GPU for the processing of increasing amount of data in computer graphics and imaging applications. We investigate 2 computational intensive problem for developing efficient GPU algorithms – the virtual character system with lip synchronization and the discrete wavelet transform, that are mainly focused on exploring the geometric and imaging computation capabilities of GPU, respectively.

In order to explore techniques and solutions for performing large geometric data computation using vertex processor, we have designed a simple and efficient framework for real-time virtual character animation. By carefully design the algorithm, we successfully map both the facial and skeleton animation computation to the GPU and achieved seamless integration of them in the shader program. By doing so, we have achieved good load balancing between the GPU and CPU, high system performance, as well as high rendering quality. The language and lip synchronization capabilities of the system is also explored. We proposed an IPA-based mapping technique to make our virtual character multilingual. Text-to-speech systems of different languages can also be easily integrated for real-time speech animation generation in our system.

For the imaging data computation on GPU, we have also demonstrated a simple but powerful and cost-effective solution to implement discrete wavelet transform on the fragment processor. No tailor-made and expensive DWT

hardware is needed to achieve such performance. It can be implemented on any SIMD-based GPU comes with normal configuration of PCs. The proposed method unifies the mathematically-different forward and inverse DWT. Different wavelet filter kernels and boundary extension schemes can be easily incorporated by modifying the filter kernel values and indirect address table respectively. We have demonstrated its applicability in real-time wavelet-based geometric deformation, stylish image processing, texture-illuminance decoupling, and JPEG2000 encoding. The current approach is still rectilinear in nature and not applicable to spherical wavelet transform [33] which is useful in modeling BRDF and environment map. In the future, we will investigate the parallelization of spherical wavelet transform on GPU.

General purpose computing on GPUs is an area of research that I find very interesting, the low cost, high speed and parallel pipelines of GPUs make them a very useful coprocessors for intensive computation in different scientific and engineering fields. Further GPUs will have to remove some limitations for achieving more wide usage. One of them will be the support for double precision throughout the pipeline, although the IEEE 32-bit single precision is generally enough in many applications, still there're computations in research work that need double precision. Another limitation is the access of texture resources in the vertex processor, removing this will allow for large data access in the vertex shader programs and also more potential applications for geometric computation on GPU. At last, one great potential change to the GPU programming paradigm will probably be the integration of vertex and fragment shader. Overall, with the increasing in speed and programmability, I see a bright future for GPU computing in graphics, scientific and engineering areas.

Bibliography

[1] E. Lindholm, M. Kilgard, and H. Moreton, A user programmable vertex engine, in *Proceedings of SIGGRAPH*, 2001.

[2] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, Interactive multi-pass porgammable shading, in *Proceedings of SIGGRAPH*, 2000.

[3] K. Proodfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan, A real-time procedural shading system for programmable graphics hardware, in *Proceedings of SIGGRAPH*, 2001.

[4] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, Cg: A system for programming graphics hardware in a c-like language, in *ACM Transactions on Graphics (TOG)*, 2003.

[5] C. Trendall and A. J. Steward, General calculations using graphics hardware with applications to interactive caustics, in *Proceedings of the Eurographics Workshop on Rendering*, 2000.

[6] J. Rhoades et al., Real-time procedural textures, in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 1992.

[7] M. Olano and A. Lastra, A shading language on graphics hardware: The pixelflow shading system, in *Proceedings of SIGGRAPH*, 1998.

[8] B. Cabral, N. Cam, and J. Foran, Accelerated volume rendering and tomographic reconstruction using texture mapping hardware, in *Proceedings of the 1994 Symposium on Volume Visualization*, 1994.

[9] J. Kniss, S. Premoze, C. Hansen, and D. S. Ebert, Interactive translucent volume rendering and procedural modeling, in *Proceedings of IEEE Visualization 2002*, 2002.

[10] N. A. Carr, J. D. Hall, and J. C. Hart, The ray engine, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002.

[11] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, Ray tracing on programmable graphics hardware, in *ACM Transactions on Graphics (TOG)*, 2002.

[12] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, Photon mapping on programmable graphics hardware, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003.

[13] N. A. Carr, J. D. Hall, and J. C. Hart, GPU algorithms for radiosity and subsurface scattering, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003.

[14] R. Strzodka and M. Rumpf, Level set segmentation in graphics hardware, in *Proceedings of the International Conference on Image Processing*, 2001.

[15] A. E. Lefohn, J. Kniss, C. Hanson, and R. T. Whitaker, Interactive deformation and visualization of level set surfaces using graphics hardware, in *Proceedings of IEEE Visualization*, 2003.

[16] K. E. Hoff, A. Zaferakis, M. Lin, and D. Manocha, Fast and simple 2d geometric proximity queries using graphics hardware, in *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 2001.

[17] N. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, Cullide: Interactive collision detection between complex models in large environments using graphics hardware, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003.

[18] N. H. Mustafa, E. Koutsofios, S. Krishnan, and S. Venkatasubramanian, Hardware assisted view dependent map simplification, in *Proceedings of the 17th Annual Symposium on Computational Geometry*, 2001.

[19] S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian, Hardware-assisted computation of depth contours, in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002.

[20] F. Parke, Computer generated animation of faces, in *Proceedings of the ACM National Conference*, 1972.

[21] F. I. Parke and K. Waters, *Computer Facial Animation*, A. K. Peters, 1996.

[22] I. Karlsson, A. Faulkner, and G. Salvi, SYNFACE – a talking face telephone, in *Proceedings of the Eurospeech*, 2003.

[23] T. E. T. Poggio, International Journal of Computer Vision **38**, 45 (2000).

[24] C. Pelachaud, E. Magno-Caldognetto, C. Zmarich, and P. Cosi, Modelling an italian talking head, in *Proceedings of Audio-Visual Speech Processing*, 2001.

[25] N. Magnenat-Thalmann, R. Laperriere, and D. Thalmann, Joint dependent local deformations for hand animation and object grasping, in *Proceedings of Graphics Interface*, 1988.

[26] K. Singh and E. Kokkevis, Skinning characters using surface-oriented free-form deformations, in *Proceedings of Graphics Interface 2000*, 2000.

[27] J. P. Lewis, M. Cordner, and N. Fong, Pose space deformations: A unified approach to shape interpolation and skeleton-driven deformation, in *Proceedings of SIGGRAPH 2000*, 2000.

[28] J. Wilhelms and A. V. Gelder, Anatomically based modeling, in *Proceedings of SIGGRAPH 1997*, 1997.

[29] F. Scheepers, R. E. Parent, W. E. Carlson, and S. F. May, Anatomy-based modeling of the human musculature, in *Proceedings of SIGGRAPH 1997*, 1997.

[30] J. Gourret, N. Magnenat-Thalmann, and D. Thalmann, Simulation of object and human skin deformations in a grasping task, in *Proceedings of SIGGRAPH 1989*, 1989.

[31] N. Magnenat-Thalmann and D. Thalmann, Human body deformations using joint–dependent local operators and finite element theory, in *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, 1991.

[32] S. Dipaola, Investigating face space, in *Proceedings of SIGGRAPH 2000*, 2000.

[33] P. Schröder and W. Sweldens, Spherical wavelets: Efficiently representing functions on the sphere, in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 1995.

[34] P. Schröder and W. Sweldens, Spherical wavelets: Texture processing, in *Rendering Techniques '95, Proceedings of the Eurographics Workshop in Dublin, Ireland, June 12-14, 1995*, pages 252–263, Springer, 1995.

[35] P. H. Christensen, E. J. Stollnitz, D. H. Salesin, and T. D. DeRose, Global illumination of glossy environments using wavelets and importance, in *ACM Transactions on Graphics (TOG)*, 1996.

[36] R. Ng, R. Ramamoorthi, and P. Hanrahan, All-frequency shadows using non-linear wavelet lighting approximation, in *ACM Transactions on Graphics (TOG)*, 2003.

[37] P. Peers and P. Dutré, Light fields and matting: Wavelet environment matting, in *Proceedings of the 13th Eurographics workshop on Rendering*, 2003.

[38] A. Khodakovsky, P. Schröder, and W. Sweldens, Progressive geometry compression, in *Computer Graphics (SIGGRAPH 2000 Proceedings)*, pages 271–278, 2000.

[39] A. Finkelstein, C. E. Jacobs, and D. H. Salesin, Multiresolution video, in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996.

[40] M. D. Adams and F. Kossentini, Jasper: A software-based JPEG-2000 codec implementation, in *Proceedings of IEEE ICIP*, 2000.

[41] W. Sweldens, SIAM Journal on Mathematical Analysis **29**, 511 (1998).

[42] K. Andra, C. Chakrabarti, and T. Acharya, IEEE Transactions on Signal Processing **50**, 966 (2002).

[43] C.-T. Huang, P.-C. Tseng, and L.-G. Chen, Hardware implementation of shape-adaptive discrete wavelete transform with the JPEG defaulted (9,7) filter bank, in *Proceedings of ICIP 2003*, 2003.

[44] J. Bolz, I. Farmer, E. Grinspun, and P. Schreoder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, in *ACM Transactions on Graphics (TOG)*, 2003.

[45] J. Krüger and R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, in *ACM TOG*, 2003.

[46] K. Moreland and E. Angel, The FFT on a GPU, in *Proceedings of HWWS*, 2003.

[47] K. E. Hillesland, S. Molinov, and R. Grzeszczuk, Nonlinear optimization framework for image-based modeling on programmable graphics hardware, in *ACM TOG*, 2003.

[48] M. Hopf and T. Ertl, Hardware accelerated wavelet transformations, in *Proceedings of EG/IEEE TCVG Symposium on Visualization*, 2000.

[49] B. M. Oh, M. Chen, J. Dorsey, and F. Durand, Image-based modeling and photo editing, in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 433–442, 2001.

[50] C. K. Chui, *An Introduction to Wavelets*, Academic Press, 1992.

[51] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin, *Wavelets for Computer Graphics, Theory and Applications*, Morgan Kaufmann Publishers, Inc, 1996.

[52] I. Daubechies, *Ten Lectures on Wavelets*, SIAM, 1992.

[53] G. Strang and T. Nguyen, *Wavelets and Filter Banks*, Wellesley-Cambridge, Cambridge, MA, 1996.

CUHK Libraries
004144772