THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

XNA-like 3D Graphics Programming on the Raspberry Pi

Lichao Wang

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

March 2014

Abstract

The Raspberry Pi is a credit-card sized computing device created by Broadcom in 2012. This device is a kind of mini PC, and it is capable of doing things that desktop PC can do. The goal of the Raspberry Pi Foundation is to allow people all over the world to learn programming. Therefore, the Raspberry Pi is designed as a small sized, low cost device that can provide reasonable data processing capability. However, because of its goal is to keep the price down to maximize openness for learning, Raspberry Pi can only run the Linux operating system.

XNA is a set of libraries developed by Microsoft to facilitate the creation and management of video games. It provides a large number of underlying functions to help the development of systems that based on runtime. Therefore, programmers may focus on programming their own code. XNA is built on Microsoft's .NET framework, and it is designed to be used with DirectX. However, as no drivers are developed to provide the low level API defined by DirectX on Linux, it is currently impossible to program with XNA on a Raspberry Pi.

This thesis investigates the possibility of developing XNA like programs directly on the Raspberry Pi. Instead of using DirectX, OpenGL ES is used to provide the low level graphics APIs. The code of a project named "JBBRXG11", which is an open source project extending XNA classes on Windows to access DirectX 10 and DirectX 11 graphics features is used as a reference for this project.

The project successfully built a library that allows an XNA like program to produce moving, textured 3D models on screen.

Acknowledgements

Firstly, I would like to thank my supervisor, Bill Rogers, for his endless support in this project.

I would also like to thank James Boud for his previous work that contribute to the complete of this project.

Finally, I would like to thank my family and friends who supported me over the research of this project.

Contents

Abstract ii

Acknowledgements.....iii

Contents.....iv

Chapter 1: Introduction.....1

Chapter 2: Literature Review.....6

2.1 MonoGame.....6

2.2 JBBRXG11.....7

Chapter 3: Underlying System.....9

3.1 XNA.....9

3.1.1 XNA Core Classes.....10

3.1.2 XNA Graphics Classes.....13

3.1.3 XNA Mathematical Classes.....16

3.1.4 Shader.....19

3.1.5 XNA Inputs.....20

3.2 OpenGL ES.....21

3.2.1 EGL.....24

3.2.2 Shaders and GLSL.....25

3.3 JBBRXG11.....27

3.4 Raspbian.....29

Chapter 4: Development.....31

4.1 C++ code with OpenGL ES 2.0 on Windows.....32

4.1.1 OpenGL ES 2.0 Sample Program.....32

4.1.2 C++ Blank Window.....42

4.1.3 C++ Triangle.....44

4.1.4 C++ Colored Rotating Cube.....46

4.1.5 C++ Textured Cube.....52

4.2 C# Code with OpenGL ES 2.0 on Windows.....58

4.2.1 C# Blank Window.....59

4.2.2 C# Triangle.....62
4.2.3 C# Colored Rotating Cube.....63
4.2.4 C# Textured Cube.....63
4.3 C# Programs on Raspbian.....64
4.3.1 C# Blank Window on Raspbian.....65
4.3.2 C# Triangle on Raspbian.....67
4.3.3 C# Colored Rotating Cube on Raspbian.....68
4.3.4 C# Textured Cube on Raspbian.....70
4.4 Rewriting XNA Classes.....73
4.4.1 Display a Blank Window with Modified XNA Classes.....73
4.4.2 Display a Colored Triangle with Modified XNA Classes.....87
4.4.3 Display a Colored Rotating Cube with Modified XNA Classes.....102
4.4.4 Display a Textured Lighting Cube with Modified XNA Classes.....109

Chapter 5: Conclusion and Future Work.....120

Appendix.....122
References.....131

Chapter 1: Introduction

The purpose of the project Pi-XNA is making it possible for people to learn 3D graphics programming on Raspberry Pi. The Raspberry Pi is a credit card sized single board, open computer that can be used with a monitor and keyboard.

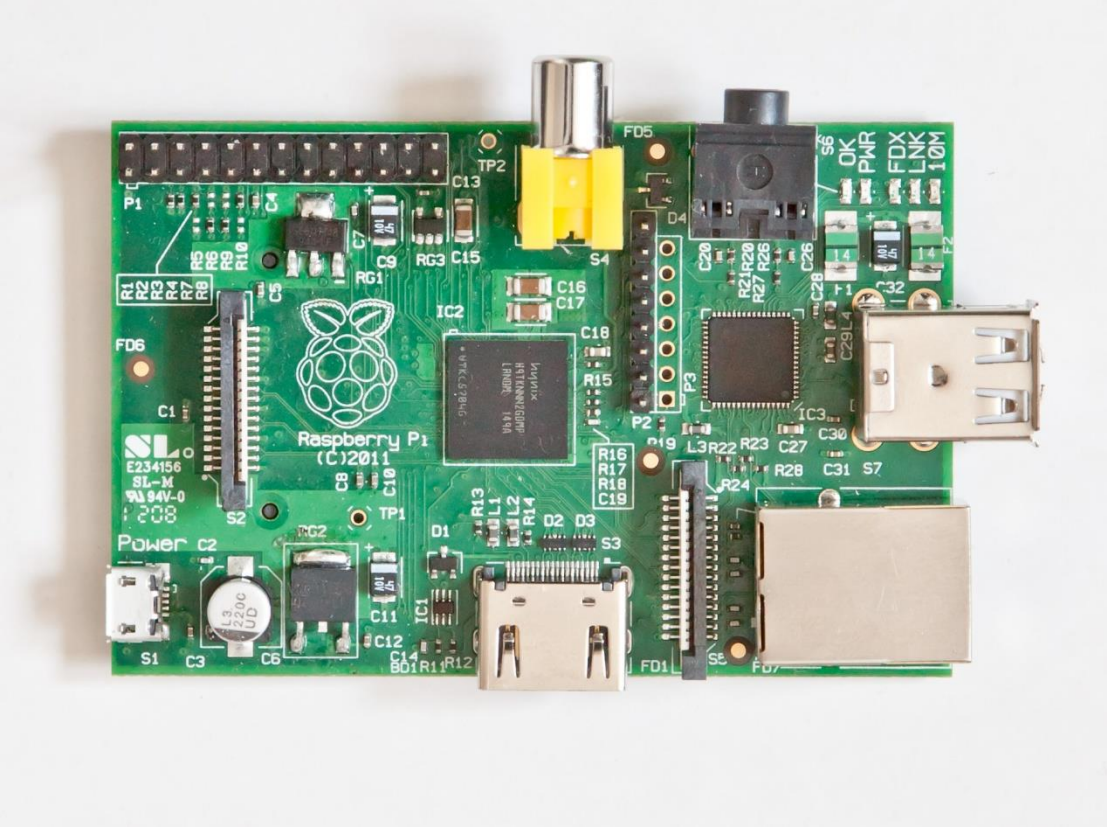


Figure.1 The Raspberry Pi [1]

The Raspberry Pi was developed by Broadcom in 2012. It was designed as an inexpensive tiny computer to help give children access to computers. The concept of tiny PC has existed for years, early product like the microcontroller manufactured by Atmel Corporation [3]. However, the Raspberry Pi is the popular fully functional tiny computing device that can provide reasonable performance. What is more, a number of accessories like mini cameras and wireless network cards can be used with it, and quite a few applications can be found on the BBS of Raspberry Pi.

There are two types of Raspberry Pi, model-A and model-B. The big difference between these two types is that model-A does not provide Ethernet controller and only has one USB port. However, it can still connect to the Internet by using an external

USB adapter. The comparison of the specific configurations and the layout of both types is shown in Figure.2 and Figure.3.

	Model A	Model B
Price	US$ 25	US$ 35
CPU:	700 MHz ARM1176JZF-S core (ARM11 family)	
GPU:	Broadcom VideoCore IV, OpenGL ES 2.0, MPEG-2 and VC-1, 1080p 30 h.264/MPEG-4 AVC	
Memory :	256 MB	512 MB
USB ports:	1	2
Video outputs:	Composite RCA, HDMI, raw LCD Panels via DSI 14 HDMI	
Audio outputs:	3.5 mm jack	
Onboard storage:	SD / MMC / SDIO card slot	
Onboard network:	None	10/100 Ethernet
Power ratings:	300 mA (1.5 W)	700 mA (3.5 W)
Size and Weight	85.60 mm × 53.98 mm, 45 g	
Operating systems:	Debian GNU/Linux, Raspbian OS, Fedora, Arch Linux ARM, RISC OS, FreeBSD, Plan 9	

Figure.2 Configuration of the Raspberry Pi

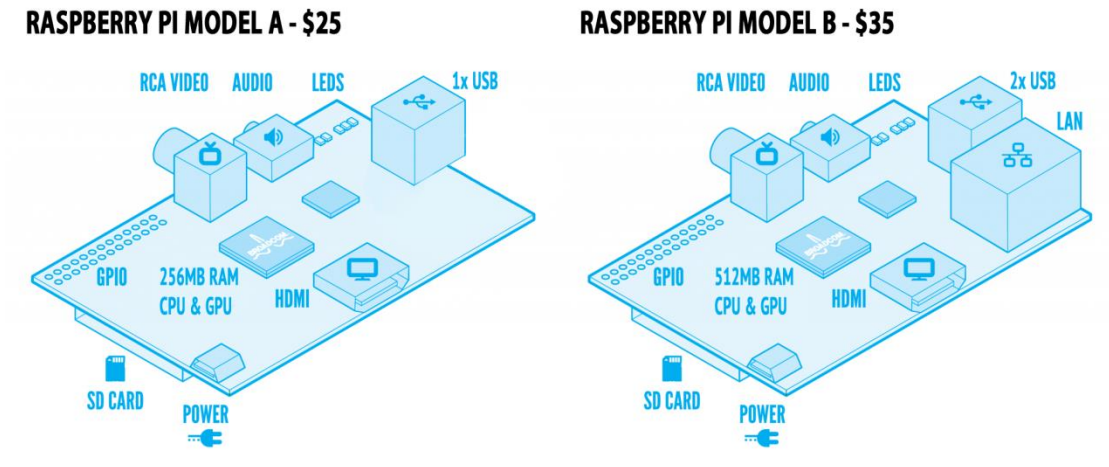


Figure.3 Layout of the Raspberry Pi [2]

In this project, B-model Raspberry Pi was used to do the development. As shown in Figure.2, B-model Pi contains almost all the components that a real desktop PC has, including: a CPU and GPU block (its GPU is as powerful as the one used on the first version of Xbox), SDRAM memory, two USB ports that can be used to connect with

input devices like mouse and keyboards, an HDMI port and a RCA port that provide video outputs (HDMI also supports audio outputs), a 3.5mm jack that provides audio outputs, an Ethernet port, and it even provides GPIO pins, so it is possible to use it with robotics. It works with Linux based operating systems, and uses SD cards (at least 2G memory space) as the onboard storage (equivalent of disk on a larger system).

XNA is a 3D graphics and game development library built around DirectX 9 and meant to be used on Windows. When programming with XNA on Windows, programmers usually use an IDE (Integrated Development Environment) like Visual Studio to get their code connected with the XNA library. However, neither XNA nor Visual Studio can be directly used on Raspberry Pi, as its operating system is one of the versions of Linux. There is a fully functional IDE named "MonoDevelop" that can be used on Linux, but XWindow implementation is too slow on Raspberry Pi because of Pi's limited memory, programmers cannot actually run MonoDevelop on Pi. Therefore, in order to allow people to do XNA programming conveniently on Raspberry Pi, two parts of work are involved.

Firstly, when programming XNA on Raspberry Pi, the code should be similar to that on Windows. In XNA programs, a lot of low level functions are provided by DirectX (i.e. when XNA needs some underlying functions to get access to the graphics cards, it makes a procedure call to DirectX). As DirectX cannot be used on Linux, some other graphics libraries that can provide graphical functionality are needed. In this project, OpenGL ES is used to replace DirectX. Therefore, the work of rewriting some of the XNA classes to make them work with OpenGL ES becomes necessary. Some similar previous work has been done at the University of Waikato that extending XNA on Windows to use DirectX 11, named "JBBRXG11". As XNA is not an open source library, the code of JBBRXG11 was used as a reference in this project. The detail of rewriting the libraries will be explained in Chapter Four of this thesis.

It was not a goal of this project to port Windows XNA programs to the Raspberry Pi. The JBBRXG11 project set out to extend XNA to use more sophisticated features on modern graphics cards. Similarly the Pi-XNA project sets out to allow 3D graphics programming in a Raspberry Pi environment, using native coding where appropriate.

For example, Pi-XNA programs should use OpenGL shader language rather than HLSL (as is used with DirectX on Windows).

Secondly, an IDE is needed to make the way of programming on Raspberry Pi more convenient. For example, programmers may add reference classes by the IDE, rather than coding all the file names in a terminal command to compile. This part of work is being undertaken by another student at the University of Waikato, and it will not be discussed in this thesis.

The remainder of the thesis is arranged into four chapters.

Chapter Two introduces some similar previous work with this project.

Chapter Three looks into the underlying software that is involved in this project. This chapter not only introduces the software themselves, but also explains how it is used to support the development of this project.

Chapter Four gives all the significant details of the implementation of this project. It firstly outlines the steps that have been taken in the project, and then discuss how each step was done, what problems occurred, and how they have been solved.

There were a number of issues which made development difficult. These include the number of libraries involved, the poor documentation of some libraries and the primitive nature of the development environment on the Raspberry Pi (test editor and command line compiler). These are compounded by the common experience of silent failure, with a blank display when errors were made in graphics programming. To accommodate these difficulties, a very careful systematic development process was followed. Working first on Windows to establish API understanding, working from sample code that could be shown to work, and developing in small steps with programs producing diagnostic output. The steps followed are documented in Chapter Four.

Chapter Five briefly conclude the work that have been done. It discusses the outcomes of this project, and compares it with the original goal. This chapter also discusses the

future work of this project. It looks into some possible research that may improve the usability, but have not been done in this project.

Chapter 2: Literature Review

This chapter briefly investigates the background of this project. It firstly looks into XNA and previous work about porting XNA on other systems, named MonoGame, and then introduces a project named JBBRXG11 developed at the University of Waikato. The source code of JBBRXG11 was used as a reference in this project.

2.1 MonoGame

"Microsoft XNA is a set of tools with a managed runtime environment that facilitates video game development and management. It is based on the .NET Framework, which is a software framework developed by Microsoft that runs primarily on Windows, and getting access to the graphics system through DirectX 9, which is a low-level API that handles tasks related to multimedia on Microsoft platforms."[4] Microsoft released its first version XNA in the year 2006. Since then, a huge number of applications and games have been developed with XNA libraries all over the world. The applications built with XNA are available to all Windows system based devices (e.g. Desktops running Windows operating system, Windows smartphones and Xbox). However, the biggest limitation is that XNA programs could not be ported to other platforms like Linux machines and android phones. In order to solve this problem, a group of software developers created MonoGame.

MonoGame is an open-source port of XNA that makes Windows operating system based games executable on other systems, like Linux and iOS. The goal of MonoGame is to allow people to program with XNA in the same way as programming on Windows (i.e. the code programmed on different platforms should be all the same). It implements the XNA 4.0 API, but replacing the Microsoft's .Net Framework with Mono.Net, which is an open source, cross-platform implementation of Microsoft's .NET Framework [5], so that programmers may develop XNA games on systems other than Windows.

The first version of MonoGame was created in 2009. The early versions only support 2D sprite based games [6]. Current versions are trying to extend MonoGame with new features like 3D rendering and multi-GameWindows. It is easy to learn and develop with MonoGame, but it is not perfect yet. Sometimes bugs occur and the programs

could crash, but with the help of MonoGame community and developers all over the world, MonoGame is getting increasingly improved. The development of MonoGame still has a long way to go.

The reason of not using MonoGame in this project is mainly because it currently does not support development on Raspberry Pi. As MonoGame is targeted to a number of different platforms, and the performance of Raspberry Pi is limited, even if MonoGame was available on Pi, it would be a little bit complicated for Pi to program with XNA. What is more, MonoGame strives for exact source code compatibility with XNA. This is not desirable for shader coding. Therefore, MonoGame is not used in this project. However, Mono is supported on Raspberry Pi. In order to port XNA on Pi, some of the XNA libraries were modified to interact with graphics cards through OpenGL ES (rather than DirectX it used), so that developers may program and smoothly execute their XNA programs on Raspberry Pi.

2.2 JBBRXG11

Most of the introduction of JBBRXG11 is referenced from jbbrxg11.codeplex.com. JBBRXG11 is a project developed by a group from the University of Waikato, trying to extend the XNA library to allow use of DirectX 10 and 11 features. DirectX 10 provides geometry shaders, and DirectX 11 allows hull and domain shaders for tessellation and also compute shaders. The motivation of this project was to fully explore the technology when programming shaders in XNA. The first version of JBBRXG11 was created by Bill Rogers in 2011, based on the XNA 3.1 and DirectX 10. In the same year, Microsoft released XNA 4.0, which modified a number of original underlying functions. The next year, this project was ported to XNA 4.0 and DirectX11 by James Boud.

XNA is not an open source library. Therefore, a lot of work of writing the code of XNA classes are involved in JBBRXG11, but this project is not simply duplicating the library. It modified some of the classes to get access to the graphics card with DirectX 11 via SlimDX. SlimDX is an open source framework, and developers may build DirectX applications with it. There was an early project lead by Bill Rogers named "SlimDXna 3", trying to extend XNA 3.1 to work with DirectX 10 (rather than

working with DirectX 9), and the JBBRXG11 is based on this project. So converting from the XNA 3.1 API to the XNA 4.0 API is necessary. This is not currently completed, which means when programming XNA with JBBRXG11, some of the program still need to be coded in the same way with that of XNA 3.1.

Although JBBRXG11 is an incomplete project, its structure and code clearly shows how XNA works with low-level APIs (i.e. DirectX). So it was used as an reference in this project. The detail of JBBRXG11 will be introduced in Chapter 3.

Chapter 3: Underlying System

This chapter introduces the software and libraries that are used to support the development of this project. It firstly looks into the detail of Microsoft's XNA, which is the library that needs to be ported on to the Raspberry Pi, then follows by the introduction of OpenGL ES, the API used to replace DirectX on Pi, and finally discusses the detail of the operating system used on Raspberry Pi, Raspbian.

This chapter not only explains the original purpose of each underlying system, but also looks into the purpose of these systems within this project and how they were used.

3.1 XNA

The goal of this project is to allow people to learn XNA programming on the Raspberry Pi. XNA is a set of object libraries accessing the DirectX graphics library with a managed runtime environment. It was developed by Microsoft, and it is built on Microsoft's .Net Framework. XNA is designed to facilitate the creation and management of video games. As described by Microsoft in 2004, "XNA is made with the intention of making the game development process easier by providing much of the underlying code and functions that are often used in games, freeing the developer to focus on programming the systems that are specific to their own games ". Figure.4 explains the architecture of an XNA program (this figure is taken from James Boud's report "Extending SlimDXna to Use XNA 4 and DirectX 11"[19]). It provides underlying functions to the users, and interacts with the graphics system through DiretX.

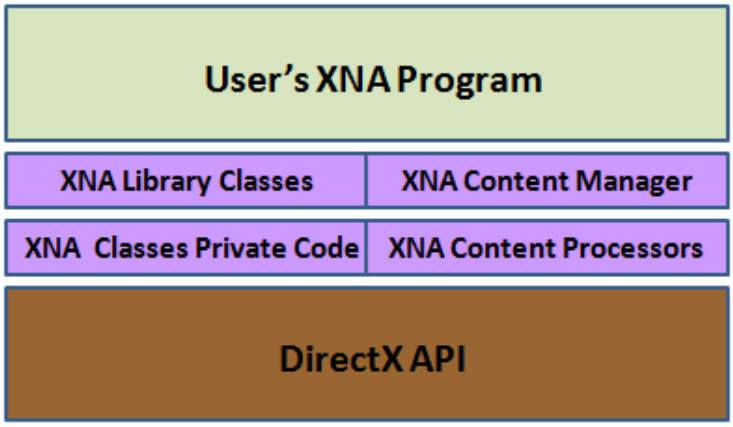


Figure.4 XNA Architecture [19]

The first version of the XNA toolset was announced in 2004, and till today, a number of versions have been developed. The latest version is XNA 4.0 Refresh, which contains a lot of new features and new methods. In an XNA project, XNA handles the running of the main loop of the program, the 3D graphics functions, and provides a number of methods that should be overridden by programmers, like the methods used for initialization and loading content. It abstracts much of the detail of using the underlying DirectX libraries, and also provides a content management system for game assets (e.g. models, textures etc.).

XNA is not an open source toolset, Microsoft only provided the name of the methods and properties of its classes. However, the source code of XNA is essential for rewriting the libraries. To solve this problem, the code of a project named "JBBRXG11" is used as reference. JBBRXG11 is a project developed by a group at Waikato University. This project rewrote the XNA classes that access the graphics system with DirectX 11 code via the SlimDX library.[7] The detail of JBBRXG 11 will be introduced in the next section.

The full XNA library is very comprehensive and complicated, it includes a large number of classes that help to develop video games. The workload of rewriting the whole system (like JBBRXG 11) would be too much. Therefore, in this project, only the classes that are involved in creating the sample programs (e.g. displaying triangle or cube) have been rewritten. The work of writing the remaining classes is leaved for future work. The project does, however, give a complete demonstration of feasibility and shows how the XNA programming model can be adapted to work with OpenGL ES on the Raspberry Pi (or any other Linux system).

3.1.1 XNA Core Classes

The 'Game' class is the core class of XNA programs. When building a new XNA project, a class named 'Game1', that inherit from 'Game', will be created for the programmer. Game provide five virtual methods that can be overridden by programmers. These are 'Initialize', 'Update', 'Draw', 'LoadContent', and 'UnloadContent'. The Update, Draw and LoadContent are of most importance to XNA programs, while the other two are not so critical and can be ignored in simple

programs. Programmers should insert their own code into these methods in 'Game1', and these methods will be called by the underlying framework when the program is executed. The details of these methods will be introduced in this section, and this section will show the source code for an XNA game. Figure.5 shows the outline source of an XNA program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace WindowsGame1
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            base.Draw(gameTime);
        }
    }
}
```

Figure.5 Outline Source of an XNA Program

The constructor of Game creates a new GameWindow object and a ContentManager object. The GameWindow object handles the creation and display of the graphics window, initializing the graphics system and allocating buffer chains etc. Programmers may change the size of the window in Game1 by modifying the value of the window properties (details will be explained later). The object of ContentManager handles the process of loading and unloading graphics resources.

After the Game object is created, the first method that will be called is 'Initialize'. As described in MSDN (the Microsoft Developer Network), "the Initialize method is called after the Game and GraphicsDevice are created, but before LoadContent and Draw. Programmers may override this method to query for any required services, and load any non-graphics resources."[8] The graphics resources can be loaded by LoadContent method. In the Game implementation, Initialize calls LoadContent, so all the resources can be loaded in the process of initialization.

"Update method is called when the game has determined that game logic needs to be processed."[9] It handles the updating of simulation data, user input, or the management of the game state. Update is called in the main loop of the game program, so nearly all the code that needs to be executed on every iteration will be placed in this method. There is a property named 'IsFixedTimeStep' that gets or sets a value indicating whether to use fixed time steps.[10] The default value of this property is true, and this made the game a fixed-step game. When a fixed-step game is running, it calls the Update method on the fixed interval specified in 'TargetElapsedTime', which is another property that gets or sets the target time between calls to Update when IsFixedTimeStep is true. [11] The default value of TargetElapsedTime is 1/60 second, but programmers may change this value.

Another critical virtual method provided by Game is 'Draw'. This method is synergistic with Update, and it will be called to draw a frame when the program determines to do so. The rate of calling Update and Draw depends on the value of IsFixedTimeStep. If its value is the default value (i.e. true), as described above, Update will be called at the interval specified in TargetElapsedTime, while Draw will continue to be called as often as possible.[12] If TargetElapsedTime has not elapsed yet after a calling from Update, the Draw method will be called. After that, if it is still

not time to call Update again, the game idles. On the other hand, if it takes too long to process the Update method, then Update will be called again without calling the Draw method, and if Update is executed longer than TargetElapsedTime, the next few frames will be dropped to catch up the runtime.[10] By contrast, if the value of IsFixedTimeStep is set to false, then both Update and Draw will be called sequentially as often as possible.

The other two virtual methods provided by Game: LoadContent and UnloadContent, are not as important as the three methods described above. LoadContent is used to load graphics resources that are needed for the game. For example, textures, videos, or shaders. This method is firstly called by Initialize when the program is running, and after that, it also can be called whenever the game content needs to be reloaded. For some simple programs, like displaying a triangle or cube, LoadContent will not be needed as there are no extra graphics resources needed. Compared to LoadContent, UnloadContent method works the other way, it is used to unload the graphics resources.

What is more, there are two other critical underlying methods in Game, 'Tick' and 'Run'. Tick controls the game time system. It updates the game's clock and calls Update and Draw at different rates according to the value of IsFixedTimeStep as introduced above. The Run method is the kernel of the XNA program. Generally speaking, all the methods described above and those that will be introduced later, are eventually called by the Run method. It firstly creates a new object of GraphicsDeviceManager and a new GraphicsDevice (these will be explained in XNA Graphics), and then calls the Initialize method (as described above, LoadContent will also be called by Initialize). After that, it builds a game window, and begin running the game loop. In the loop of the game, the Tick method will be called (so Update and Draw will be called by Tick in every iteration of the loop).

3.1.2 XNA Graphics Classes

Most of the classes of the XNA graphics system are contained by the 'Graphics' namespace of the XNA framework, which has a number of low-level API (Application Programming Interface) methods that accelerate hardware to display 3D graphics.[13] Two core classes of the XNA graphics system are

'GraphicsDeviceManager' and 'GraphicsDevice'. In this project, these two classes have been rewritten to use OpenGL ES library to operate the graphics system.

The GraphicsDeviceManager class handles the configuration and management of the graphics device.[14] In this class, the default properties of the game window are set (e.g. the size and color of the back buffer), and programmers may change the value of those properties when creating an object of GraphicsDeviceManager in Game1. When the game starts, GraphicsDeviceManager calls a function named 'CreateDevice' in GraphicsDevice to create a new graphics device object.

GraphicsDevice is a class in the Graphics namespace. As described in MSDN, "GraphicsDevice class performs primitive-based rendering, creates resources, handles system-level variables, adjusts gamma ramp levels, and creates shaders."[15] GraphicsDevice contains a variety of functions that handle the drawing of primitives.

In both XNA and OpenGL ES, 3D objects are eventually assembled from a number of primitives. Primitive is a geometric object, it could be a triangle ,a line or a point. An object like a cube or a sphere in XNA is actually drawn as a number of flat triangles, sorted in orders to make the shape look like a cube or a sphere. Using triangles to draw a sphere makes the surface appear faceted. After smoothing, using an appropriate lighting model, the effect looks much better.

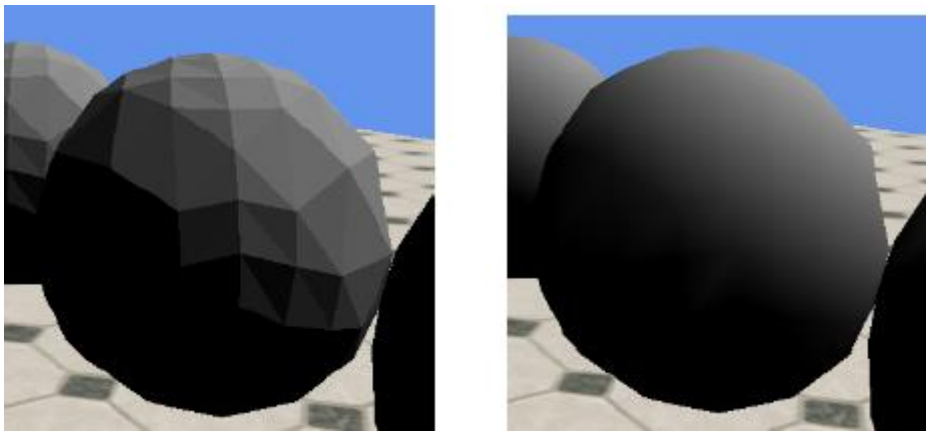


Figure.6 Smoothing

A triangle is described by its three vertices. The position of a vertex is represented by its coordinate (three float values representing the position on X axis, Y axis, and Z

axis respectively). Most of the time, the coordinate of a vertex is stored in a Vector3(will be explained later) object, shown as Figure.7. The data structures used for storing vertex data will be described in Chapter 4. Programmers should indicate the way of assigning those vertices, either sequentially use three vertices to draw a triangle, or give it another list of indices to explicitly indicate the drawing sequence. The code using a list of indices is shown as Figure.8.

```
VertexDeclaration vdecl;
vdecl = VertexPositionColor.VertexDeclaration;

VertexPositionColor[] vertices;
vertices = new VertexPositionColor[8];
//                    4--------5
//    Y              /|       /|
//    |             / |      / |
//    |            0--------1  |
//    +----X       |  6-----|--7
//   /             | /      | /
//  Z              |/       |/
//                 2--------3
vertices[0].Position = new Vector3(0, 1, 1);
vertices[1].Position = new Vector3(1, 1, 1);
vertices[2].Position = new Vector3(0, 0, 1);
vertices[3].Position = new Vector3(1, 0, 1);
vertices[4].Position = new Vector3(0, 1, 0);
vertices[5].Position = new Vector3(1, 1, 0);
vertices[6].Position = new Vector3(0, 0, 0);
vertices[7].Position = new Vector3(1, 0, 0);
```

Figure.7 Declaring the coordinates of vertices

```
int[] indices;

indices = new int[] {0, 1, 2, 1, 3, 2,  // front
                     1, 5, 3, 5, 7, 3,  // right
                     5, 4, 7, 4, 6, 7,  // back
                     4, 0, 6, 0, 2, 6,  // left
                     4, 5, 0, 5, 1, 0,  // top
                     2, 3, 6, 3, 7, 6}; // bottom
```

Figure.8 Using a list of indices to indicate the sequence of drawing

3.1.3 XNA Mathematical Classes

The flow of an XNA program can be expressed as a pipeline diagram. As shown in Figure.9, there is one step before primitive assembly in the XNA graphics pipeline, which is transform and lighting. Transform handles the positioning and transforming of objects in the virtual environment from a 3D space to a 2D screen (i.e. to project the scene of 3D objects captured by the camera on to the specified area of the screen). XNA provides a series of mathematical classes that helps the transformation between different coordinate systems, including 'Vector2', 'Vector3', 'Vector4', 'Rectangle', and 'Matrix'. For this project, these classes were all rewritten to make them executable with OpenGL ES on Linux. In XNA programs, a set of matrices can be created to do the transformation, which are the World matrix, View matrix, and Projection matrix. The World matrix mainly handles the rotation, scaling, and translation of the objects in the 3D virtual world that is defined by the programmer. Compare to World, the View matrix sets the camera position, it also handles the rotation and translation of the coordinate system. The Projection matrix specifies the attribute of the camera, like perspective, field of view, and some depth limitations. The Matrix class provides a function named 'createPerspectiveFieldOfView' to set those values. After all three matrices are set, the final step is to multiply them in order (World * View * Projection), thus the final matrix that used for calculating the position can be acquired.

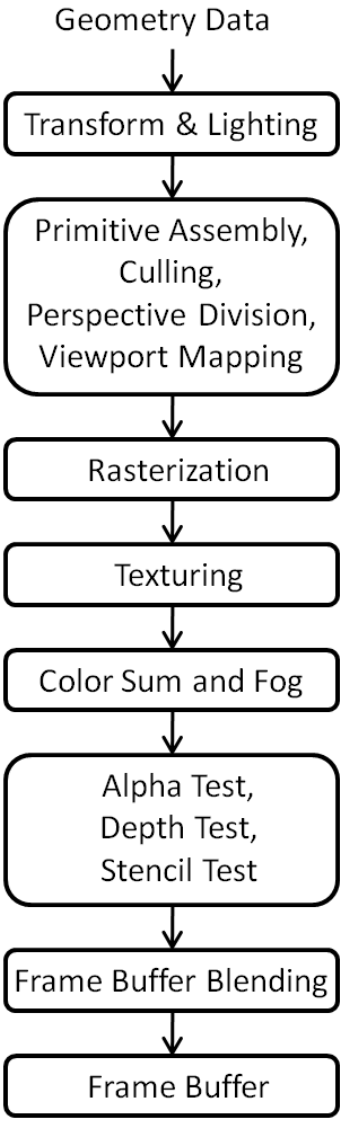


Figure.9 XNA Fixed Function Graphics Pipeline

Lighting is built from four components, ambient lighting, diffuse lighting, specular lighting, and emissive lighting. With lighting effect, programmers are able to calculate reflections and shadows, which make the game graphics seems more real. When building a model in XNA, programmers may give every vertex a normal. Normal is a vector that pointing out from a surface, and it is used to calculate interaction with light sources. The sample code of drawing a cube with normals is shown as Figure.10. Coordinates of a set of vertices are given on the left part of the code, and each vertex is given a normal vector as shown on the right part of the code.

```
vertices = new VertexPositionNormalTexture[24];

vertices[00].Position = new Vector3(0, 1, 1); vertices[00].Normal = new Vector3(0, 0, +1);
vertices[01].Position = new Vector3(1, 1, 1); vertices[01].Normal = new Vector3(0, 0, +1);
vertices[02].Position = new Vector3(0, 0, 1); vertices[02].Normal = new Vector3(0, 0, +1);
vertices[03].Position = new Vector3(1, 0, 1); vertices[03].Normal = new Vector3(0, 0, +1);
vertices[04].Position = new Vector3(1, 1, 1); vertices[04].Normal = new Vector3(+1, 0, 0);
vertices[05].Position = new Vector3(1, 1, 0); vertices[05].Normal = new Vector3(+1, 0, 0);
vertices[06].Position = new Vector3(1, 0, 1); vertices[06].Normal = new Vector3(+1, 0, 0);
vertices[07].Position = new Vector3(1, 0, 0); vertices[07].Normal = new Vector3(+1, 0, 0);
vertices[08].Position = new Vector3(1, 1, 0); vertices[08].Normal = new Vector3(0, 0, -1);
vertices[09].Position = new Vector3(0, 1, 0); vertices[09].Normal = new Vector3(0, 0, -1);
vertices[10].Position = new Vector3(1, 0, 0); vertices[10].Normal = new Vector3(0, 0, -1);
vertices[11].Position = new Vector3(0, 0, 0); vertices[11].Normal = new Vector3(0, 0, -1);
vertices[12].Position = new Vector3(0, 1, 0); vertices[12].Normal = new Vector3(-1, 0, 0);
vertices[13].Position = new Vector3(0, 1, 1); vertices[13].Normal = new Vector3(-1, 0, 0);
vertices[14].Position = new Vector3(0, 0, 0); vertices[14].Normal = new Vector3(-1, 0, 0);
vertices[15].Position = new Vector3(0, 0, 1); vertices[15].Normal = new Vector3(-1, 0, 0);
vertices[16].Position = new Vector3(0, 1, 0); vertices[16].Normal = new Vector3(0, +1, 0);
vertices[17].Position = new Vector3(1, 1, 0); vertices[17].Normal = new Vector3(0, +1, 0);
vertices[18].Position = new Vector3(0, 1, 1); vertices[18].Normal = new Vector3(0, +1, 0);
vertices[19].Position = new Vector3(1, 1, 1); vertices[19].Normal = new Vector3(0, +1, 0);
vertices[20].Position = new Vector3(0, 0, 1); vertices[20].Normal = new Vector3(0, -1, 0);
vertices[21].Position = new Vector3(1, 0, 1); vertices[21].Normal = new Vector3(0, -1, 0);
vertices[22].Position = new Vector3(0, 0, 0); vertices[22].Normal = new Vector3(0, -1, 0);
vertices[23].Position = new Vector3(1, 0, 0); vertices[23].Normal = new Vector3(0, -1, 0);
```

Figure.10 Sample code of normal

By using lighting effect, the color of each surface of a white cube will be different. The effect is shown as Figure.11. Surfaces that face towards the light direction are brighter than those face away from the light.

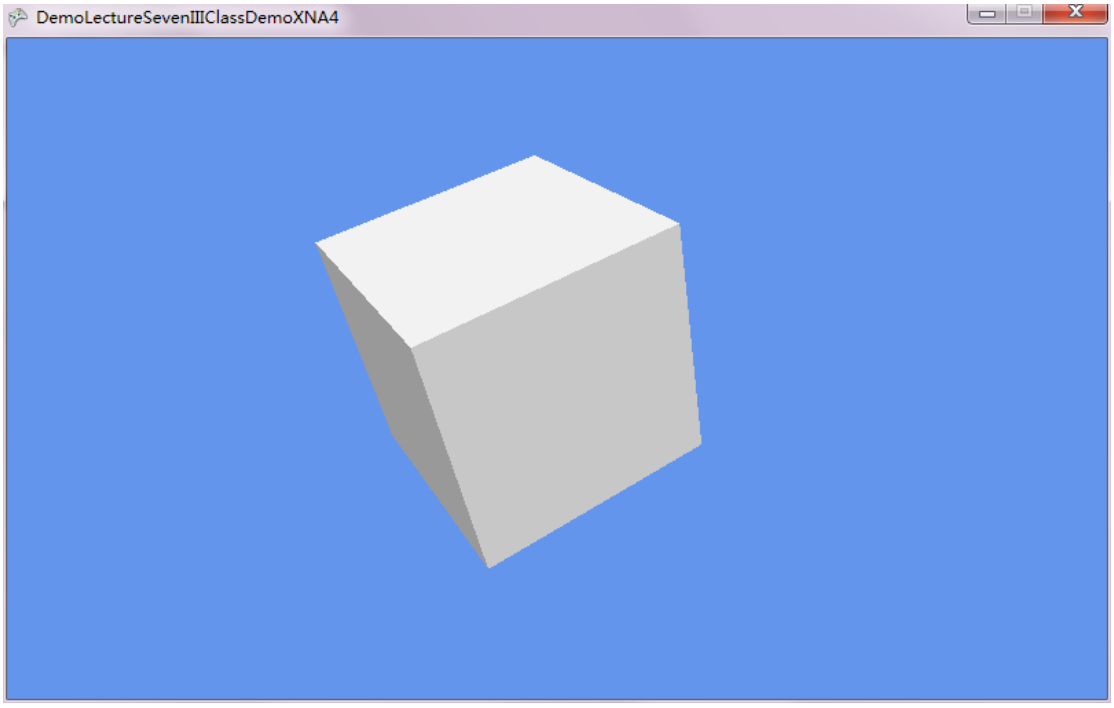


Figure.11 Lighting effect

3.1.4 Shader

Before programmable graphics pipeline was created, programmers needed to send all the model data and textures to the graphics card, and they could only use a fixed set of functions to configure the way of drawing the objects. Shaders were then developed to improve the efficiency of programs. Shaders are small programs that will be compiled and sent to the graphics card to handle the graphical data. In XNA, they are written in HLSL (stands for High Level Shader Language), which is a kind of special programming language that fairly similar to C programming language (OpenGL uses a different language named GLSL, it is similar to HLSL, but not identical).[16]

Generally speaking, there are two kinds of shaders used in XNA, which are Vertex Shader, Pixel Shader(also known as Fragment Shader). As shown in Figure.12, in the programmable pipeline, vertex shader allows programmers to program their own codes to operate the transform and lighting, and the position of objects can be altered in this stage. Compare to vertex shader, pixel shader replaces the texturing stage. It handles advanced lighting and determines the color of each pixel. In addition, JBBRXG11 and DirectX 10 give access to geometry shaders. These operate in the rasterization stage, and can create or destroy primitives. Programmers may create a ".fx" file to put in their shader code, and load it as a content into the XNA program. XNA also provides a built-in shader named 'BasicEffect', which is created by the graphics device and supports a variety of basic graphics effects (similar to the fixed function pipeline). XNA 4.0 runs on a variety of platforms, and DirectX 10 and DirectX 11 do not provide fixed function pipeline anymore.

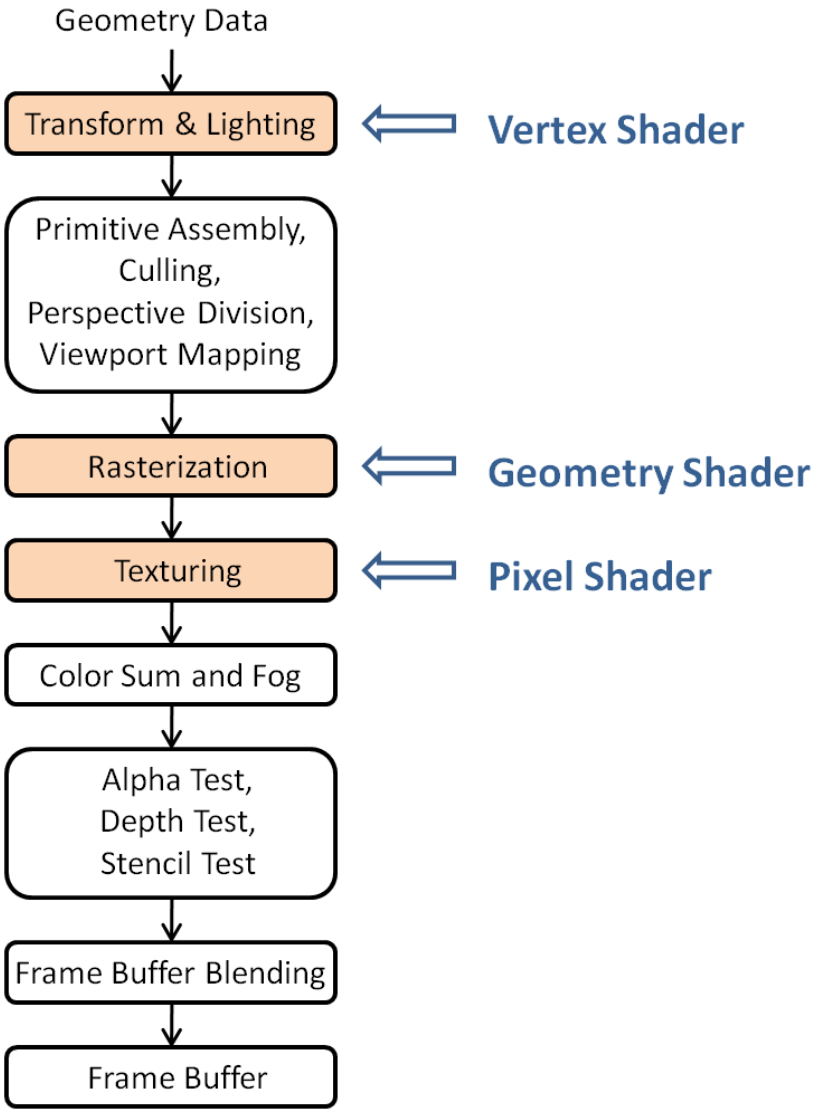


Figure.12 Shaders

3.1.5 XNA Inputs

XNA framework provides a flexible library (the Microsoft.Xna.Framework.Input namespace) that handles user inputs. Most input devices can be supported (e.g. keyboard, mouse, touch panel etc.). In this project, the Input library is only used to take keyboard and mouse input to the program.

A structure named 'KeyBoardState' is used to represent the state of keystrokes. It contains a variety of methods, and two of them would be frequently used by programmers, which are 'IsKeyDown' method and 'Is KeyUp' method. These two methods return bool values to indicate if a specified key is currently being pressed or

not respectively. The current keyboard state can be checked by 'GetState' method provided by Keyboard class.

Compare to keyboard, the mouse position and the state of mouse buttons (i.e. mouse button click) can be retrieved by calling the GetState method of Mouse class. The MouseState structure provides three properties, 'LeftButton', 'MiddleButton', and 'RightButton', that return the state (Pressed or Released) of mouse buttons. Another two properties, 'X' and 'Y', specify the position of the cursor. The position of the mouse is its related position to the upper-left corner of the game window, and the values of X and Y represent the horizontal distance and vertical distance between the cursor and the upper-left corner respectively.

3.2 OpenGL ES

In this project, EGL and OpenGL ES were used to get access to the graphics system. OpenGL ES, created by the Khronos Group, is a subset of OpenGL (Open Graphics Library) that designed for embedded devices like mobile phones, computer tablets and personal digital assistants. "It is a cross-language, multi-platform application programming interface (API) for rendering 3D graphics."

DirectX and OpenGL are two standard 3D APIs for programs to interact with a Graphics Processing Unit (GPU). DirectX can be used on any devices that running the Microsoft's Windows operating system. In an XNA program, nearly all the low-level APIs are provided by DirectX. Compare with DirectX, OpenGL is a cross-platform API that can be used on both Windows and Linux. In this project, OpenGL ES is used to replace DirectX to provide API functions to the XNA, so that programming with XNA on Raspberry Pi becomes possible.

The OpenGL ES used in this project is version 2.0, which is derived from the OpenGL 2.0 specification. Because of the constrained performance of embedded systems, OpenGL ES is not as complex as OpenGL, a number of redundant functions have been removed to ensure the efficiency of the program. Compare with the early versions of OpenGL ES (i.e. ES 1.0 and ES 1.1), the ES 2.0 implements a programmable graphics pipeline, rather than a fixed function pipeline, which makes

use of the programmable graphics capabilities available on embedded systems.[17] The OpenGL ES 2.0 graphics pipeline is shown in Figure.13. This pipeline is similar with the XNA graphics pipeline. The systems are the same because they run on similar hardware.

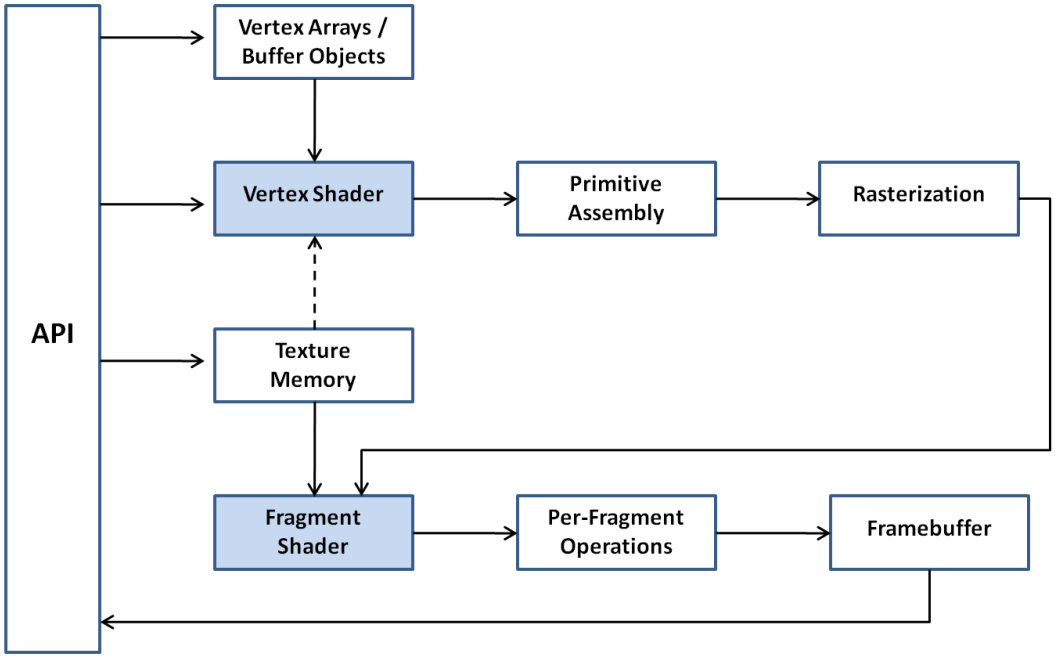


Figure.13 OpenGL ES 2.0 Graphics Pipeline [17]

The shaded stages (i.e. Vertex Shader and Fragment Shader) are the programmable stages in the pipeline. The details of shaders will be discussed later in this chapter. In the OpenGL ES 2.0 graphics pipeline, the vertex arrays and buffer objects are firstly passed into the vertex shader to implement some operations on the vertices (e.g. transforming, lighting etc.). After the vertex shader, is the primitive assembly stage. A primitive is a basic geometric object that can be drawn on the screen. There are three types of primitives, which are points, lines and triangles. In primitive assembly, the shaded vertices are assembled into individual geometric primitives, and then the 3D coordinates are converted into the screen 2D coordinates.[17] The next stage is the rasterization that each primitive is converted into a 2D fragment. These fragments will then be sent to the fragment shader where the fragments' color is generated. The last stage before framebuffers are generated is the per-fragment operations. This stage consists of a series of tests, and the fragment color generated in the fragment shader can be modified. Compare with the per-fragment operations in OpenGL 2.0, Alpha

Test and Logical Operations are eliminated in OpenGL ES 2.0. An overview of the operations is shown in Figure.14.

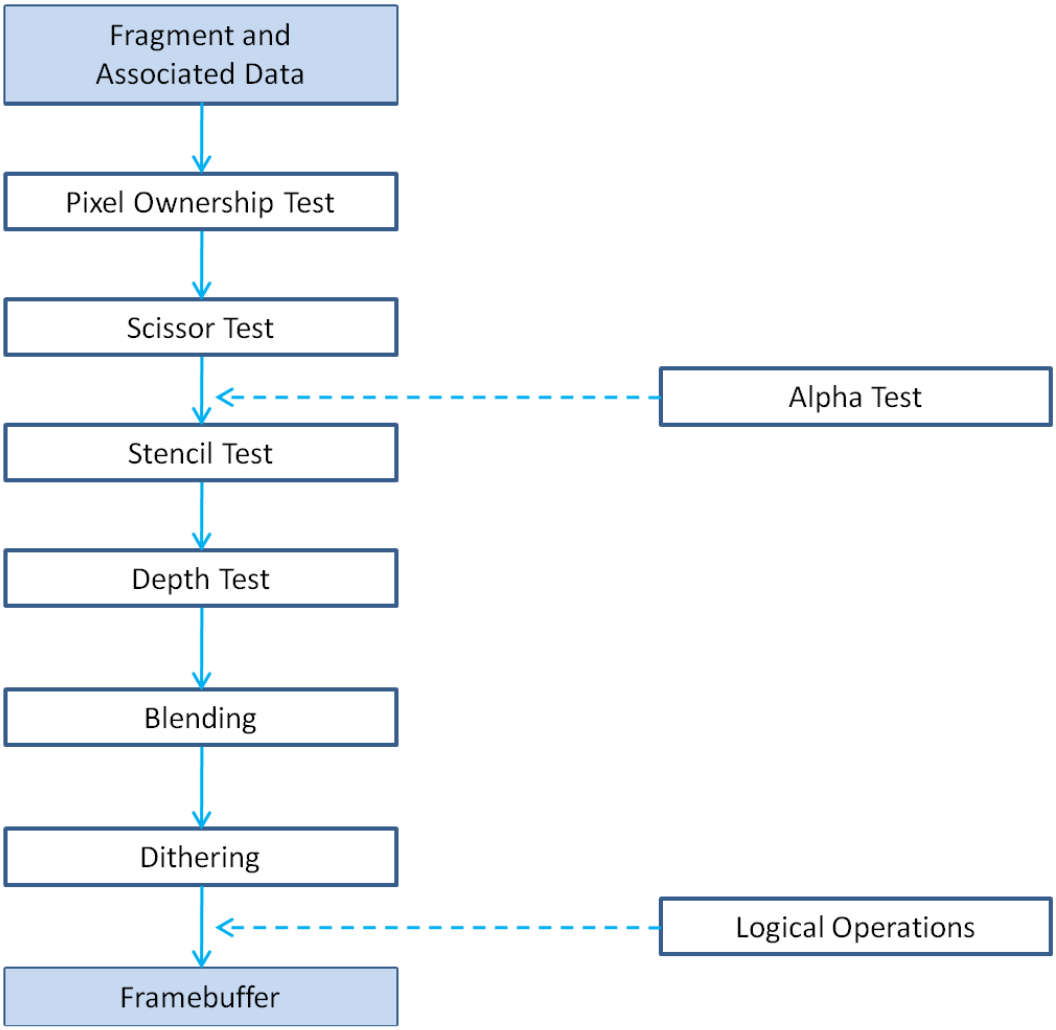


Figure.14 Per-Fragment Operations [21]

The investigation in this project started with an OpenGL ES sample, the sample code mainly contains three parts, initialize, create program and render, and they were coded as three major functions in the program.

Initialize is the first function called by the main function, and it will be called only once. This stage handles the initialization of EGL (details of EGL will be explained in the next section in this chapter), communication with the native windowing system, creating rendering surface and some other works to set up a display environment.

Once the initialization is finished, the CreateProgram function is called. This function creates shader objects and program object (the details of shaders will be discussed later in this chapter), loading the shader source code and compiling the shaders, attaching the shader objects as a program object, and linking the shader program. This function is called only once as well.

Compared with Initialize and CreateProgram, the Render function is called on every iteration of the main loop. It handles the creation of buffers, loading the vertex data into the buffers, using the program object, linking vertex data with variables in the shaders, and finally drawing the primitives on the screen.

3.2.1 EGL

EGL (Embedded-System Graphics Library) is a platform-independent API created by the Khronos Group. It is an interface between rendering APIs like OpenGL ES and the underlying native platform windowing system.[18] The mechanisms provided by EGL mainly include creating the drawing surfaces and binding the buffers, querying configurations of drawing surfaces, managing the graphics context and synchronizing rendering between OpenGL ES 2.0 and other graphics-rendering APIs (e.g. the native windowing system).[17]

This project used EGL version 1.4, which contains dozens of functions. However, only six of them were used during the initialization stage.

The first function called by the program is "eglGetDisplay". It is used to communicate with the windowing system, and it returns a token that representing the native display type. After that, "eglInitialize" is called to initialize EGL's data structures. Once the initialization is done, the next step would be choosing an available rendering surface configuration. Generally, there are two ways to do this. Programmers either query all the configurations and find one themselves, or just let EGL make the determination by specifying a set of requirements. In this project, the second way was used by calling "eglChooseConfig". After a suitable configuration of rendering has been determined, the EGL window and rendering context can be created by calling the function "eglCreateWindowSurface", which connects to the native display manager, the configuration that has been chosen, and "eglCreateContext" respectively. When the

program creates more than one context, "eglMakeCurrent" is needed to associate a context with the rendering surface. If all these six functions work properly, then the initialization process has been successfully done.

Another EGL function named "eglSwapBuffers" was used in the rendering stage in this project. The reason of using this function is that when displaying the frame buffer on the screen, there are actually two buffers alternately displaying the images. The image of a frame is represented by a two-dimensional array of pixel data. If there is only one buffer used to update the frames, because of the fixed updating rate from its memory, artifacts might be displayed when only part of the buffer data has been updated. A scheme called double buffering is used to solve this problem. A front buffer and a back buffer are used, and all the rendering process only occurs to the back buffer. Once the rendering is complete, the back buffer will be swapped to the front, and the original front buffer becomes the current back buffer to render the next frame. Thus, artifacts will no longer be displayed on the screen.

3.2.2 Shaders and GLSL

As explained earlier, OpenGL ES 2.0 implements a programmable graphics pipeline. Programmers may put their own code in the shaders to do postprocessing and produce special effects. Similar to XNA programming, OpenGL ES provides vertex shader and fragment shader (it is called pixel shader in XNA), and programmers may use OpenGL Shading Language (GLSL) to program shader code in the graphics pipeline.

The vertex shader is responsible for transforming the positions of vertices from 3D space to 2D coordinate space, and it can also modify the coordinates and colors of those vertices. In OpenGL ES, a vertex shader takes attributes, uniforms and samplers as its input. Attributes could be the positions and colors of the vertices, or texture coordinates. These values are always stored in vertex arrays in the program. In this project, an array of structures that contains vertices' coordinates, colors, and other information is used to provide those attributes. The details will be introduced in Chapter.4. Uniforms provide constant data to the vertex shader, like the world, view, and projection matrices. Samplers are an optional input to a vertex shader, and are used to represent textures. The output of vertex shader is called varyings, like vertex colors and texture coordinates. These values are interpolated in rasterization stage,

and the results will be sent to the fragment shader.

The fragment shader takes the output varyings of the vertex shader as its inputs. It mainly handles the calculation of the colors and some other attributes of the fragments, and finally outputs the color of each fragment. The inputs and outputs of the shaders are shown in Figure.15.

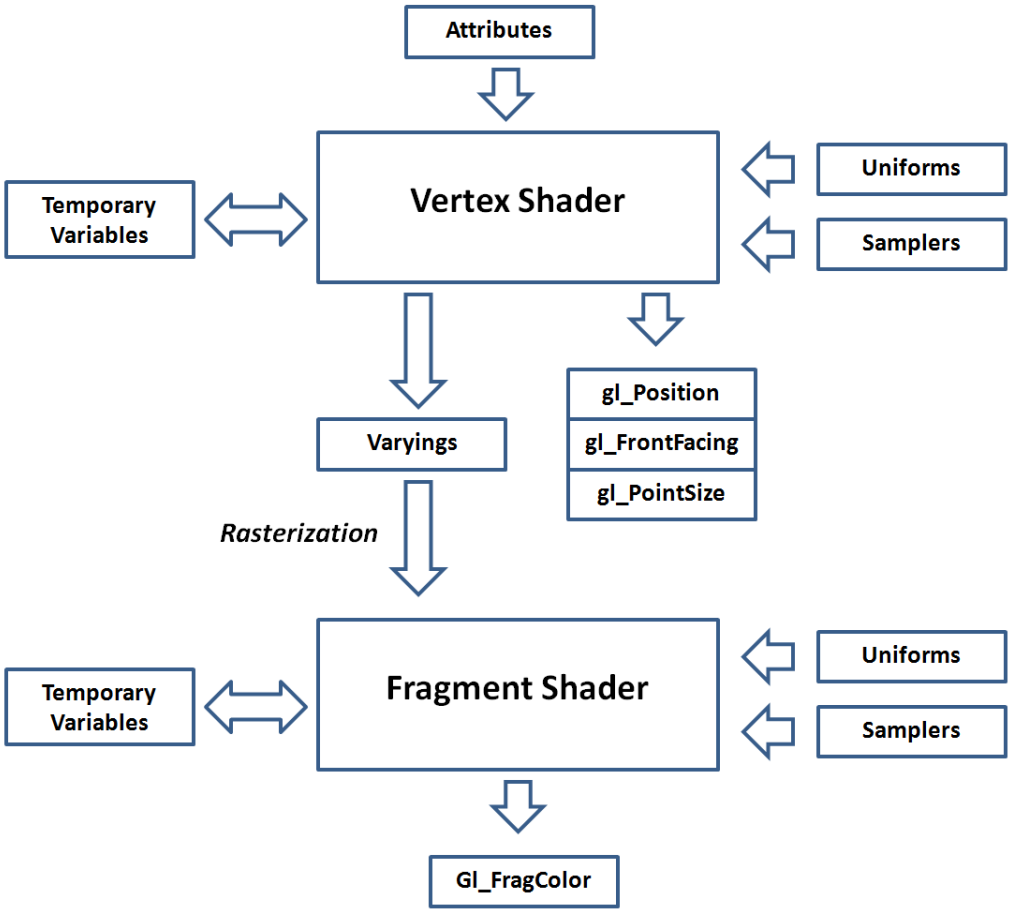


Figure.15 Inputs and Outputs of the Shaders [17]

In OpenGL ES, shader objects and program objects are essential for using shaders. In the process of using shaders, shader objects are firstly created. In the sample code of this project, two shader objects were created, one for the vertex shader, and the other for the fragment shader. Shader objects contains the shader source code (one shader object contains the source code of one shader), and these objects will be linked to a later created program object after compiling. A program object is attached with one vertex shader and one fragment shader, no more and no less. The program object is then linked to the program, and the shaders are ready for rendering.

The programming language used for coding OpenGL ES shaders is called OpenGL ES Shading Language (GLSL), which is a high-level shading langrage created by OpenGL Architecture Review Board. GLSL and HLSL are similar to each other. They are all based on the syntax of C programming language, but there are still some differences between them, like the keyword of variable types (e.g. "vec4" in GLSL, but "float4" in HLSL) and the names of some built-in functions.

However, the goal of this thesis is to investigate how XNA programming can be ported to the Raspberry Pi. Since this project focuses on rewriting the classes of XNA, and the differences between both shading languages are not so numerous, GLSL was directly used to replace the HLSL for coding shaders in this project.

3.3 JBBRXG11

As described in section 2.2, the JBBRXG11 project is built with the XNA 4.0 library, trying to extend XNA to work with DirectX 11 via SlimDX. As XNA is not an open source library, the JBBRXG11 project involves a large amount of work to rewrite the source code of XNA classes.

SlimDX is an open source framework that allows developers to create DirectX applications. Essentially it is a wrapper for DirectX. In JBBRXG11, SlimDX was used to access DirectX 11 methods when replacing the XNA classes that access the graphics systems. Figure.16 shows the generalized system architecture of JBBRXG11.

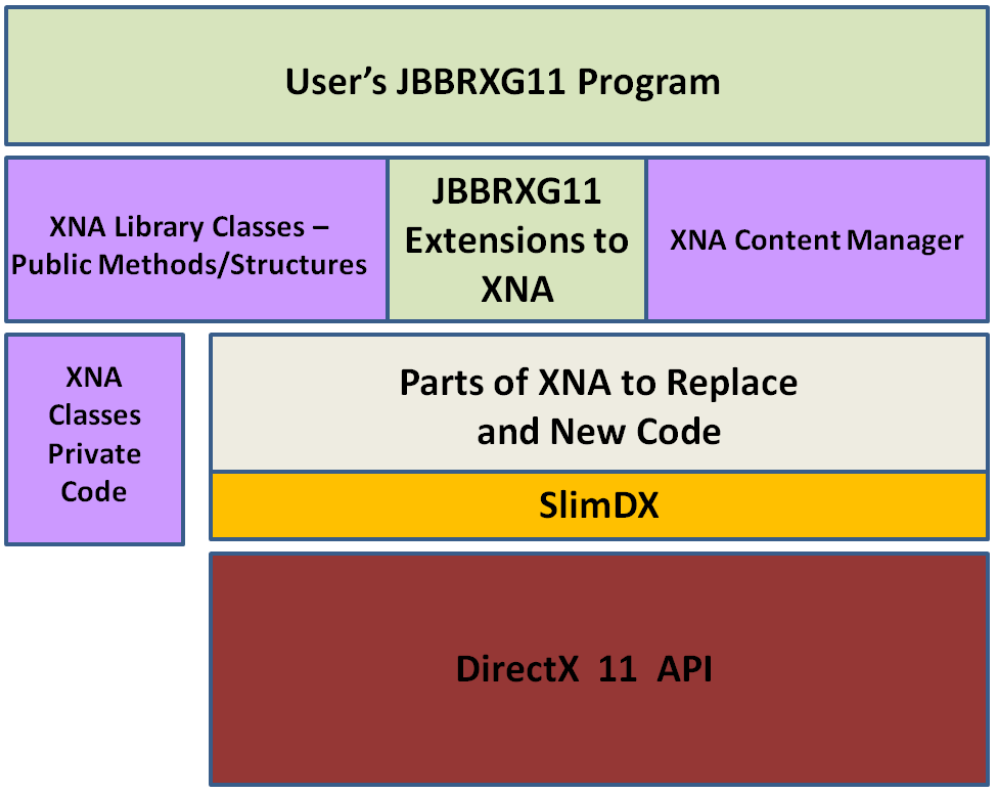


Figure.16 Architecture of JBBRXG11 [19]

JBBRXG11 replaced all classes that interact with DirectX 9 using SlimDX to get access to DirectX 11, and the XNA's content manager system was also replaced by modified content loaders. When users program with the JBBRXG11 system, their code will be similar to XNA programs, but the data is actually send to DirectX 11 through SlimDX. The JBBRXG11 system can be downloaded from the web page http://jbbrxg11.codeplex.com/ [20]

This thesis investigates how to port XNA to Raspberry Pi, trying to replace DirectX with OpenGL ES. The JBBRXG11 system is an extension of XNA. It still works with parts of the XNA framework. However, the XNA framework cannot be used in the modified classes as the operating system used on Pi is not Windows.

In this project, the source code of JBBRXG11 was used as a reference to rewrite some XNA classes. The modified classes provides methods that have the same names with those in XNA to the users, and the XNA classes that interact with DirectX were replaced with modified classes to make procedure calls from OpenGL ES. Figure.17 shows the architecture of the modified system in this project. XNA classes are still

used in JBBRXG11, but they were replaced with new versions appropriate for the Raspberry Pi in this project.

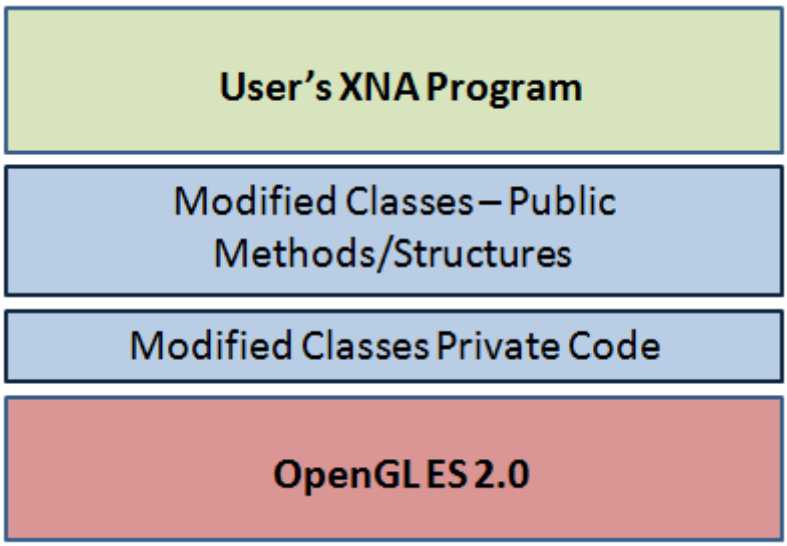


Figure.17 Architecture of Modified System on Raspberry Pi

When programming with the modified classes on Raspberry Pi, users' programs are similar to XNA programs on Windows, but the underlying classes actually interact with graphics systems through the OpenGL ES library.

3.4 Raspbian

Raspbian is the most widely used operating system for Raspberry Pi. This project used Raspbian as well. However, when Raspberry Pi was first announced, the operating system used on Pi was the ARM based Fedora created by Red Hat company. As the early version of Raspberry Pi only has 256M memory space, which does not meet the requirement of Fedora, it cannot smoothly run this system. In August 2012, Raspbian was released and replaced Fedora.

Raspbian is a Debian based operating system created by Mike Thompson and Peter Green. A large amount of work has been done by these two volunteers, including a home-built cluster of ARM computers, and the rebuilding of 19,000 Linux software packages.[7]

A number of versions of the Raspbian have been developed. The one used in this

project is the "Hard-float" version Raspbian. A Floating-point Unit makes mathematical calculations very quick. Although Debian supports floating point for ARMv7 processors, it does not support ARMv6 (the version used on the Raspberry Pi). As explained by Thompson, "Debian didn't see a product like the Raspberry Pi coming on the horizon. Even though ARMv6 in Pi has a pretty capable floating point unit, they didn't support it, all the thousands or tens of thousands of software packages they built wouldn't support the Raspberry Pi. A floating point unit performs all the math very quickly, it's a peripheral that not every computer has, but when it does you really want to take advantage of it."[7] Using floating point unit may increase the calculation speed of the system, which is important for any numerically intensive tasks.

The biggest difference between the "Hard-float" and the "Soft-float" is speed. The Hard-float uses on-chip floating point unit, while the Soft-float emulates one in software. Making software handle a large amount of mathematical operations will obviously slow down the speed of Pi. Therefore, the "Hard-float" version Raspbian was used in this project to take advantage of floating point capability in the hardware.[7] However, Mono does not work with the Hard-float version Raspbian. The way of installing Mono will be explained in the Appendix.

Chapter 4: Development

This chapter discusses all the significant details of the implementation in this project. As explained earlier, XNA is an library designed for the Windows operating system, and it can interact with the graphics card through DirectX, which is an API designed for Windows as well. This project investigates the possibility of porting XNA on to Raspberry Pi, which is a computing device that can run the Linux operating system. In order to achieve this goal, OpenGL ES was used to replace DirectX, and a number of XNA classes needed to be rewritten. However, before rewriting the code of XNA, some other work needed to be done.

The two biggest challenges of this project are, firstly, how to make the modified XNA interact with the graphics system through OpenGL ES 2.0 on Raspberry Pi, and secondly, when users programming with the modified XNA on Raspberry Pi, how to make their code similar to that programmed on Windows.

Therefore, before doing any modification to XNA, the first thing is to learn how OpenGL ES works. The operating system used on Raspberry Pi, the Raspbian, provides the OpenGL ES 2.0 library, and also some sample code that uses this library to display objects. However, these sample projects are all written in C++, while XNA classes are all written in C#. So the second thing to do before modifying XNA is to investigate how to make procedure call to OpenGL ES using C# language. What is more, it is also necessary to find out how to use the native windowing system on the Raspberry Pi.

As there are no suitable IDEs that can be used on Raspberry Pi currently, and OpenGL ES is a cross-platform API, in order to more easily find errors in the program, the two steps described above (i.e. learning OpenGL ES and getting access to OpenGL ES with C# code) were firstly done on Windows (with Visual Studio 2010), and then move the program on Raspberry Pi to do further modifications. Experiments with windowing were done on the Pi itself.

Once programming with OpenGL ES in C# language on Raspberry Pi is successful, the last step is to rewrite some of the XNA classes to allow coding similar to

programming in XNA on Windows.

The remainder of this chapter is divided into three sections, explaining the details of how each step was done in this project. Each step created four sample programs, started with displaying a blank window, followed by displaying a triangle, then a colored rotating cube, and finished with showing a textured rotating cube. In the end of this project, the modified XNA allows programmers to create a textured rotating cube with lighting effect on Raspberry Pi, shaders are in separate files, and the code is similar to that on Windows (all the programs mentioned in this thesis are included in the Appendix).

4.1 C++ code with OpenGL ES 2.0 on Windows

This project started with learning OpenGL ES and writing four sample programs (displaying blank window, triangle, cube and textured cube) in C++ with OpenGL ES. As errors may occur when developing sample programs, it is better to program with an IDE that is capable of indicating syntax errors and debugging. However, because there were no suitable IDEs that could be used on Pi, all the sample programs were firstly developed on Windows. Once a program worked, it was moved to Raspberry Pi, and further modified to be executable on Raspbian.

The next section which explains how to rewrite the four sample programs in the C# language was also started on Windows, and then moved to Raspbian.

4.1.1 OpenGL ES 2.0 Sample Program

Before writing sample programs, the way of programming with OpenGL ES should be understood. This project used OpenGL ES 2.0 to replace DirectX, so a sample program that uses GLES version 2.0 was used as a reference.

As shown in Figure.18, this sample code is written in C++, displaying a textured ninja model on a colored background, and the angle of the view can be changed by clicking and moving the mouse. It contains a number of classes and functions to implement the display, but only four of them are of importance to the program.

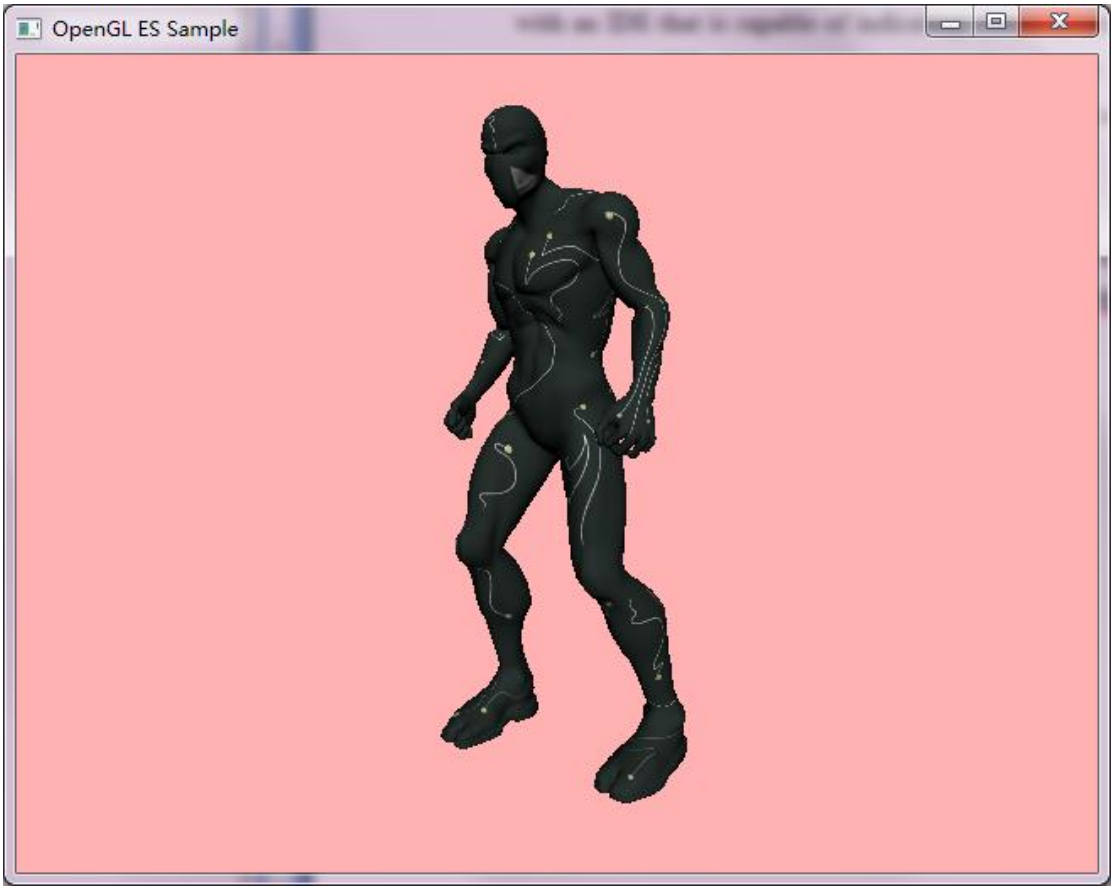


Figure.18 Ninja Sample

Figure.19 explains the basic structure of the Ninja program.

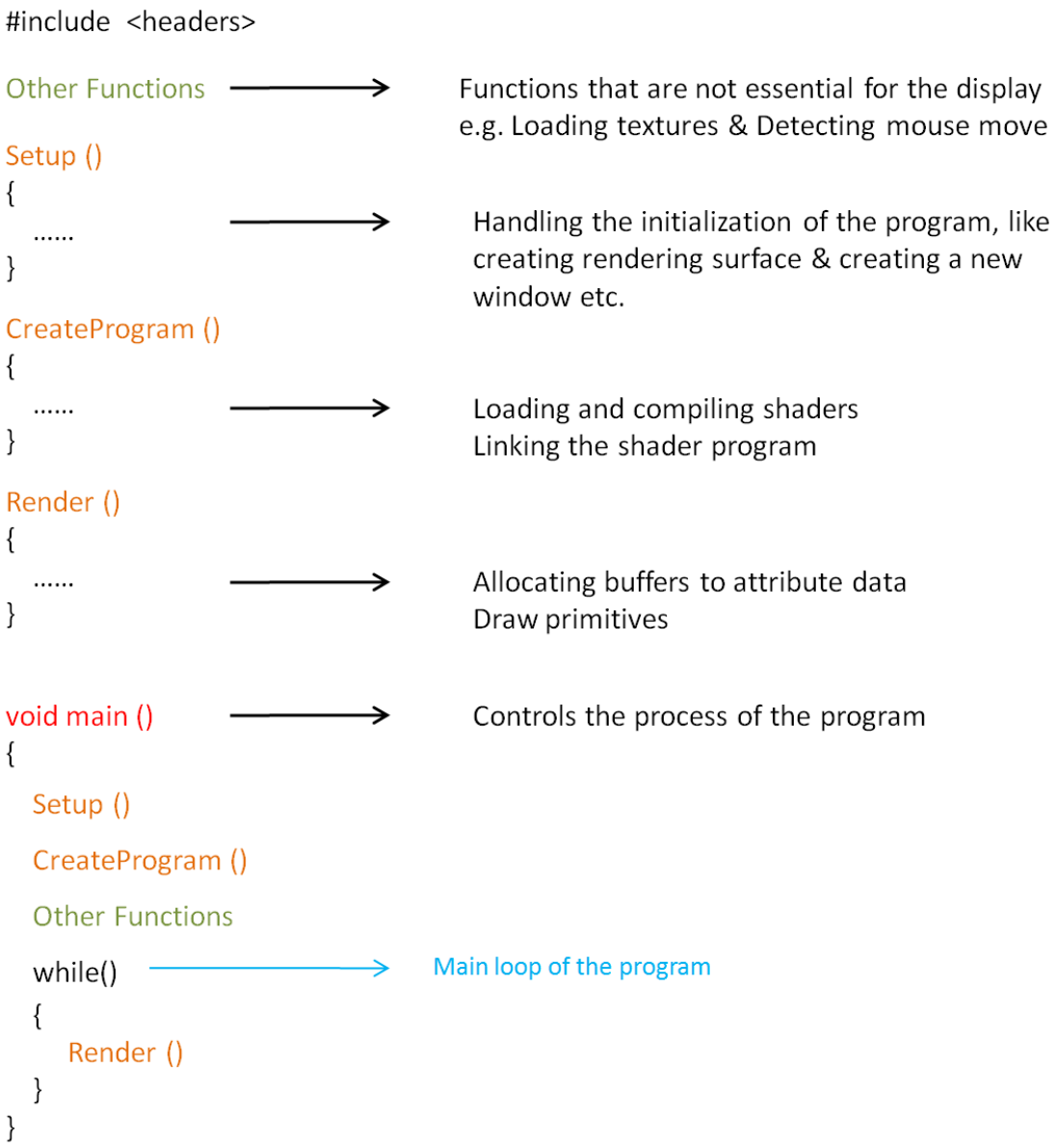


Figure.19 Structure of the Ninja Program

As a C++ program, before writing the classes and functions, the first thing to do is to include header files that are needed in the program. Figure.20 shows the header files used in the Ninja program, the "egl.h" and "gl2.h" allows programmers to use the functions in EGL and OpenGL ES 2.0 libraries (the detail of EGL and OpenGL ES 2.0 have been described in Chapter 3). "nativewin.h" is used for interacting with the native windowing system.

```
#include <EGL/egl.h>
#include <GLES2/gl2.h>

#include "nativewin.h"
#include "sbm.h"
#include "vecmath.h"

#include <iostream>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
```

Figure.20 Header Files in the Ninja Program

The most important function in this sample program is the main function. It controls the process of the program by calling other functions.

The first function called by main is the 'Setup' function, which is responsible for initializing all the component of display. In the Setup function, it firstly calls OpenNativeDisplay to interact with the native windowing system and creates a new window, and then calls eglGetDisplay to connect to the EGL display server, and gets an EGL display handle. Once the connection is successful, EGL needs to be initialized by calling eglInitialize function, which uses the EGL display handle as one of its parameters. After that, eglChooseConfig is called to obtain the first display configuration with a depth buffer. Once a suitable EGL configuration has been chosen, the next step is to create an on-screen rendering surface for the main window, and an OpenGL ES rendering context. "A rendering context is a data structure that contains all of the state required for operation, like the references to the vertex and fragment shaders and the array of vertex data."[17] These are done by calling the eglCreateWindowSurface and the eglCreateContext respectively. The last function called by Setup is eglMakeCurrent, which makes the EGL context and the rendering surface current, because multiple contexts may have been created in the program. The process of initialization of the program (i.e. the Setup function) is shown as Figure.21.

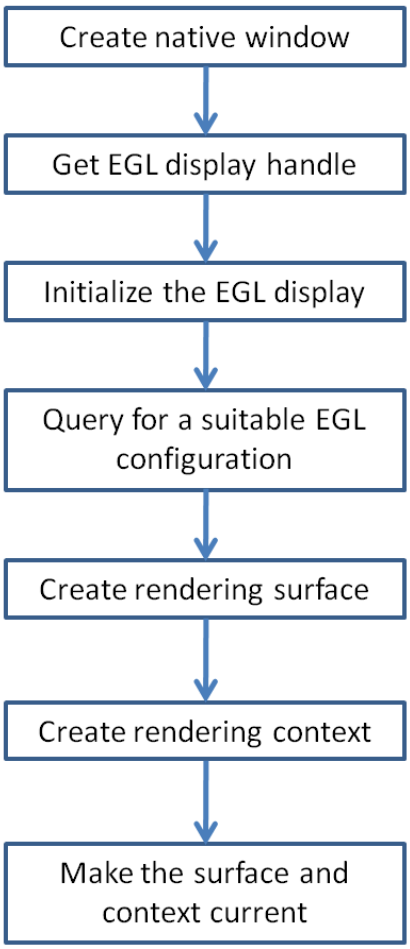


Figure.21 Process of Initialization (Setup Function)

After the initialization process is done, the second function called by main is 'CreateProgram'. This function handles the loading of shaders, compiling shaders, and link shaders to form the shader program. In this sample program, shader code is included in the CreateProgram function in literal strings. The shader source code is shown in Figure.22. Both the vertex shader and the fragment shader contains only several lines of code, so it can be included in the program. If the shaders contain a large amount of code, then they should be created as separate files, and be loaded by the program. In this project, the final rewritten XNA system will load shader files just as programmers use XNA on Windows. This will be explained later.

Figure.22 Shader Code of Ninja Sample Program

After the shader source code has been defined, the next step is to create shader objects and compile the shaders. As described in Chapter.3, shader object is an OpenGL ES object that can be used for attachment to a shader program object.[17] The shader object is created by the glCreateShader function. Then the shader source code can be loaded by using glShaderSource, and the shaders can be compiled by using glCompileShader. glGetShaderiv can be used to check if the shader has been successfully compiled. The code of loading and compiling shaders is shown in Figure.23. However, it is difficult to get syntax errors

```
GLint status;

// create and compile the vertex shader
GLuint vs;
vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vsSource, NULL);
glCompileShader(vs);
glGetShaderiv(vs, GL_COMPILE_STATUS, &status);
if (status == 0)
{
    printf("Failed to create a vertex shader.\n");
    glDeleteShader(vs);
    return GL_FALSE;
}

// create and compile the fragment shader
GLuint fs;
fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fsSource, NULL);
glCompileShader(fs);
glGetShaderiv(fs, GL_COMPILE_STATUS, &status);
if (status == 0)
{
    printf("Failed to create a fragment shader.\n");
    glDeleteShader(vs);
    glDeleteShader(fs);
    return GL_FALSE;
}
```

Figure.23 Create Shader Objects and Compiling Shaders

Once the shaders are ready for use, a shader program object should be created using glCreateProgram. The shader objects are attached to the program object using glAttachShader. The next step is to link the shaders together. Linking a program object is accomplished using glLinkProgram.[17] The linking status can be checked by using glGetProgramiv. Finally the individual compiled shader code objects can be deleted, as they are no longer needed. The code for assembling and linking the shader program is shown in Figure.24.

```
// create the program and attach the shaders
GLuint po;
po = glCreateProgram();
if (po == 0)
{
    printf("Failed to create a program.\n");
    return GL_FALSE;
}
glAttachShader(po, vs);
glAttachShader(po, fs);

// link the program
glLinkProgram(po);
glGetProgramiv(po, GL_LINK_STATUS, &status);
if(!status)
{
    printf("Failed to link program.\n");
    glDeleteProgram(po);
    glDeleteShader(vs);
    glDeleteShader(fs);
    return GL_FALSE;
}
```

Figure.24 Linking Shaders

The last job of CreateProgram is to assign the locations of the attributes in the shaders (i.e. "to map a generic vertex attribute index to an attribute variable name in the shader"[17]). There are two ways to do that. The first, which is also the approach used in this project, is to let OpenGL ES 2.0 to bind the index to the attribute name by calling the glGetAttribLocation function. The other approach is to let the application bind the index to an attribute name by using glBindAttribLocation. As this thesis is not about how to program with OpenGL ES 2.0, the detail of the second approach will not be explained.

glGetAttribLocation returns the generic attribute index (which is actually an integer) bound to the attribute variable name. The index will later be used in another function to allocate a buffer to the attribute data. The code is shown in Figure.25. The "po" in the program is the shader program object. The Ninja program uses a structure (i.e. ctx.rs in the program) to hold these values. Later in the Render function, the three vertex streams will be bound to three attributes in the vertex shader.

```
ctx.rs.vertLoc     = glGetAttribLocation( ctx.rs.po, "vertPosition" );
ctx.rs.normalLoc   = glGetAttribLocation( ctx.rs.po, "normal" );
ctx.rs.texcoordLoc = glGetAttribLocation( ctx.rs.po, "texCoord0" );
```

Figure.25 Mapping Indices to Attributes

The process of the CreateProgram function is shown in Figure.26.

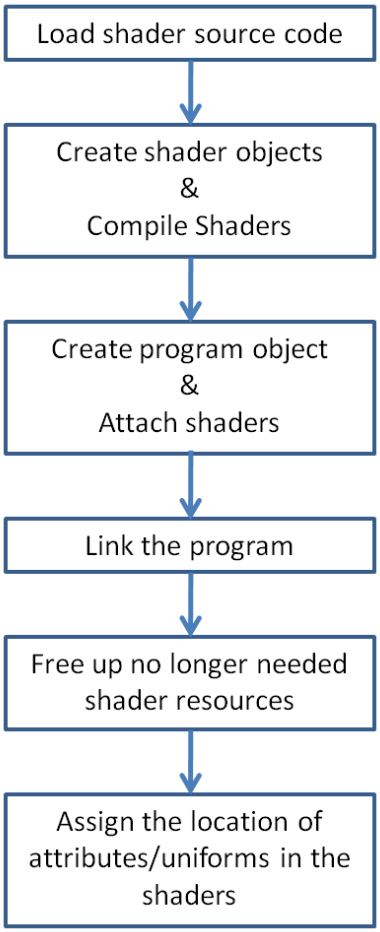


Figure.26 Process of CreateProgram

The last essential function called by main is the 'Render' function. Render is the function that actually draw objects on the screen, and it is called on every iteration of the main loop. The Ninja program implements a number of effects, so its Render function includes a lot of code. However, most of the code is not of importance for displaying objects. For example the transformation effect requires some calculation of matrices, but models can still be displayed without this effect. Therefore, in order to make the introduction more clear, this section only discusses the most important processes involved in Render function. Other OpenGL ES provided functions were also used in this project, and will be explained later in this thesis.

In the Ninja program, there are three vertex arrays storing vertex positions, vertex

normal and texture coordinates respectively.

The Render function mainly implements four tasks. It firstly uses glClearColor to set the color for the background, then calls glUseProgram to bind the shader program to the graphics device (i.e. load the shader code onto the GPU). Once the program is bound, glVertexAttribPointer is used to specify the vertex arrays (the vertex attribute indices acquired in CreateProgram is used as one of the parameters in this function), and glEnableVertexAttribArray is called to enable the generic vertex attribute array. The code is shown as Figure.27.

```
GLint posAttrib = ctx.rs.vertLoc;
GLint normAttrib = ctx.rs.normalLoc;
GLint uvAttrib = ctx.rs.texcoordLoc;

// set vertex pointers
glVertexAttribPointer(posAttrib, posSize, GL_FLOAT, GL_FALSE, 0, posPtr);
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(normAttrib, normSize, GL_FLOAT, GL_FALSE, 0, normPtr);
glEnableVertexAttribArray(normAttrib);
glVertexAttribPointer(uvAttrib, uvSize, GL_FLOAT, GL_FALSE, 0, uvPtr);
glEnableVertexAttribArray(uvAttrib);
```

Figure.27 Specifying Vertex Arrays

Figure.28 shows the way of specifying vertex attributes, and binding them to attribute names in the shader. This figure is referenced from the book "OpenGL ES 2.0 Programming Guide"[17].

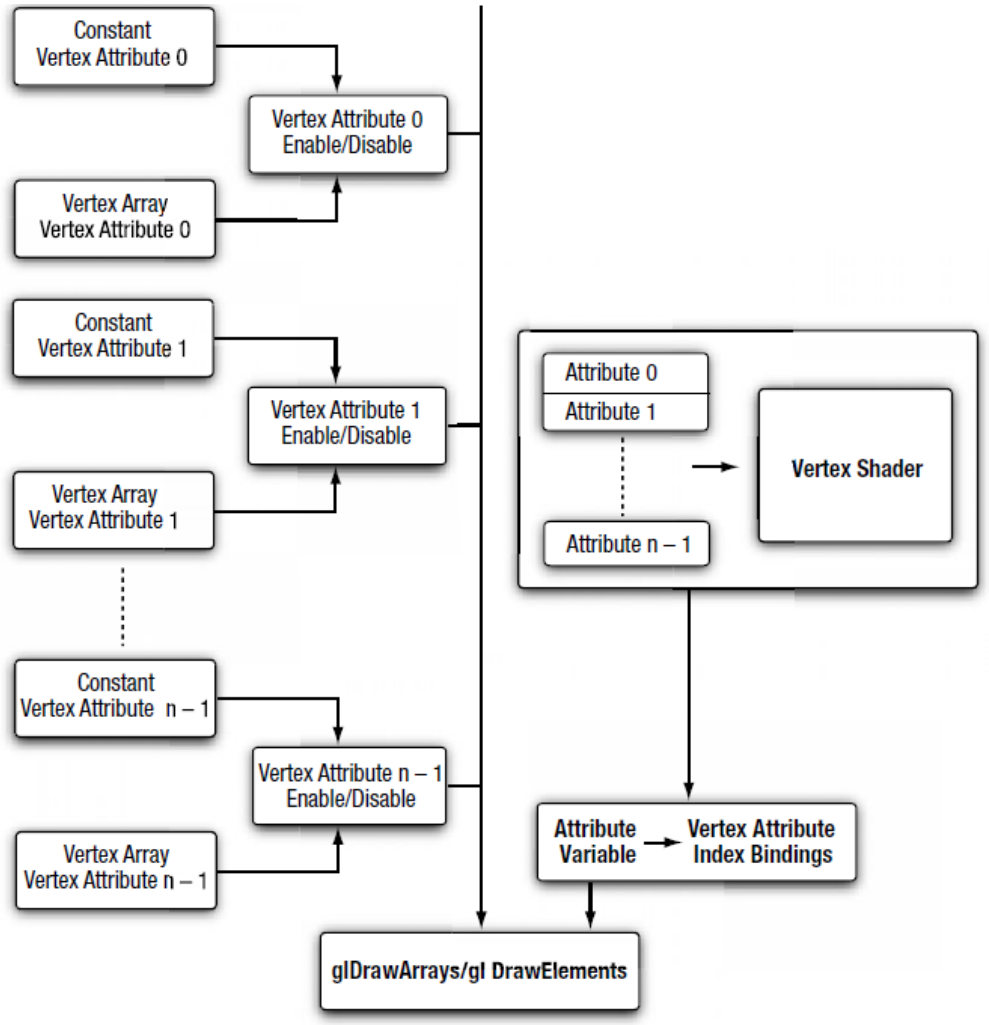


Figure.28 Specifying and Binding Vertex Attributes for Drawing Primitives

The last job of Render is calling glDrawArrays to draw primitives on the screen, and using eglSwapBuffers to flip the visible buffer.

All the other functions called by main are not important to the display, like detecting the mouse move and loading textures.

4.1.2 C++ Blank Window

The Ninja sample program was used as a reference to create the sample programs for this project. The best way of writing sample code is to start with creating something really simple. For example, displaying a blank window. When creating a new XNA project on Windows, programmers may execute the outline program provided by the system without doing any modifications to it, and a blank window will be displayed.

Therefore, the first sample program created in this project is to display a blank screen with OpenGL ES 2.0 using the C++ language. This program was built by modifying the Ninja program. Only three functions were kept, which are the Setup function, the Render function and the main function. All the other functions like loading textures were deleted from this program. As nothing needs to be displayed in the window, shaders were not needed, so the whole CreateProgram function was deleted as well. The Setup function only contains the initialization routines, so it was not modified. The code in the Render function was largely minified, as it does not need to do any transformation calculations, nor allocate buffers to attribute data. The current Render function only clears the background to a given color, and swaps buffers. The code is shown as Figure.29.

```
void Render(esContext &ctx)
{
    // Clear the color buffer
    glClearColor ( 0.0f, 1.0f, 1.0f, 0.0f );
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // flip the visible buffer
    eglSwapBuffers(ctx.eglDisplay, ctx.eglSurface);
}
```

Figure.29 Render Function for Displaying a Blank Window

The main function was not modified much, just deleting the invoke of CreateProgram. Figure.30 shows the execution effect of this program. It was possible to verify that the program was working as expected by altering the clear color and seeing that this changed the display properly.

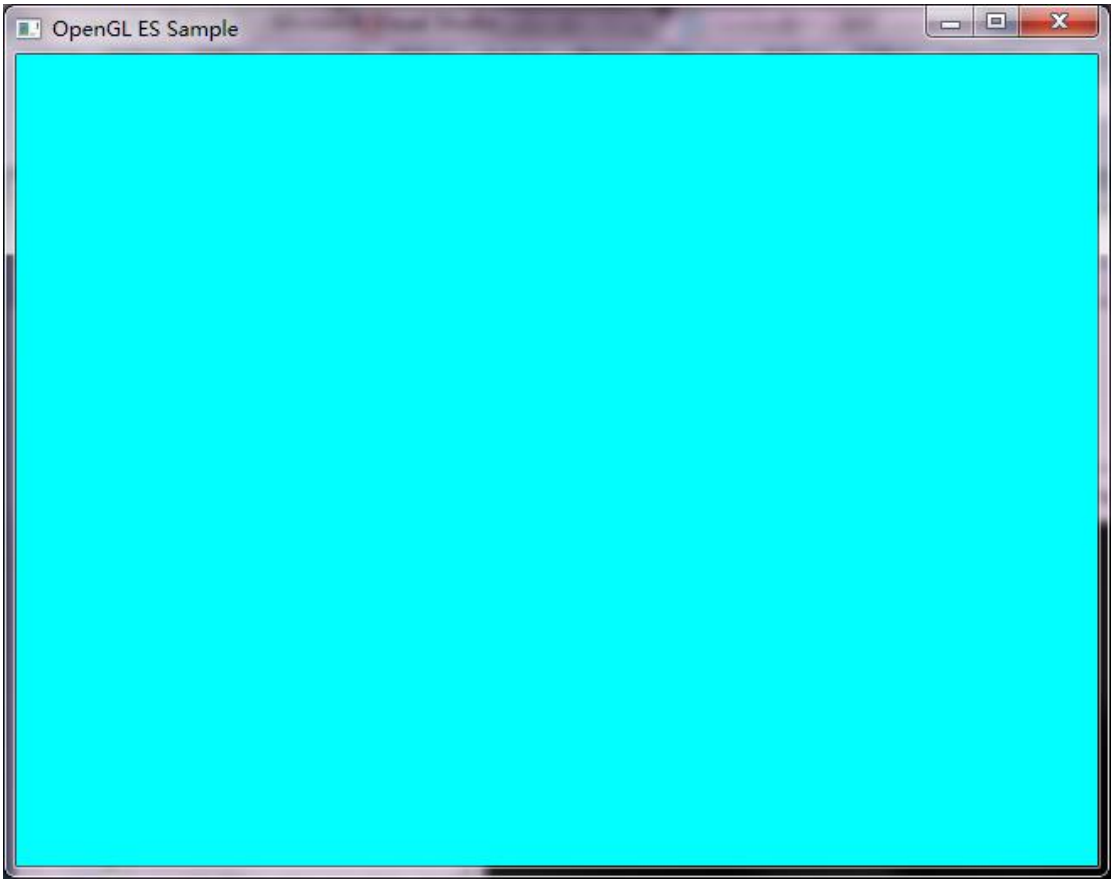


Figure.30 Displaying Blank Window with OpenGL ES 2.0 on Windows

4.1.3 C++ Triangle

Once the program for displaying a blank window can be successfully compiled, the next step is to display something on the window. As an object consists of a number of triangles, the simplest object to built is a single triangle.

The Setup function does not need to be changed. Compared to the blank window program, shaders are needed to display a triangle, so the CreateProgram function should be added. The shader source code is quite simple. In the vertex shader, the vertex data of the triangle is sent to an attribute named "vertPosition", and its value is passed to gl_Position without any other calculations. In our sample data, vertex coordinates were supplied in projection space, so no transformations were needed. In the fragment shader, a vector4 is created to define the value of gl_FragColor. The source code of shaders is shown in Figure.31.

Vertex Shader

```
const GLchar* vsSource =
  "attribute vec4 vertPosition;"
  "void main()"
  "{"
  "    gl_Position = vertPosition;"
  "}";
```

Fragment Shader

```
const GLchar* fsSource =
  "void main()"
  "{"
  "   gl_FragColor =  vec4(1, 1, 1, 0);"
  "}";
```

Figure.31 Shaders to Display a Triangle

The process of compiling shaders and linking the program does not change. At the end of CreateProgram, there is only one generic vertex attribute index that needs to be mapped, with the attribute vertPosition.

```
ctx.rs.vertLoc = glGetAttribLocation( ctx.rs.po, "vertPosition" );
```

In the Render function, the vertex coordinates are provided, and the index of vertPosition (vertLoc) is used to load the vertex data to the shader. Figure.32 shows the source code of the modified Render function.

```
void Render(esContext &ctx)
{
    // Vertex data
    GLfloat vVertices[] = {  0.0f,  0.5f, 0.0f,
                            -0.5f, -0.5f, 0.0f,
                             0.5f, -0.5f, 0.0f };

    // Clear the color buffer
    glClearColor ( 0.0f, 0.0f, 1.0f, 0.0f );
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Use the program (shaders) object
    glUseProgram ( ctx.rs.po );

    // Load the vertex data
    glVertexAttribPointer ( ctx.rs.vertLoc, 3, GL_FLOAT, GL_FALSE, 0, vVertices );
    glEnableVertexAttribArray ( ctx.rs.vertLoc );

    glDrawArrays ( GL_TRIANGLES, 0, 3 );


    // clean up state
    glUseProgram(0);

    // flip the visible buffer
    eglSwapBuffers(ctx.eglDisplay, ctx.eglSurface);
}
```

Figure.32 The Render Function for Displaying a Triangle

The main function is all the same as that of the Ninja program. Figure.33 shows the execution effect of the program. The white color of the triangle is as defined in the fragment shader. Again the function s of the program can be verified by changing color in the fragment code and positions in the vertex array.

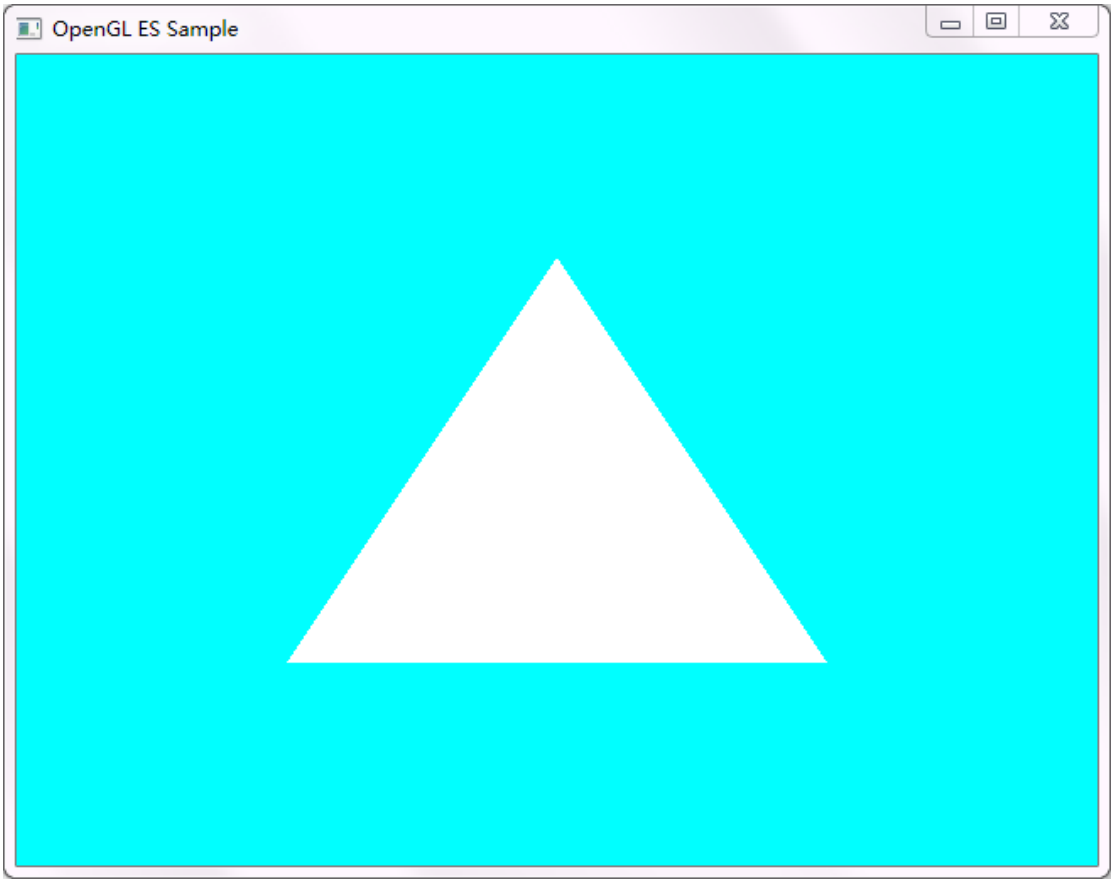


Figure.33 C++ Triangle on Windows

4.1.4 C++ Colored Rotating Cube

The triangle sample has successfully shown how to render a 2D object with OpenGL ES 2.0. In order to learn how to create and display 3D objects, the next step is to draw a colored rotating cube by extending the triangle program.

Before implementing the transformation and coloring effect, the first goal is trying to draw a cube on the window. There is not too much code to change. The Setup function, the CreateProgram function and the main function do not need to be modified at all (except the color of the cube defined in the fragment shader). In the Render function, the glDrawElements is used to draw primitives. This function asks for an array of indices as one of its parameters, so an array of indices should be

defined in Render. The code of the Render function for displaying a cube is shown in Figure.34.

```
void Render(esContext &ctx)
{
    // Vertex data
    GLfloat vVertices[] = {  0.0f,  0.0f, 0.0f,
                            -0.1f, 0.0f, 0.0f,
                            -0.1f, -0.1f, 0.0f,
                             0.0f,  -0.1f, 0.0f,
                             0.0f,  -0.1f, -0.1f,
                             0.0f, 0.0f, -0.1f,
                            -0.1f, 0.0f, -0.1f,
                            -0.1f,  -0.1f, -0.1f
                          };

    GLushort indices[36] = { 0, 1, 2, 0, 2, 3,
                             0, 3, 4, 0, 4, 5,
                             0, 5, 6, 0, 6, 1,
                             7, 6, 1, 7, 1, 2,
                             7, 4, 5, 7, 5, 6,
                             7, 2, 3, 7, 3, 4 };

    // Clear the color buffer
    glClearColor ( 0.0f, 1.0f, 1.0f, 0.0f );
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Use the program (shaders) object
    glUseProgram ( ctx.rs.po );

    // Load the vertex data
    glVertexAttribPointer ( ctx.rs.vertLoc, 3, GL_FLOAT, GL_FALSE, 0, vVertices );
    glEnableVertexAttribArray ( ctx.rs.vertLoc );

    glDrawElements(GL_TRIANGLES, sizeof(indices)/sizeof(GLushort),
                   GL_UNSIGNED_SHORT, indices);

    // clean up state
    glUseProgram(0);

    // flip the visible buffer
    eglSwapBuffers(ctx.eglDisplay, ctx.eglSurface);
}
```

Figure.34 Source Code of the Render Function for Displaying a Cube

The array of indices declares the sequence for drawing the vertices. There is one thing that should be noticed. As some graphics cards cannot use arrays of 32 bit integers as indices, the array of indices needs to be declared as 'GLushort' (16 bit) type. Figure.35 shows the index numbers on the cube.

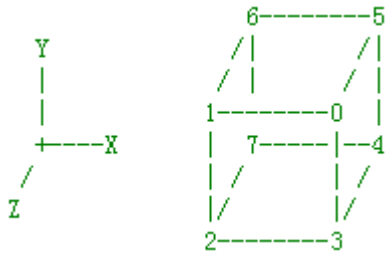


Figure.35 Index Numbers on the Cube

The effect of executing the program is shown as Figure.36.

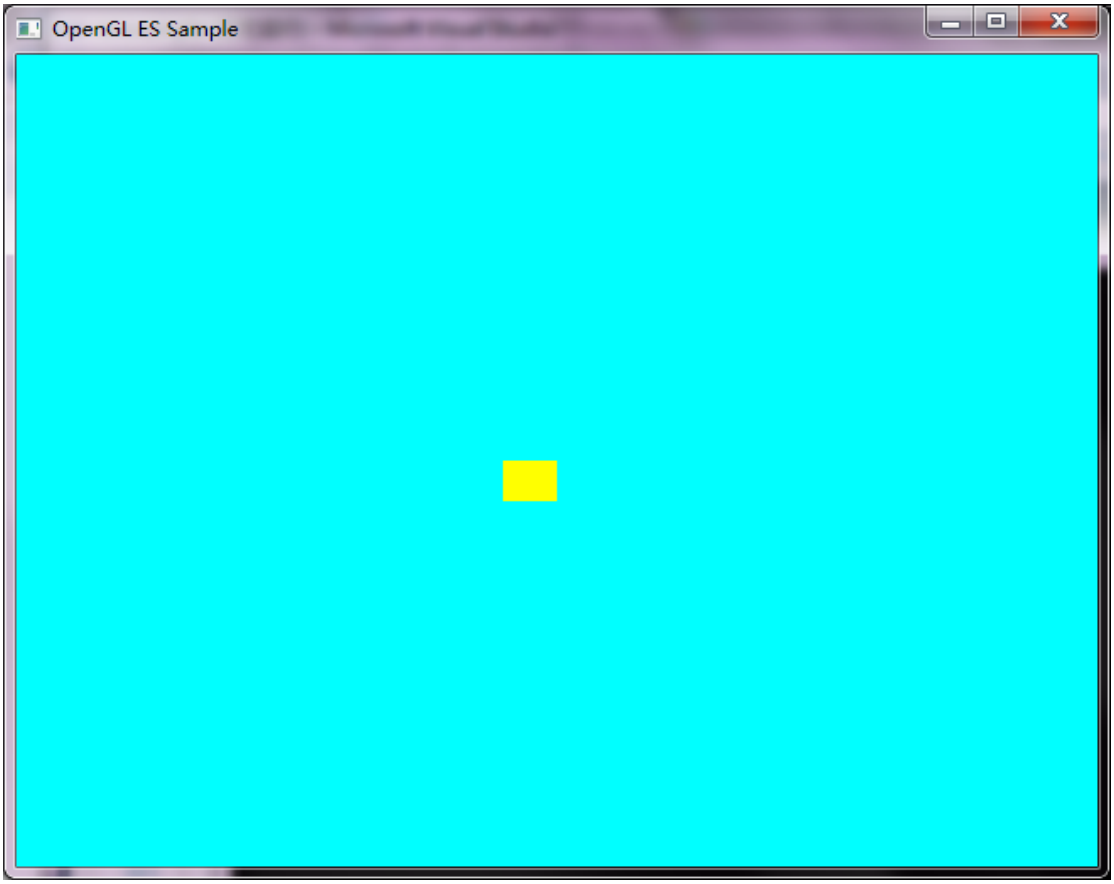


Figure.36 Displaying a Cube with OpenGL ES 2.0

At this stage, as neither the cube nor the view position can be changed, there is only a square shown on the window, but it is actually a cube. This can be checked by modifying some coordinates of the vertices. Once the cube was successfully rendered, color and transformation could be added to the program.

There are two changes that need to be done from the current program to display a colored rotating cube.

Firstly, in addition to an array of vertex coordinates, an array of vertex colors is also needed. Compared to former programs, the colored rotating cube program needs to allocate two buffers to store vertex attribute data, one for the vertex coordinate data, and the other for the vertex color data. The modified program used vertex buffer objects to solve this problem. The reason for using vertex buffer objects is that they ensure good performance for the program. When drawing primitives, the data of the vertex arrays that was originally stored in client (CPU) memory needs to be copied to the graphics memory (GPU). "Vertex buffer objects allow OpenGL ES 2.0 applications to allocate and cache vertex data in graphics memory and render from this memory, rather than copying vertex data on every iteration"[17].

Secondly, in the Render function, a calculated matrix is needed to implement transformations. This program uses three 4x4 matrices, which are known as the World, View and Projection matrices, to calculate the transformation matrix. The World matrix defines the position and orientation of the object, and it handles the rotation, scaling and translation of the object. The View matrix sets the viewport (i.e. the camera position). The Projection matrix describes how the object should be projected onto the screen. The transformation matrix is calculated by multiplying these three matrices in order, and then it is sent to the vertex shader as a uniform to be used to calculate the position of the vertices.

Figure.37, Figure.38 and Figure.39 describe the modifications of the source code based on the first cube program.

In CreateProgram, the data for vertex coordinate and vertex color are defined in two separate arrays. To use vertex buffer objects, glGenBuffers should be called to generate a buffer, followed by calling the glBindBuffer function to bind the buffer objects with the buffers (as shown in Figure.37, vbo_cube and vbo_cube_colors are two GLuint type variables, holding the index of each vertex buffer object). glBufferData is used to load data into the buffers. What is more, "mvpLoc" and "attribute_v_color" are calculated, representing the location of the transformation matrix and that of the vertex color in the vertex shader.

```
// Vertex data
GLfloat vVertices[] = {  0.5f,  0.5f, 0.5f,
                        -0.5f, 0.5f, 0.5f,
                        -0.5f, -0.5f, 0.5f,
                        0.5f,  -0.5f, 0.5f,
                        0.5f,  -0.5f, -0.5f,
                        0.5f, 0.5f, -0.5f,
                        -0.5f, 0.5f, -0.5f,
                        -0.5f,  -0.5f, -0.5f
                      };

//vertex buffer object
glGenBuffers(1, &vbo_cube);
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube);
glBufferData(GL_ARRAY_BUFFER, sizeof(vVertices),
            vVertices, GL_STATIC_DRAW);


GLfloat cube_colors[] = {
                        1.0, 1.0, 0.0,
                        0.0, 0.0, 1.0,
                        1.0, 0.0, 0.0,
                        0.0, 1.0, 1.0,
                        1.0, 1.0, 1.0,
                        0.0, 1.0, 0.0,
                        0.5, 0.5, 0.5,
                        0.7, 0.3, 1.0
                        };
glGenBuffers(1, &vbo_cube_colors);
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_colors);
glBufferData(GL_ARRAY_BUFFER, sizeof(cube_colors), cube_colors, GL_STATIC_DRAW);
```

Figure.37 Creating Vertex Buffer Objects

In shader source code, the uniform named mvpMatrix is the transformation matrix calculated from World, View and Projection matrices. It is used to calculate the value of gl_Position in vertex shader. Color data of the vertices is passed to the fragment shader in the f_color varying vector, to supply the color of each vertex.

```
const GLchar* vsSource =
  "uniform mat4 mvpMatrix;"
  "attribute vec4 vertPosition;"
  "attribute vec3 v_color;"
  "varying vec3 f_color;"
  "void main()"
  "{"
  "    gl_Position = mvpMatrix * vertPosition;"
  "    f_color = v_color;"
  "}";
const GLchar* fsSource =
  "varying vec3 f_color;"
  "void main()"
  "{"
  "gl_FragColor = vec4(f_color.x, f_color.y, f_color.z, 1.0);"
  "}";
```

Figure.38 Shader Source Code of Colored Rotating Cube

In the Render function, rebind the buffer first, and then specify vertex attribute data with glVertexAttribPointer. Note that the second call to glVertexAttribPointer has all its parameters listed with comments showing their purpose (this will be important later).

```
// Clear the color buffer
glClearColor ( 0.0f, 1.0f, 1.0f, 0.0f );
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Use the program (shaders) object
glUseProgram ( ctx.rs.po );
glUniformMatrix4fv(ctx.rs.mvpLoc, 1, GL_FALSE, &mvp.x.x);

// set state
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);

// Load the vertex data
glEnableVertexAttribArray ( ctx.rs.vertLoc );
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube);
glVertexAttribPointer ( ctx.rs.vertLoc, 3, GL_FLOAT, GL_FALSE, 0, 0 );


glEnableVertexAttribArray(attribute_v_color);
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_colors);
glVertexAttribPointer(
                      attribute_v_color, // attribute
                      3,                 // number of elements per vertex, here (r,g,b)
                      GL_FLOAT,          // the type of each element
                      GL_FALSE,          // take our values as-is
                      0,                 // no extra data between each position
                      0                  // offset of first element
                      );


glDrawElements(GL_TRIANGLES, sizeof(indices)/sizeof(GLushort),
               GL_UNSIGNED_SHORT, indices);
```

Figure.39 Rebinding Vertex Buffer Objects

Figure.40 shows the execution effect of colored rotating cube program. Again correct operation can be verified by altering colors and transformations.

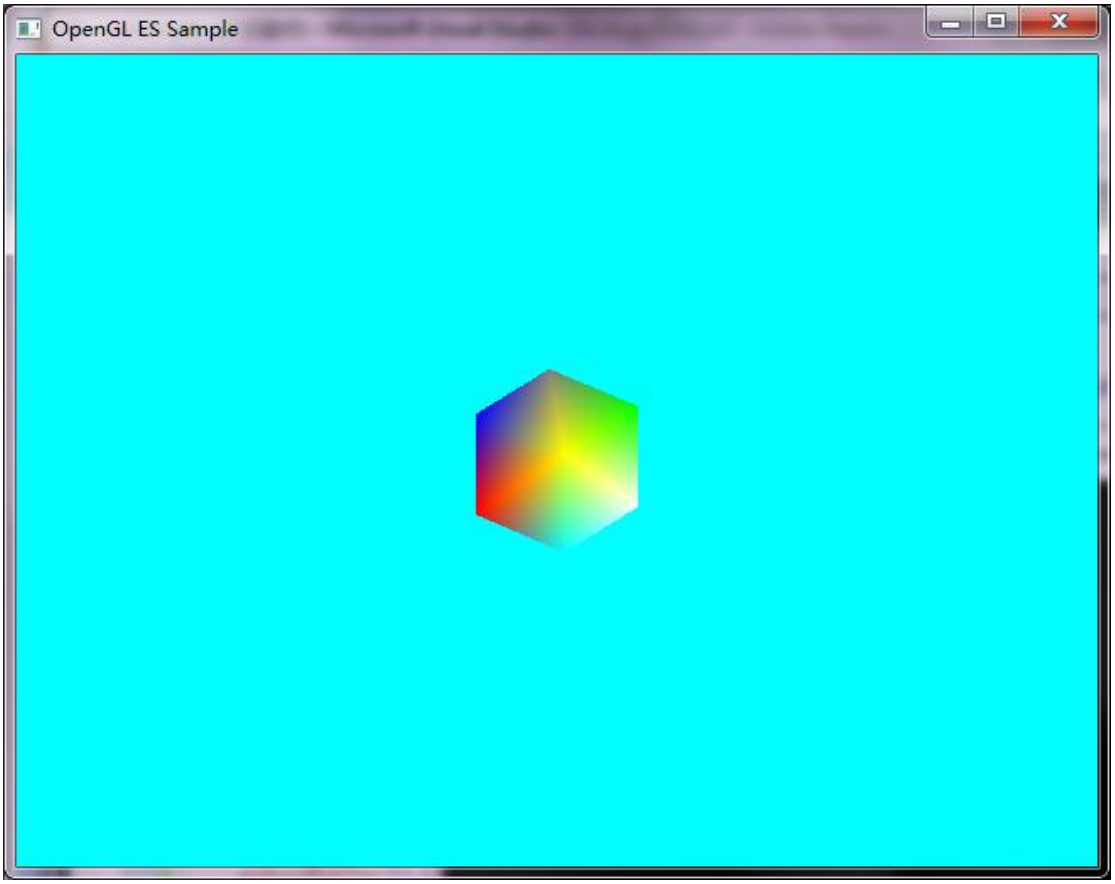


Figure.40 Colored Rotating Cube

4.1.5 C++ Textured Cube

In an XNA project, external files like images, videos and font can be loaded into the program by XNA's content manager. Therefore, the last sample program was built to determine how to load and paste textures on to a 3D object with OpenGL ES 2.0.

First of all, in the textured cube program, a new function named 'LoadTexture' was created to load the data of the texture. OpenGL ES 2.0 does not include functions for decompressing or interpreting different image file formats. This sample uses a texture file in .BMP format. .BMP files are not (usually) compressed, so this avoids issues with decompression. It was not difficult to write code to extract the raw pixel data from the file. This function specifies the path of the image file, and calls glTexImage2D to load the image data. glTexImage2D is a function provided by the

OpenGL ES 2.0, which requires a number of parameters to provide some information about the image. Some of the parameters are set with pre-defined values, like the type of the incoming pixel data, while the others, like the width and height of the image in pixels, and the actual pixel data of the image, need to be read from the file. As shown in Figure.41, four structures were created to read the image data. The width and height information are stored in the BMPInfoHeader structure, and the pixel data of the image is stored in an array of RGB structures. The texture data will later be sent to the fragment shader as a sampler2D uniform to calculate the color of the cube.

```
struct RGB {
  GLbyte blue;
  GLbyte green;
  GLbyte red;
  GLbyte alpha;
};

struct BMPInfoHeader {
  GLuint    size;
  GLuint    width;
  GLuint    height;
  GLushort  planes;
  GLushort  bits;
  GLuint    compression;
  GLuint    imageSize;
  GLuint    xScale;
  GLuint    yScale;
  GLuint    colors;
  GLuint    importantColors;
};

struct BMPHeader {
  GLushort  type;
  GLuint    size;
  GLushort  unused;
  GLushort  unused2;
  GLuint    offset;
};

struct BMPInfo {
  BMPInfoHeader     header;
  RGB               colors[1];
};
```

Figure.41 Structures for Reading Image Data

What is more, as textures are going to be pasted on each surface of the cube, the direction of pasting the textures need to be defined. This is done by adding an array of vertex texture coordinates in the Render function. The texture coordinates use the 2D

UV coordinate system to specify how the textures should be pasted on the surfaces. Figure.42 shows how the texture coordinates are declared. Each surface of the cube is drawn with two triangles. For example, as shown in Figure.42, the front face of the cube consists of triangle 0-1-2 and triangle 0-2-3. The first two members in the texture coordinates array, the "1,1,", specifies that the upper right corner of the texture should be pasted on the upper right corner of the front face. The texture coordinates array will be sent to the fragment shader to indicate the positioning of texture.

```
//                    6--------5
//     Y             /|       /|
//     |            / |      / |
//     |           1--------0  |      UV coord  0,1 ------ 1,1
//     +----X      |  7-----|--4                  |        |
//    /            | /      | /                   |        |
//   Z             |/       |/                    |        |
//                 2--------3                    0,0 ------ 1,0

GLfloat texture_coordinates[] = {
                                    1,1,//0
                                    0,1,//1
                                    0,0,//2
                                    1,1,//0     Front
                                    0,0,//2
                                    1,0,//3

                                    0,1,//0
                                    0,0,//3
                                    1,1,//5
                                    1,1,//5     Right
                                    0,0,//3
                                    1,0,//4

                                    1,0,//0
                                    0,0,//1
                                    1,1,//5
                                    0,0,//1     Up
                                    1,1,//5
                                    0,1,//6

                                    1,1,//1
                                    1,0,//2
                                    0,1,//6
                                    1,0,//2     Left
                                    0,1,//6
                                    0,0,//7

                                    0,0,//4
                                    0,1,//5
                                    1,1,//6
                                    0,0,//4     Back
                                    1,1,//6
                                    1,0,//7

                                    0,1,//2
                                    1,1,//3
                                    1,0,//4
                                    0,1,//2     Down
                                    1,0,//4
                                    0,0 //7
                                };
```

Figure.42 Texture Coordinates

The source code of the new shaders is shown in Figure.43. The texCoord0 attribute contains the texture coordinate data, and textureUnit0 references the texture sampler hardware and through that, the image data. The varying vTexCoord is used to pass texture coordinates from the vertex shader to the fragment shader.

```
const GLchar* vsSource =
  "uniform mat4 mvpMatrix;"
  "attribute vec4 vertPosition;"
  "attribute vec3 v_color;"
  "varying vec3 f_color;"
  "attribute vec2 texCoord0;"
  "varying vec2 vTexCoord;"
  "void main()"
  "{"
  "    gl_Position = mvpMatrix * vertPosition;"
  "    f_color = v_color;"
  "    vTexCoord   = texCoord0;"
  "}";

const GLchar* fsSource =
  "uniform sampler2D textureUnit0;"
  "varying vec2 vTexCoord;"
  "varying vec3 f_color;"
  "void main()"
  "{"
  "vec4 vertcolor = vec4(f_color.x, f_color.y, f_color.z, 1.0);"
  "  gl_FragColor =   texture(textureUnit0, vTexCoord).bgra;"
  "}";
```

Figure.43 Shaders for Displaying a Texture Cube

There is one problem that should be noticed when specifying vertex data in the Render function. As explained earlier, the sample code uses vertex buffer objects to allocate and cache data in graphics memory (i.e. in a GPU buffer) for both vertex coordinates and vertex colors. In this sample program, the texture coordinates were defined in the Render function, and did not use vertex buffer objects, which means the data of the texture coordinates is stored in the client memory (i.e. in a CPU buffer). There is no special reason for doing this other than to establish that mixed CPU/GPU buffers were possible in OpenGL ES. Managing buffer data is a major issue in this project as a whole, and as it turns out an issue did arrise. After specifying the data of vertex colors, the buffer is currently bound, and glBindBuffer with a 0 as its second parameter need to be called to remove the binding, so that UV data can be properly connected. Figure.44 shows how it was done in Render.

```
// Load the vertex data
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube);
glVertexAttribPointer ( ctx.rs.vertLoc, 3, GL_FLOAT, GL_FALSE, 0, 0 );
glEnableVertexAttribArray ( ctx.rs.vertLoc );

// Load the color data
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_colors);
glVertexAttribPointer(
                  attribute_v_color, // attribute
                  3,                 // number of elements per vertex, here (r,g,b)
                  GL_FLOAT,          // the type of each element
                  GL_FALSE,          // take our values as-is
                  0,                 // no extra data between each position
                  0                  // offset of first element
                  );
glEnableVertexAttribArray(attribute_v_color);

// Load the texture coordinate data
glBindBuffer(GL_ARRAY_BUFFER, 0);    // Remove the binding
glVertexAttribPointer(uvAttrib, uvSize, GL_FLOAT, GL_FALSE, 0, uvPtr);
glEnableVertexAttribArray(uvAttrib);
```

Figure.44 Specifying Vertex Data for Drawing a Textured Cube

The executing effect of textured cube is shown in Figure.45.

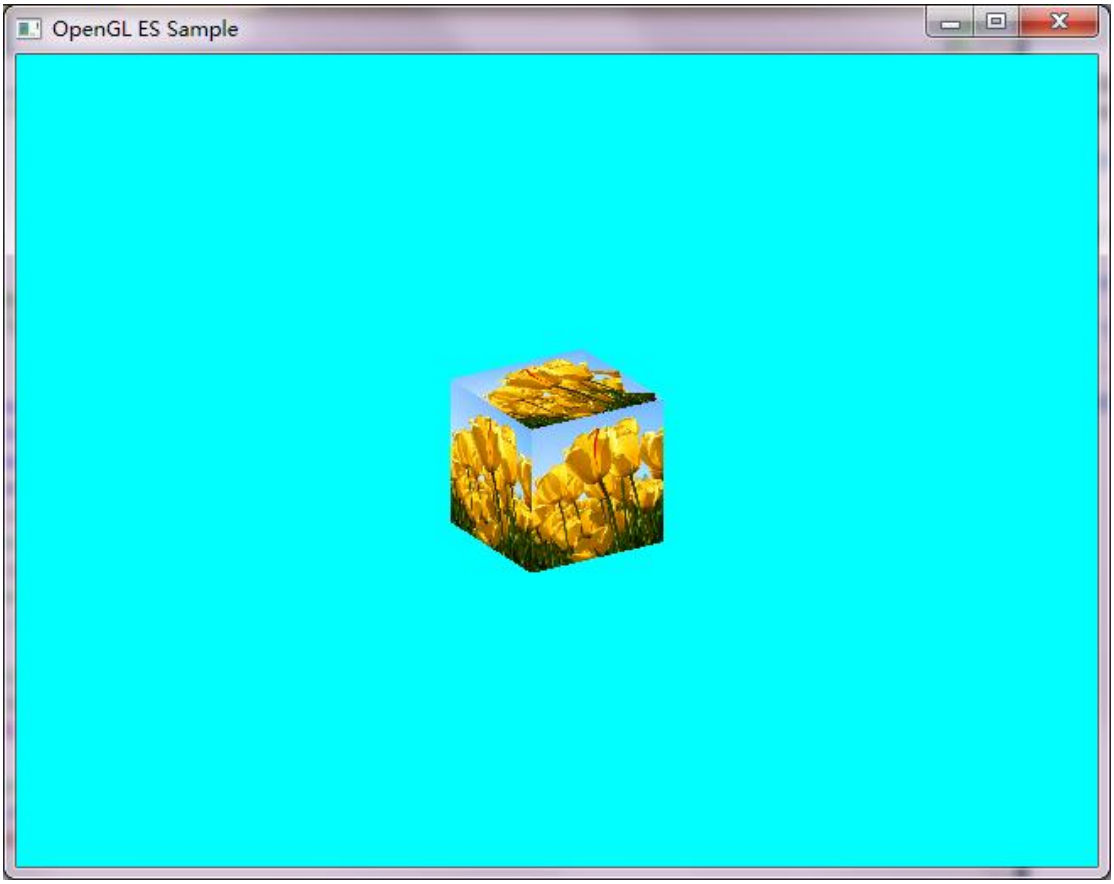


Figure.45 Textured Cube

4.2 C# Code with OpenGL ES 2.0 on Windows

XNA is written in C#, and XNA projects are all written in C# as well. Therefore, once the C++ sample programs have been acquired, the next step is to rewrite the sample programs with C# language (later on Raspberry Pi, Mono was used to compile and run the C# programs).

One of the challenges in this project is how to make procedure calls from OpenGL ES 2.0 in a C# program on Raspberry Pi. As described in section 4.1, the C++ programs use a number of "#include <header file name>" commands to get access to the headers and libraries. However, C# does not provide "#include" key word, so it is impossible to get access to OpenGL ES library with just one command line.

The way of solving this problem is to use the "DllImport" command to explicitly import every OpenGL ES function and every EGL function into the program. What is more, as the OpenGL ES 2.0 library is written in C language, the data types of the functions and their parameters need to be changed into C# defined types. For example, in initialization stage, the first EGL function called by the program is eglGetDisplay. Figure.46 shows the declarations of this function in the EGL library and in the C# program. The C language is excellent for defining data types. In the EGL library, the return value of eglGetDisplay is defined as a type named "EGLDisplay", which is actually a void pointer type. The parameter is declared as "EGLNativeDisplayType", which is an integer handle. In C program, they are two different data types. However, in the C# program, both two types become "IntPtr". In this way, all the EGL functions and OpenGL ES functions can be imported into the C# program. In C#, IntPtr is essentially an integer, but large enough to hold a pointer value. No arithmetic is permitted on an IntPtr value. With the Windows operating system, it is often used to hold handles or pointers when accessing unmanaged code.

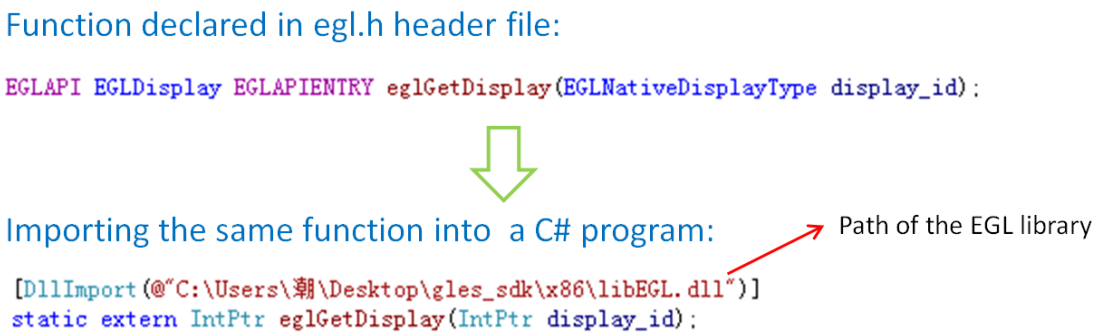


Figure.46 Importing an EGL Function into a C# Program

The type mappings developed and used in this project between EGL and C# are shown in Figure.47.

```
/* EGL Type Mappings
 *              EGLint          =>  either IntPtr (where no arithmetic should be allowed)
 *                                      or int  (this will be a 32 bit integer)
 *              EGLBoolean      =>  int (true and false are 1 and 0 respectively)
 *              EGLDisplay      =>  IntPtr (this is the way an HDC should be treated in C#)
 *              EGLConfig       =>  IntPtr (is a void pointer in C implementation)
 *              EGLSurface      =>  IntPtr      "                       "
 *              EGLContext      =>  IntPtr      "                       "
 *              GLbitfield      =>  uint
 *              GLclampf        =>  float
 *              char[]          =>  string
 * */
```

Figure.47 EGL Type Mappings

4.2.1 C# Blank Window

As when writing C++ sample programs, to experiment with coding an OpenGL ES program in the C# language, the first step was to create a program as simple as possible. The first C# sample program again displays a blank window to ensure an executable environment for the following programs.

Compared with the C++ program, the structure of the C# program does not change too much. It also starts by calling an Initialize function (which is known as the Setup function in C++ programs) to create the render surface and the window for display, then calls the Render function in the main loop to clear the color of the background and swap buffers to display the rendered surface. The Windows forms library, which provides access to display windows for C# programs gives access to the underlying Win32 "window handle" which can be passed as the EGLNativeDisplayType value. The content of each function has not changed, only the code was written in C#

language. As the functions of every procedure call from the OpenGL ES 2.0 library need to be declared in C# program, there are dozens of code lines declaring the imported functions and predefined values that are used in those functions. Furthermore, a function named 'Terminate' was added to the program to delete buffers and to release resources when needed.

Figure.48 and Figure.49 show the code. In the following sample programs (i.e. displaying triangle, cube and textured cube), with the increasing number of the OpenGL ES functions used to display objects, there will be even more command lines to import functions into the C# programs. As the way of importing functions are all the same, the screen shot of the importing source code will not be shown again.

The output display of the C# sample programs looks the same as that of the C++ programs. So screen shots from the C# programs will not be shown in this section.

```
// EGL Constants
IntPtr EGL_DEFAULT_DISPLAY = (IntPtr)0;
IntPtr EGL_NO_CONTEXT = (IntPtr)0;
IntPtr EGL_NO_DISPLAY = (IntPtr)0;
IntPtr EGL_NO_SURFACE = (IntPtr)0;
const int EGL_FALSE = 0;
const int EGL_TRUE = 1;

// ClearBufferMask
const uint GL_DEPTH_BUFFER_BIT   = 0x00000100;
const uint GL_STENCIL_BUFFER_BIT = 0x00000400;
const uint GL_COLOR_BUFFER_BIT   = 0x00004000;

Config attributes
const int EGL_BUFFER_SIZE               = 0x3020;
const int EGL_ALPHA_SIZE                = 0x3021;
const int EGL_BLUE_SIZE                 = 0x3022;
const int EGL_GREEN_SIZE                = 0x3023;
const int EGL_RED_SIZE                  = 0x3024;
const int EGL_DEPTH_SIZE                = 0x3025;
const int EGL_STENCIL_SIZE              = 0x3026;
const int EGL_CONFIG_CAVEAT             = 0x3027;
const int EGL_CONFIG_ID                 = 0x3028;
const int EGL_LEVEL                     = 0x3029;
const int EGL_MAX_PBUFFER_HEIGHT        = 0x302A;
const int EGL_MAX_PBUFFER_PIXELS        = 0x302B;
const int EGL_MAX_PBUFFER_WIDTH         = 0x302C;
const int EGL_NATIVE_RENDERABLE         = 0x302D;
const int EGL_NATIVE_VISUAL_ID          = 0x302E;
const int EGL_NATIVE_VISUAL_TYPE        = 0x302F;
const int EGL_SAMPLES                   = 0x3031;
const int EGL_SAMPLE_BUFFERS            = 0x3032;
const int EGL_SURFACE_TYPE              = 0x3033;
const int EGL_TRANSPARENT_TYPE          = 0x3034;
const int EGL_TRANSPARENT_BLUE_VALUE    = 0x3035;
const int EGL_TRANSPARENT_GREEN_VALUE   = 0x3036;
const int EGL_TRANSPARENT_RED_VALUE     = 0x3037;
const int EGL_NONE                      = 0x3038
const int EGL_BIND_TO_TEXTURE_RGB       = 0x3039;
const int EGL_BIND_TO_TEXTURE_RGBA      = 0x303A;
const int EGL_MIN_SWAP_INTERVAL         = 0x303B;
const int EGL_MAX_SWAP_INTERVAL         = 0x303C;
const int EGL_LUMINANCE_SIZE            = 0x303D;
const int EGL_ALPHA_MASK_SIZE           = 0x303E;
const int EGL_COLOR_BUFFER_TYPE         = 0x303F;
const int EGL_RENDERABLE_TYPE           = 0x3040;
const int EGL_MATCH_NATIVE_PIXMAP       = 0x3041
const int EGL_CONFORMANT                = 0x3042;
```

Figure.48 Predefined Variables

```
// EGL Procedures
[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern IntPtr eglGetDisplay(IntPtr display_id);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglInitialize(IntPtr dpy, out int major, out int minor);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglTerminate(IntPtr dpy);


[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglGetConfigs(IntPtr dpy, IntPtr[] configs, int config_size, out int num_config);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglChooseConfig(IntPtr dpy, int[] attrib_list, IntPtr[] configs, int config_size, out int num_config);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglGetConfigAttrib(IntPtr dpy, IntPtr config, int attribute, out int value);


[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern IntPtr eglCreateWindowSurface(IntPtr dpy, IntPtr config, IntPtr win, int[] attrib_list);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglDestroySurface(IntPtr dpy, IntPtr surface);


[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern IntPtr eglCreateContext(IntPtr dpy, IntPtr config, IntPtr share_context, int[] attrib_list);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglDestroyContext(IntPtr dpy, IntPtr ctx);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglMakeCurrent(IntPtr dpy, IntPtr surface_draw, IntPtr surface_read, IntPtr ctx);


[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern int eglSwapBuffers(IntPtr dpy, IntPtr surface);


// GL procedures
[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libGLESv2.dll")]
static extern void glClear(uint mask);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libGLESv2.dll")]
static extern void glClearColor(float red, float green, float blue, float alpha);
```

Figure.49 Importing Functions from EGL and OpenGL ES 2.0

4.2.2 C# Triangle

The C# triangle program has the same structure with the C++ triangle. A CreateProgram function was added to the program to handle the loading and compiling of the shaders, and linking the program. The process of the Initialize function, the CreateProgram function and the Render function are all the same with those of the C++ program. Shader source code did not change. An interesting issue when compiling the shader was passing a C# string value to a C char array. The C# marshalling code managed that transparently. Only the syntax of the programming language was changed to C#. The details of the code can be checked in the Appendix.

4.2.3 C# Colored Rotating Cube

The colored rotating cube sample was also modified from the C++ program. As vertex buffer objects are used in this program, the glVertexAttribPointer function need to be declared as two overloaded functions. The data type of its last parameter is different. When using VBO (vertex buffer objects), the last parameter is an integer indicating the buffer. If there is no VBO used, the last parameter should be an array of float that stores the vertex data. Figure.50 shows the overload of the glVertexAttribPointer function.

```
[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libGLESv2.dll")]
static extern void glVertexAttribPointer(int indx, int size, uint type, char normalized, int stride, int ptr);

[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libGLESv2.dll")]
static extern void glVertexAttribPointer(int indx, int size, uint type, char normalized, int stride, float[] ptr);
```

Figure.50 Overloaded Functions of glVertexAttribPointer

In this program, matrices and vectors are used to calculate transformations. However, rewriting the Matrix and the Vector classes involves a large amount of workload. Therefore, the XNA framework was used to provide mathematical classes to the program. These classes were later rewritten for the sample programs on Raspberry Pi.

4.2.4 C# Textured Cube

Compared to the C++ textured cube sample program, the LoadTexture function becomes much simpler in the C# program. It still uses glTexImage2D function to load textures. The image can be loaded by the FileStream function provided by the System namespace, and the width and height of the bitmap texture can be queried by using the Bitmap function. The pixel data of the image is acquired from calling the LockBits function (declared in the Bitmap class), and the data is copied into a byte array, which is later used as a parameter in glTexImage2D. Figure.51 shows the source code of the LoadTexture function.

Note the frequent use of Console.WriteLine. Correcting errors was a continuing source of difficulty during this project, and much of the sample code has frequent debug displays to show progress and warn of problems.

```
bool LoadTexture()
{

    int width, height;

    var bm = new System.IO.FileStream("./TulipSquare.bmp", System.IO.FileMode.Open);
    var bmp = new Bitmap(bm);

    width = bmp.Width;
    height = bmp.Height;

    Console.WriteLine("width = "+ width);
    Console.WriteLine("height = "+ height);

    byte[] pBits;
    var bitmapData = bmp.LockBits(new System.Drawing.Rectangle(0, 0, bmp.Width, bmp.Height) ,
                                  System.Drawing.Imaging.ImageLockMode.ReadOnly, bmp.PixelFormat);
    var length = bitmapData.Stride * bitmapData.Height;

    Console.WriteLine("Stride = " + bitmapData.Stride);
    Console.WriteLine("bitmapData.Height = " + bitmapData.Height);

    pBits = new byte[length];
    Console.WriteLine("length = " + length);

    //copy bitmap to byte[]
    Marshal.Copy(bitmapData.Scan0, pBits, 0, length);
    bmp.UnlockBits(bitmapData);


    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, pBits);

    return true;
}
```

Figure.51 LoadTexture Function in C# Program

4.3 C# Programs on Raspbian

Once all the C# sample programs were successfully executed on Windows, the next step was to copy these programs on the Raspberry Pi, further modifying them to make these C# programs executable on Pi. The introduction of how to set up Raspberry Pi, and how to make programs find the path of the libraries will be explained in the Appendix.

The Raspberry Pi had very simple OpenGL ES 1.0 and OpenGL ES 2.0 samples. These were not used directly, but porvided information on how to interface with the Raspberry Pi's native graphics system.

4.3.1 C# Blank Window on Raspbian

The program to display a blank window must be modified in several ways to be executable on Pi. Firstly, as Raspbian has a different display system from Windows, the Initialize function needed some change. In this project, the DispmanX API was used to handle the display on Raspberry Pi. Different from the XWindow system, DispmanX does not create any windows in the X-Window's sense. Instead it takes over an area of the screen (or all of the screen, it depends on users' setting) to display rendered objects. DispmanX is the display manager provided by Broadcom.

In the process of the Initialize function, the rendering surface is created by the eglCreateWindowSurface function. In the C# sample programs created on Winodws, the third parameter of this function was declared as an IntPtr variable, which was passed a Win32 window handle obtained from the .NET form window using the GetHandle function.

```
[DllImport(@"C:\Users\潮\Desktop\gles_sdk\x86\libEGL.dll")]
static extern IntPtr eglCreateWindowSurface(IntPtr dpy, IntPtr config, IntPtr win, int[] attrib_list);
```

By contrast, the data type of the third parameter in eglCreateWindowSurface was declared as EGL_DISPMANX_WINDOW_T on Raspbian.

```
[DllImport("libEGL")]
static extern IntPtr eglCreateWindowSurface(IntPtr dpy, IntPtr config, ref EGL_DISPMANX_WINDOW_T win, int[] attrib_list);
```

EGL_DISPMANX_WINDOW_T is a structure that contains three members.

```
public struct EGL_DISPMANX_WINDOW_T{
                                    public IntPtr element;
                                    public int width;
                                    public int height;
                                   }
```

The width and height represent the size of the rendering area. There is a graphics_get_display_size() function in DispManX that detects the resolution of the monitor, and display for full screen. In this project, the width and height of the rendering surface were hard coded as 640 and 480 respectively. The other member, element, is a handle to a dispmanx area, which can be acquired by calling the vc_dispmanx_element_add function.

Figure.52 shows the source code for creating a window surface on Raspberry Pi. The "nativewindow" in the code is an EGL_DISPMANX_WINDOW_T type structure.

```
int screen_width = 640;
int screen_height = 480;

dst_rect.x = 0;
dst_rect.y = 0;
dst_rect.width = screen_width;
dst_rect.height = screen_height;

src_rect.x = 0;
src_rect.y = 0;
src_rect.width = screen_width << 16;
src_rect.height = screen_height << 16;

dispman_display = vc_dispmanx_display_open( 0 /* LCD */);
dispman_update = vc_dispmanx_update_start( 0 );

dispman_element = vc_dispmanx_element_add ( dispman_update, dispman_display,
                                    0/*layer*/, ref dst_rect, 0/*src*/,
                                    ref src_rect,0 /*DISPMANX_PROTECTION_NONE*/,
                                    0 /*alpha*/, 0/*clamp*/, 0/*transform*/);

nativewindow.element = dispman_element;
nativewindow.width = screen_width;
nativewindow.height = screen_height;
vc_dispmanx_update_submit_sync( dispman_update );

    surface = eglCreateWindowSurface(display, configs[0], ref nativewindow, null);
```

Figure.52 Creating Window Surface on Raspberry Pi

The running effect of the modified "blank window" program is shown as Figure.53.

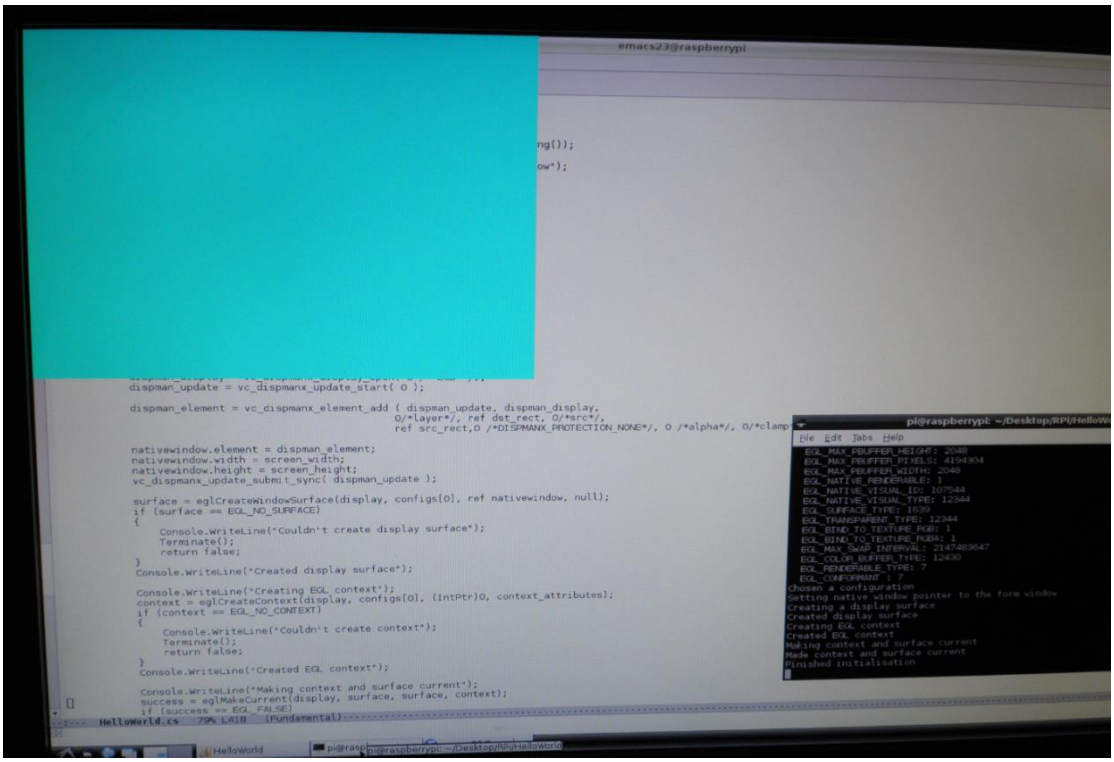


Figure.53 Blank Window on Raspberry Pi

As shown in Figure.53, the upper left corner of the screen displays a blank area with size 640 x 480, and there is no frame for the window. Instead of clicking a "Close" button to stop the program, users may press "Ctrl + C" to stop the test program and therefore close the window.

4.3.2 C# Triangle on Raspbian

To make the C# triangle sample program executable on Raspberry Pi, there is only one part of the source code that needs to be modified. In the Initialize function, once the display surface has been successfully created, the rendering context can then be created by calling the eglCreateContext function. The last parameter of this function is a list of attributes that specifies attributes and attribute values for the context being created, and the list has the same structure as described for eglChooseConfig (eglChooseConfig returns a list of EGL frame buffer configurations that match the attributes specified in attribute list). In programs created on Windows, this parameter can take "null" as its value (when there are no attributes recognized, this parameter will normally be null or empty as though the first attribute was EGL_NONE). However, this does not work on Raspberry Pi. It is possible that some platforms will define attributes specific to those environments, as an EGL extension. A non-null attribute list that is terminated with EGL_NONE will be passed to the underlying EGL implementation.[22] The context attributes need to be explicitly defined on Raspbian, or nothing will be displayed. The declaration of the attributes is shown in Figure.54.

```
int[] context_attributes= new int[3]
   {
    EGL_CONTEXT_CLIENT_VERSION, 2,
        EGL_NONE
   };
```

Figure.54 Context Attributes

EGL_CONTEXT_CLIENT_VERSION is followed by an integer "2" indicating an OpenGL ES 2.x context should be created.

Once the attributes are used to create the rendering context, the triangle program can be successfully executed on the Raspberry Pi. Figure.55 shows the running effect of this program.

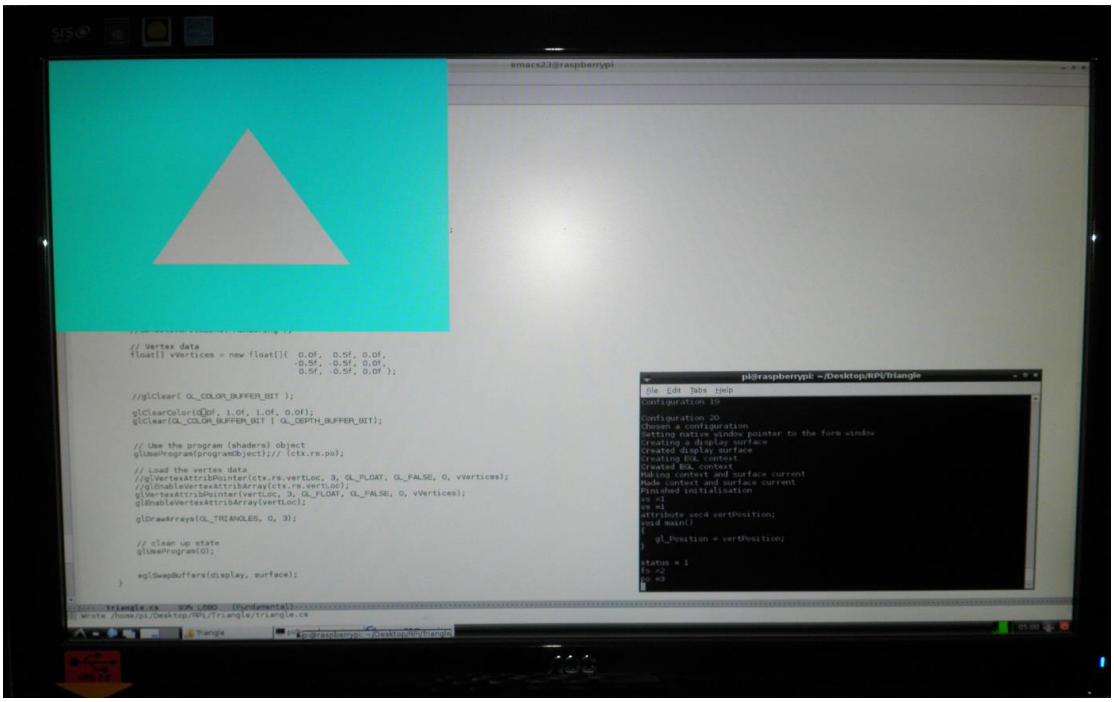


Figure.55 Triangle on Raspberry Pi

4.3.3 C# Colored Rotating Cube on Raspbian

The program that displays a blank window and the one displays a triangle only have one ".cs" document to be compiled. By contrast, the cube program implements transformation effects, so the rewritten XNA mathematical structures are needed to do the calculations. The modified mathematical structures include Vector2, Vector3, Vector4 and Matrix. The work of rewriting these structures are done by the supervisor of this thesis, Bill Rogers.

Although the C# programs created on Windows make procedure calls to the OpenGL ES 2.0 library, the shading language used in those programs is still HLSL. Compared with the C# cube program on Windows, the program on Raspberry Pi needed modified shader source code, as GLSL has a different syntax from HLSL. Note the last program's shaders were so simple (only vertex position was passed to the vertex shader) that the source code of last program's shaders were not given.

Figure.56 shows the differences between the shader source code. Firstly, there is no "mul" function provided in GLSL. Matrices can be simply multiplied with a "*" symbol. Secondly, the float number in GLSL do not have an "f" suffix. If the number is written as "1.0f" in GLSL, then the rendering of the objects will be failed and nothing can be displayed on the window.

HLSL:

```
string vsSource =
"uniform mat4 mvpMatrix;"+
"attribute vec4 vertPosition;"+
"attribute vec3 v_color;"+
"varying vec3 f_color;"+
"void main()"+
"{"+
"    gl_Position = mul(mvpMatrix, vertPosition);" +
"    f_color = v_color;" +
"}";

string fsSource =
"varying vec3 f_color;"+
"void main()" +
"{" +
"gl_FragColor = vec4(f_color.x, f_color.y, f_color.z, 1.0f);" +
"}\n";
```

GLSL:

```
string vsSource =
"uniform mat4 mvpMatrix;"+
"attribute vec4 vertPosition;"+
"attribute vec3 v_color;"+
"varying vec3 f_color;"+
"void main()"+
"{"+
"    gl_Position = mvpMatrix * vertPosition;" +
"    f_color = v_color;" +
"}";

string fsSource =
"varying vec3 f_color;"+
"void main()"+
"{"+
"gl_FragColor = vec4(f_color.x, f_color.y, f_color.z, 1.0);" +
"}\n";
```

Figure.56 Modified Shading Language

What is more, another important change of the program on Raspberry Pi is the sequence of drawing the vertices. The triangle primitives are assumed to face in a direction, which is defined by the order of the vertices. These primitives can be

discarded based on their apparent facing, and this process is known as Face Culling.[23] The default cull mode of OpenGL ES is anti-clockwise (i.e. when drawing a triangle primitive, the three vertices should be connected in an anti-clockwise order, this was not noted on the triangle program as the vertices of the triangle was drawn in the right direction by lucky), which is different from that of XNA. Therefore, the sequence of drawing vertices on each primitive should be modified.

When compiling the program on Raspberry Pi, the several ".cs" files should be placed in same folder. As there is no suitable IDE for compilation, the command line for compiling the program needs to contain all the file names (command "gmcs cube.cs Vector2.cs Vector3.cs Vector4.cs Matrix.cs" can be used in the Terminal to compile the cube program on Pi). Figure.57 shows the running effect of the colored rotating cube on Raspberry Pi.

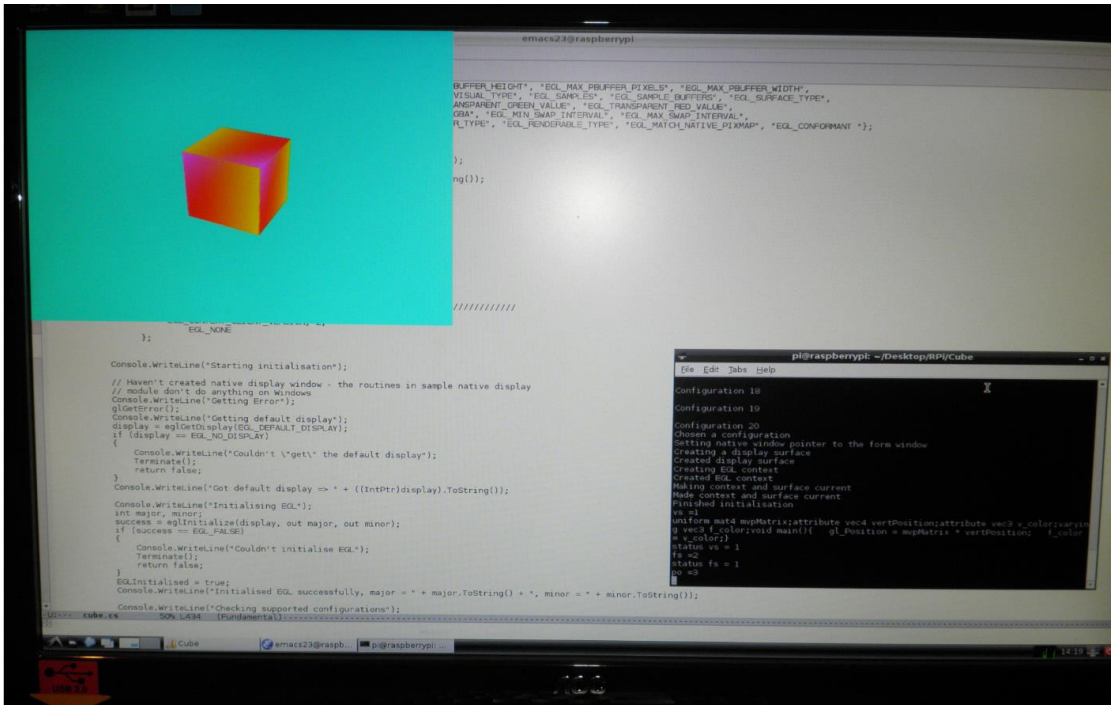


Figure.57 Colored Rotating Cube on Raspberry Pi

4.3.4 C# Textured Cube on Raspbian

As described in section 4.2.4, the LoadTexture function used in Windows programs read the image data by the Bitmap class. This class belongs to the System.Drawing namespace. However, Mono does not provide the Drawing namespace, so a new function is needed to load textures. The C++ on Windows program had its own code

for reading Bitmap files (see section 4.1.5). A new version of this code was written in C#.

The data of a bitmap image contains two parts, the FileHeader and the InfoHeader. The LoadTexture function created on Raspberry Pi put this starts by calling the BinaryReader function to get the data stream (the image should be copied into the same folder as the other files, so the path of the image can be found), then reads data of the FileHeader and the InfoHeader in separate byte arrays. The InfoHeader contains the width, height and pixel data of the image, which are used as parameters in the glTexImage2D function. The remainder of the file holds the pixel data. Figure.58 shows the source code of the new LoadTexture function.

```
bool LoadTexture()
{
    RPIbitmap rpi = new RPIbitmap();

    string filename;
    filename = "./TulipSquare.bmp";
    BinaryReader br = new BinaryReader(new FileStream(filename, FileMode.Open));

        ////////////////////////// Read BitmapFileHeader - 14 bytes //////////////////////////
        byte[] fileheader = new byte[14];
        br.Read(fileheader, 0, 14);
        // First two bytes should be ASCII codes for B and M
        Console.WriteLine("First two bytes as characters: " + Convert.ToChar(fileheader[0]) + Convert.ToChar(fileheader[1]));
        if (fileheader[0] != 66 || fileheader[1] != 77)
        {
            Console.WriteLine("ERROR: This is not a BMP file");
            br.Close();
            return false;
        }
        int filesize = getint32(fileheader, 2);
        Console.WriteLine("File size is " + filesize.ToString() + " bytes");
        // Ignore bytes 6, 7, 8, and 9
        int offset = getint32(fileheader, 10);
        Console.WriteLine("Offset to pixel data is " + offset.ToString() + " bytes");

        ////////////////////////// Read the BitmapInfoHeader //////////////////////
        byte[] sizebytes = new byte[4];
        br.Read(sizebytes, 0, 4);
        int headersize = getint32(sizebytes, 0);
        Console.WriteLine("Header size is " + headersize + " (40 means BITMAPINFOHEADER)");
        if (headersize < 40 || headersize == 64)
        {
            Console.WriteLine("ERROR: Cannot process this kind of BMP file");
            br.Close();
            return false;
        }
        byte[] restofheader = new byte[headersize - 4];
        br.Read(restofheader, 0, headersize - 4);
        rpi.width = getint32(restofheader, 0);
        rpi.height = getint32(restofheader, 4);
        Console.WriteLine("Width is " + rpi.width + ", Height is " + rpi.height);

        rpi.imagesize = getint32(restofheader, 16);
        Console.WriteLine("Image size is " + rpi.imagesize + " bytes");
    // Read the pixel data
    rpi.imagedata = new byte[rpi.imagesize];  // the imagedata array is the pBits[], i.e. the last param for glTexImage2D()
        br.Read(rpi.imagedata, 0, rpi.imagesize);
        Console.WriteLine("Read image data");

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, rpi.width, rpi.height, 0, GL_RGB, GL_UNSIGNED_BYTE, rpi.imagedata);

    return true;
}
```

Figure.58 LoadTexture Function for Raspberry Pi

The other parts of the program remains unchanged. Figure.59 shows the effect of running the textured cube program on Raspberry Pi.

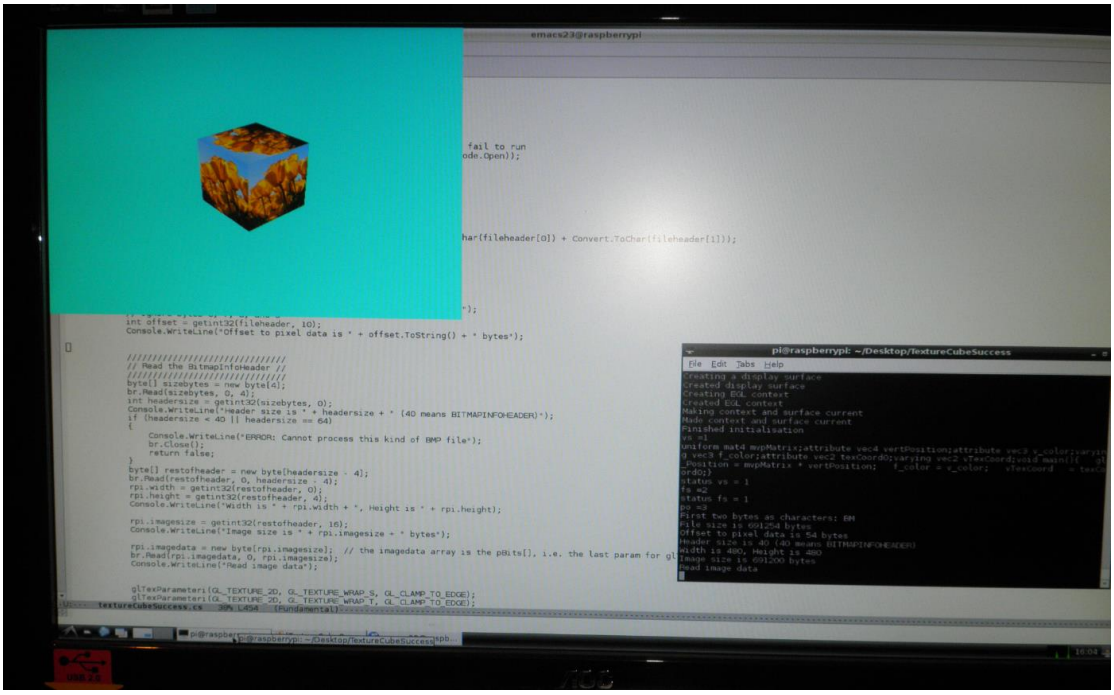


Figure.59 Textured Cube on Raspberry Pi

4.4 Rewriting XNA Classes

Once C# programs can be successfully run with OpenGL ES on Raspberry Pi, the last step is to make the code look like XNA programs. The program should have a Game1.cs class for users to put their code in, and also have other essential classes that control the processes of the program and provide underlying functions to the programmers.

As XNA contains a large number of classes, only the ones involved in the sample programs were rewritten. The source code of JBBRXG11 was used as a reference to write those classes. JBBRXG11 still uses some of the XNA framework, and those parts were also replaced with C# code in this project.

4.4.1 Display a Blank Window with Modified XNA Classes

Once again, starts with creating a blank window. Before writing the classes, the first step was to create a program with JBBRXG11 (almost identical to an equivalent XNA program) that displays a blank window, and set the color of the background into PaleGreen. As usual, the successful setting of windows color shows that the graphics code is working correctly. The classes involved in this program are Program, Game1, Game, GraphicsDevice, GraphicsDeviceManager, GameWindow, Gametime, Color,

DepthFormat and SurfaceFormat. In the JBBRXG11 system, these classes all either use routines from the XNA framework or DirectX. JBBRXG11 is built on XNA and only rewrote the parts that called DirectX methods. None of XNA is available on Raspberry Pi, so classes from the XNA framework must be redeveloped here. Classes that were part of JBBRXG11 are available in source code form and can be modified to use the Raspberry Pi graphics system (i.e. EGL, Dispmanx and OpenGL ES 2.0).

Data Structure:

The Program class is the class of the program that contains the Main function. Figure.60 shows the source code of Program. In the Main function, an instance of the Game1 class is created. Game1 inherits from the Game class. The Main function creates an instance of Game1 and calls the Run method (a method defined in the Game class. "This method is called to initialize the game, begin running the game loop, and start processing events for the game"[24]) through the instance of Game1. The Run method calls the GetGameManager method to access the instance of GraphicsDeviceManager that is created in the Game1 constructor, and added to the device manager list. In Game1 constructor, users may set the width and height of the game window through the PreferredBackBufferWidth property and the PreferredBackBufferHeight property defined in GraphicsDeviceManager. In the Game class, the value of these two properties is read via the GraphicsDeviceManager object, and passed as the parameters of the GameWindowInitialize method (defined in GameWindow) to initialize and create a new window. After that, in the game loop of Run method, the Draw method is called. The Draw method calls the Clear function (defined in GraphicsDevice) to set the color of the window.

```
1   using System;
2
3   namespace TestGame
4   {
5       static class Program
6       {
7           static void Main(string[] args)
8           {
9               using(Game1 game = new Game1())
10              {
11                  game.Run();
12              }
13          }
14      }
15  }
```

Figure.60 Program Class

Game1 Class:

Figure.61 shows the source code in modified Game1 class. When users programming with the modified XNA classes to display a blank screen, their code in Game1 class would be like this. Firstly declares an object of GraphicsDeviceManager, then defines the size of the window in the constructor of Game1, and finally calls the Clear method to set the color of the window in the Draw method. The code is the same with that programmed with XNA on Windows.

```
using System;
using JBBRXG11V2;

namespace TestGame
{
    public class Game1 : JBBRXG11V2.Game
    {
        GraphicsDeviceManager graphics;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            graphics.PreferredBackBufferHeight = 800;
            graphics.PreferredBackBufferWidth = 800;
        }

        protected override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.PaleGreen);
            base.Draw(gameTime);
        }
    }
}
```

Figure.61 Source Code in Game1 Class for Displaying a Blank Screen with JBBRXG11 on Raspberry Pi

Rewriting the GameWindow Class for Raspberry Pi:

The modified GameWindow class only contains a constructor, a read-only property that returns a handle of the window, and a SwapBuffers function.

The whole Initialize function of the rotating cube experimental C# program was copied into the constructor of GameWindow, as it contains all the instructions for creating rendering surfaces and displaying windows. Therefore, when creating a GameWindow object in the Game class, all the initialize routine can be done and window will be ready to use.

The read-only property "Handle" is supposed to return a handle that represents the window. Therefore, the value of dispman_element (acquired in the constructor) should be returned.

As described in previous sections, in the Render function of the C# programs, eglSwapBuffers is called in the program loop after objects have been drawn. In XNA programs, the Game class controls the process of the program. In every iteration of the game loop, after the Draw method is called, the buffers need to be swapped. However, the two parameters of eglSwapBuffers, display and surface, are declared in the constructor of GameWindow, so the SwapBuffers method is also defined in the GameWindow class (this method just calls eglSwapBuffers function to swap the buffers). In the Game class, SwapBuffers can be called through a GameWindow object.

Replacing the XNA GameTime Class:

The modified GameTime class contains overloaded constructors and three properties. Figure.62 shows the source code of the modified GameTime. The three read-only properties, ElapsedGameTime (the amount of elapsed game time since the last update), IsRunningSlowly (a bool value represents whether the game is running multiple updates this frame) and TotalGameTime (the amount of game time since the start of the game) are used in the Tick method in Game class to control the running speed of the game. The three overloaded constructors can be called to get the value of these properties. In this class, the value returned from the ElapsedGameTime property and the TotalGameTime property are declared as TimeSpan type, which is a structure defined in the System namespace. So this structure does not need to be modified.

```
using System;

namespace JBBRXG11V2
{
    public class GameTime
    {
        TimeSpan currentgametime, currentelapsedtime;
        bool isrunningslowly;


        // Summary: ...
        public GameTime()
        {
            currentgametime = currentelapsedtime = TimeSpan.Zero;
            isrunningslowly = false;
        }

        // ...
        public GameTime(TimeSpan totalGameTime, TimeSpan elapsedGameTime)
        {
            currentgametime = totalGameTime;
            currentelapsedtime = elapsedGameTime;
            isrunningslowly = false;
        }
        // ...
        public GameTime(TimeSpan totalGameTime, TimeSpan elapsedGameTime, bool isRunningSlowly)
        {
            currentgametime = totalGameTime;
            currentelapsedtime = elapsedGameTime;
            isrunningslowly = isRunningSlowly;
        }

        // Summary: ...
        public TimeSpan ElapsedGameTime { get { return currentelapsedtime; } }
        // ...
        public bool IsRunningSlowly { get { return isrunningslowly; } }
        // ...
        public TimeSpan TotalGameTime { get { return currentgametime; } }
    }
}
```

Figure.62 New GameTime Class

Replacing the XNA Color Class:

In the XNA's Color class, a large amount of source code is written to define the values of different colors. A color is represented by a four byte value, and the four bytes stand for R, G, B and A (red, green ,blue and alpha) respectively.

In the modified Color class, four properties were declared to read and write the value of RGBA of a color. The constructor just copies the value from four integers to the properties. As the purpose of this project is to investigate the possibility of programming with modified XNA classes on Raspberry Pi, it is not necessary to rewrite every part of the original classes in this project. Therefore, in the modified Color class, only one predefined color, the PaleGreen, was declared to show it is usable for the program (more colors can be predefined in the same way). There is another property named PackedValue that gets or sets the current color as a 4 byte

packed value. When reading the value of the color from PackedValue, the data should be read in an reversed order (i.e. in ABGR order) as it is written in the RGBA order. Once a byte is read, the packed value is moved to the left with 8 bits to read the next byte.

Figure.63 shows the source code of the new Color class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JBBRXG11V2
{
    public class Color
    {
        public byte R { get; set; }
        public byte G { get; set; }
        public byte B { get; set; }
        public byte A { get; set; }

        // Summary: ...
        public Color(int r, int g, int b, int a)
        {
            R = (byte)r;
            G = (byte)g;
            B = (byte)b;
            A = (byte)a;
        }

        public static Color PaleGreen { get { return new Color(196, 255, 196, 255); } }

        // ...
        public uint PackedValue
        {
            get
            {
                return (((uint)A << 8 | (uint)B) << 8 | (uint)G) << 8 | (uint)R;
            }
            set
            {
                uint temp = value;
                R = (byte)(value & 255);
                temp = temp >> 8;
                G = (byte)(value & 255);
                temp = temp >> 8;
                B = (byte)(value & 255);
                temp = temp >> 8;
                A = (byte)(value & 255);
            }
        }

    }
}
```

Figure.63 Modified Color Class

Modifying the GraphicsDevice Class:

The GraphicsDevice class contains a number of properties and functions (e.g. the functions used to draw primitives). However, at this stage, a lot of its routine can be temporarily deleted as no objects need to be drawn currently.

As explained in the modified Game1 class, the Clear function used to set the color of the window is called from a GraphicsDevice object. Therefore, the modified GraphicsDevice class contains a constructor and a Clear function. Figure.64 shows the source code of the modified GraphicsDevice class.

```
using System;
using System.Runtime.InteropServices;

namespace JBBRXG11V2
{
    public class GraphicsDevice
    {
        [DllImport("libGLESv2")]
        static extern void glClear(uint mask);

        [DllImport("libGLESv2")]
        static extern void glClearColor(float red, float green, float blue, float alpha);

        const uint GL_COLOR_BUFFER_BIT = 0x00004000;

        internal GraphicsDevice(bool iswin, int prefwidth, int prefheight,
                                SurfaceFormat prefbackbufferformat,
                                DepthFormat prefdepthstencilformat,
                                GameWindow gamewin)
        {
            bool IsWindowed = iswin; //used to determine if it is set to full screen

            int Width = (iswin ? prefwidth : 0);
            int Height = (iswin ? prefheight : 0);
            SurfaceFormat modedescFormat = prefbackbufferformat;
            DepthFormat depthbufferdescFormat = prefdepthstencilformat;

            IntPtr outputHandle = gamewin.Handle;

        }

        public void Clear(Color color)
        {
        Console.WriteLine("Clearing Color");

            float r, g, b, a;
            r = color.R * 1.0f / 255;
            g = color.G * 1.0f / 255;
            b = color.B * 1.0f / 255;
            a = color.A * 1.0f / 255;

            glClearColor(r, g, b, a);

            glClear(GL_COLOR_BUFFER_BIT);
        }
    }
}
```

Figure.64 Source Code of Modified GraphicsDevice

The constructor of the GraphicsDevice class sets the displaying window according to the parameters passed in. It determines whether the window should be displayed in full screen, the size of the window, the surface pixel format and the depth buffer format, and a handle of the window (the handle of the window is the "Handle" property declared in GameWindow, it returns the IntPtr "dispman_element"). The DepthFormat and the SurfaceFormat are two enumerations, are built as copies of the XNA versions. Figure.65 and Figure.66 shows the source code of SurfaceFormat and DepthFormat respectively.

The Clear function makes procedure calls to the OpenGL ES 2.0 library to set the color of background by the glClearColor function, and then the glClear function. As the predefined colors in the Color class use a 0-255 range value to define RGBA, while the glClearColor function takes four values ranging from 0 to 1, so the RGBA values of the colors must be divided 255 and converted to floats.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JBBRXG11V2
{
    // ...
    public enum SurfaceFormat
    {
        // ...
        Color = 0,
        // ...
        Bgr565 = 1,
        // ...
        Bgra5551 = 2,
        // ...
        Bgra4444 = 3,
        // ...
        Dxt1 = 4,
        // ...
        Dxt3 = 5,
        // ...
        Dxt5 = 6,
        // ...
        NormalizedByte2 = 7,
        // ...
        NormalizedByte4 = 8,
        // ...
        Rgba1010102 = 9,
        // ...
        Rg32 = 10,
        // ...
        Rgba64 = 11,
        // ...
        Alpha8 = 12,
        // ...
        Single = 13,
        // ...
        Vector2 = 14,
        // ...
        Vector4 = 15,
        // ...
        HalfSingle = 16,
        // ...
        HalfVector2 = 17,
        // ...
        HalfVector4 = 18,
        // ...
        HdrBlendable = 19,
    }
}
```

Figure.65 Source Code of SurfaceFormat

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JBBRXG11V2
{
    public enum DepthFormat
    {
        // ...
        None = 0,
        // ...
        Depth16 = 1,
        // ...
        Depth24 = 2,
        // ...
        Depth24Stencil8 = 3,
    }
}
```

Figure.66 Source Code of DepthFormat

Modifying the GraphicsDeviceManager Class:

The GraphicsDeviceManager uses another class named game_registration (as shown in Figure.67). game_registration is an object that is used to form a game "registration list". The list associates one instance of each of Game, GraphicsDeviceManager and GraphicsDevice. The reason for the list is that some of the XNA framework method implementations assume that these classes of a game can access information from each other. Method calls do not always provide the required access, so the registration list provides a "behind-the-scenes" linkage. This is not well developed in JBBRXG11, as facilities for games with multiple windows, or computers with multiple displays have not yet been implemented. The registration list provides some connections and the "hooks" for further development.

```
class game_registration
{
    public Game game;
    public GraphicsDeviceManager device_manager;
    public GraphicsDevice device;
    public game_registration(Game g, GraphicsDeviceManager m, GraphicsDevice dev)

    {
        Console.WriteLine("Game Registered");
        game = g;
        device_manager = m;
        device = dev;
    }
}
```

Figure.67 Source Code of game_registration

The modified GraphicsDeviceManager class declared two properties named PreferredBackBufferHeight and PreferredBackBufferWidth. As described earlier, users may define the size of the window by setting values to these properties in Game1.

The constructor of this class add the generated Game instances and the GraphicsDeviceManager instances into the registration list. It also provides default values of back buffer height and back buffer width to PreferredBackBufferHeight and PreferredBackBufferWidth respectively. Therefore, if users do not set the size of the window themselves, the window will be created with its default value (600 height, 800 width).

The internal static constructor establishes the registration list.

What is more, the modified GraphicsDeviceManager class declares a CreateDevice method to create an instance of GraphicsDevice and add it to the registration list. This method returns the created GraphicsDevice instance.

Modifying the Game Class:
The Game class is the most important class in an XNA project. It is the parent class of Game1, and controls the process of the whole program.

In the modified Game class for displaying a blank window, its constructor creates a GameWindow object and three variables for controlling the game time.

The Game class also provides a number of protected virtual methods that will be overriden in Game1 by the users. As described in Chapter 3, the four important methods for the users are Initialize, LoadContent, Update and Draw.

In addition to the virtual methods, Game class has a Tick method that controls the running speed of the game. In the modified Game class, the Update method, the Draw method and the SwapBuffers method (declared in GameWindow) are called by Tick, and Tick is called on every iteration of the game loop.

The most important method in Game class is the Run method. It handles the creation and registration of the GraphicsDeviceManager objects and GraphicsDevice objects. The Run method contains the game loop, but before that, a procedure call from the OpenGL ES 2.0 need to be done to invoke the bcm_host_init() function, because the Raspberry Pi requires that this function is called first before any GPU calls can be made. It then loads content (there is no content to be loaded for displaying a blank screen), resets the game time and create a window. Once the window has been created, the game loop starts and the Tick method is repeatedly called to draw the window.

Figure.68 shows the source code of the Run method in the modified Game class. The game loop was set to run for 1000 frames. The reason for the fixed number of frames is that the program was hard to stop and sometimes it was necessary to power off the Pi during development, especially when experimenting with full screen mode. This provides a safe experimental system. This can be changed to make the program keep running, and users may press "Ctrl + C" to stop the program.

```
public void Run()
{
    GraphicsDeviceManager dev_manager = GraphicsDeviceManager.GetGameManager(this);

    int height = dev_manager.PreferredBackBufferHeight;
    Console.WriteLine("PreferredBackBufferHeight is " + height);

    int width = dev_manager.PreferredBackBufferWidth;
    Console.WriteLine("PreferredBackBufferWidth is " + width);

    if (device != null)
        throw new Exception("DXna(NYI): Cannot handle multiple device managers");
    device = dev_manager.CreateDevice();

    bcm_host_init();

    window.GameWindowInitialize(height, width);

    Initialize();

    ResetElapsedTime();
    //Run window
    close_requested = false;

    BeginRun();

    int i = 0;
    while (i < 100) //(!close_requested)
    {
        Tick();
        i++;
    }
    EndRun();


    Window.Terminate();
}
```

Figure.68 Source Code of Modified Run Method

The effect of running is shown as Figure.69. This effect is different from that of the previous C# program. With the Game1 class, the color of the window was changed to "PaleGreen", and the size of the window has been successfully modified to 800 x 800, so it becomes a square window rather than a 600 x 800 rectangle window. This proves that all the modified classes are working well, and users may simply program their code in Game1 class to display a blank screen.

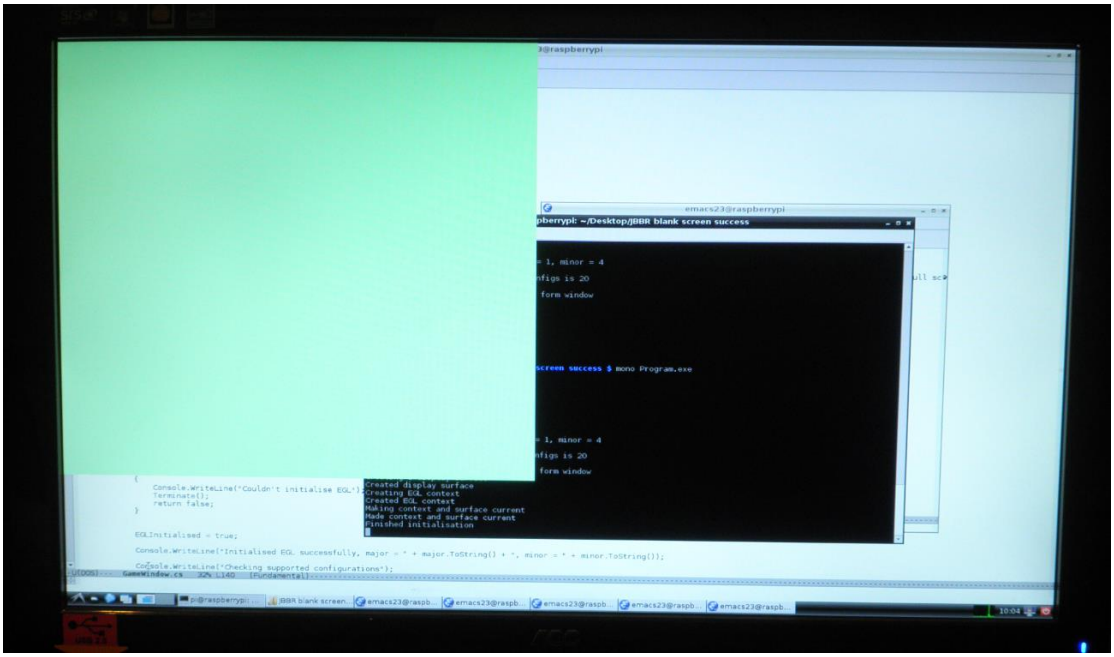


Figure.69 Blank Window Displayed by Modified XNA Classes

4.4.2 Display a Colored Triangle with Modified XNA Classes

At this stage, the modified classes allow users to create a blank window, and they can change the size and the background color of the window in the same way as programming XNA on Windows. The current task is to do further modifications to some of the existing classes, and add some new classes to allow users to draw a colored triangle in Game1 class.

The first step is to create an XNA project on Windows that draws a colored triangle to see how the user's code looks in Game1. In addition to the declaration of a GraphicsDeviceManager object, users need to declare a VertexPositionColor type array to store both vertex coordinate data and vertex color data. What is more, BasicEffect, which is the built-in shader program of XNA, should be used to draw the triangle.

Figure.70 shows the source code of the LoadContent method in Game1 for drawing a colored triangle. The values of vertex coordinates and vertex colors are separately assigned, and the color display of the vertices need to be enabled through the BasicEffect object.

```
protected override void LoadContent()
{
    vertices = new VertexPositionColor[3];
    vertices[0].Position = new Vector3(-0.5f, -0.5f, 0.5f);
    vertices[1].Position = new Vector3(0.0f, 0.5f, 0.5f);
    vertices[2].Position = new Vector3(0.5f, -0.5f, 0.5f);
    vertices[0].Color = Color.Red;
    vertices[1].Color = Color.Green;
    vertices[2].Color = Color.Blue;
    effect = new BasicEffect(GraphicsDevice);
    effect.VertexColorEnabled = true;
}
```

Figure.70 Source Code of the LoadContent method for Drawing Colored Triangle with XNA on Windows

In the Draw method, the DrawUserPrimitives<T> method is called to draw the triangle. This method is declared in the GraphicsDevice class, and takes four parameters (the primitive type, the vertex data, the vertex offset, and the number of primitive) to draw primitives. The "<T>" is a generic type, which indicates the data type of its second parameter.

The source code of the Draw method is shown in Figure.71.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    effect.CurrentTechnique.Passes[0].Apply();

    GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleList, vertices, 0, 1);
    base.Draw(gameTime);
}
```

Figure.71 Source Code of the Draw method for Drawing Colored Triangle with XNA on Windows

The running effect is shown as Figure.72

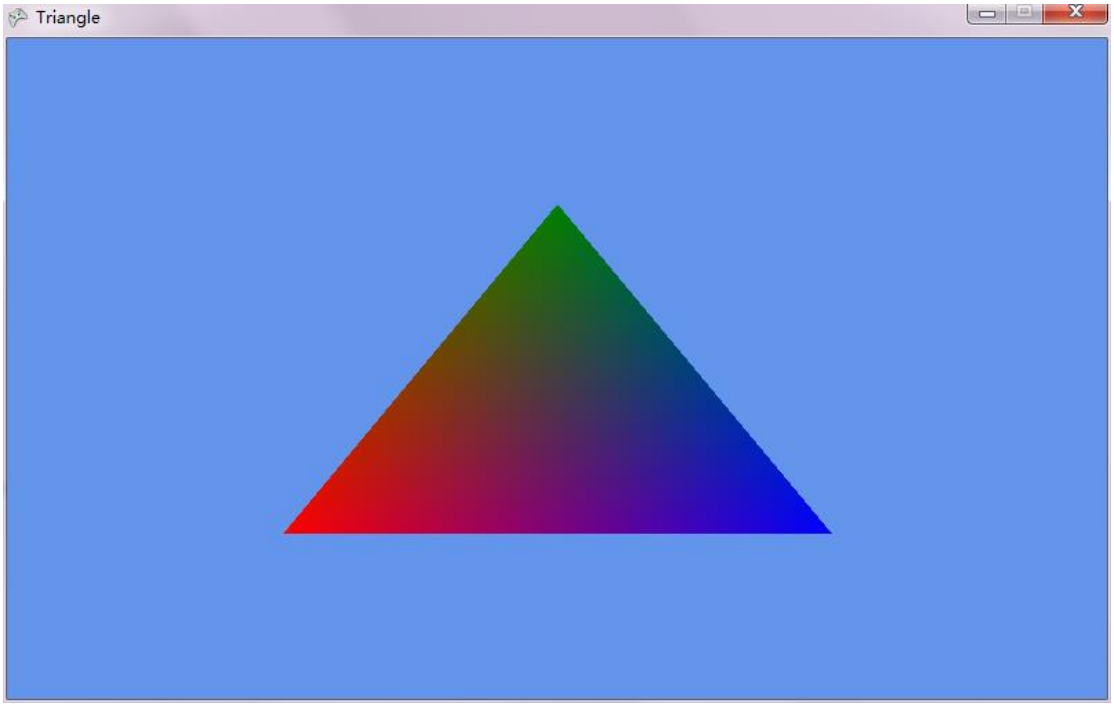


Figure.72 XNA Colored Triangle on Windows

The triangles drawn by the previous programs on Raspberry Pi were all in single colors, so before modifying the XNA classes, a new C# sample program that draws a colored triangle on Raspberry Pi was needed.

In the new colored triangle sample program, two vertex buffer objects were used to store the vertex position and vertex data respectively (like the colored cube program). Figure.73 shows the source code for using VBOs (vertex buffer objects) in the CreateProgram function.

```
float[] vVertices = new float[]{  0.0f,   0.5f, 0.0f,
                                 -0.5f, -0.5f, 0.0f,
                                  0.5f, -0.5f, 0.0f };
glGenBuffers(1, out vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, vVertices.Length * 4,
             vVertices, GL_STATIC_DRAW);

float[] cube_colors = new float[]{ 1.0f, 0.0f, 0.0f,
                                   0.0f, 1.0f, 0.0f,
                                   0.0f, 0.0f, 1.0f };
glGenBuffers(1, out vbo_colors);
glBindBuffer(GL_ARRAY_BUFFER, vbo_colors);
glBufferData(GL_ARRAY_BUFFER,
               cube_colors.Length * 4, cube_colors, GL_STATIC_DRAW);
```

Figure.73 VBOs of Colored Triangle

In the Render function, the vertex buffers are bound again, and then calls to glVertexAttribPointer are used to indicate the attribute location in vertex shader. The source code is shown as Figure.74.

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(vertLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(vertLoc);


glBindBuffer(GL_ARRAY_BUFFER, vbo_colors);
glVertexAttribPointer(
                      attribute_v_color, // attribute
                      3,                 // number of elements per vertex, here (r,g,b)
                      GL_FLOAT,          // the type of each element
                      GL_FALSE,          // take our values as-is
                      0,                 // no extra data between each position
                      0                  // offset of first element
                      );
glEnableVertexAttribArray((uint)attribute_v_color);


glDrawArrays(GL_TRIANGLES, 0, 3);
```

Figure.74 Using Two VBOs to Draw a Colored Triangle

In this way, a C# program that draws a colored triangle on Pi was acquired. The next step is to do further modifications to the Pi-XNA classes to allow users coding like an XNA program.

In order to draw a colored triangle, more classes are needed. Firstly, a VertexPositionColor structure was added to declare vertices with position and color. Secondly, when calling the DrawUserPrimitives method to draw primitives, its first parameter indicates the primitive type. Primitive types define the way of using vertices to draw primitives. Figure.75 shows some triangle primitive types supported by OpenGL ES 2.0 (this figure is referenced from the book "OpenGL ES 2.0 Programming Guide"). As shown in Figure.75, GL_TRIANGLES draws a series of separate triangles, GL_TRIANGLE_STRIP draws a series of connected triangles, and GL_TRIANGLE_FAN also draws a series of connected triangles, but they are based on one center vertex (the V0 vertex). Therefore, a PrimitiveType class is also needed.

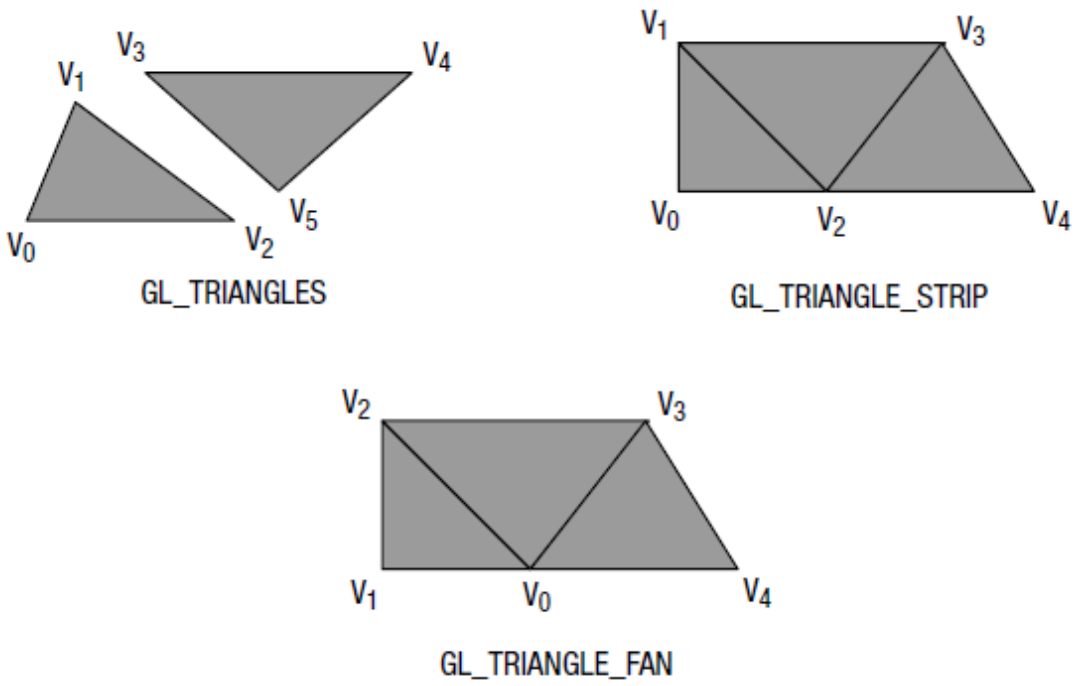


Figure.75 Triangle Primitive Types

Thirdly, as the vertex coordinates were declared as Vector3 in the users' XNA program, the mathematical classes are needed as well.

Modifying the VertexPositionColor Structure:

In this project, the VertexPositionColor structure only needs to contain a Color type variable to store the vertex colors, and a Vector3 variable to store the vertex positions. Its constructor just needs to pass the values of vertex data from the parameters to the fields declared in VertexPositionColor structure. Figure.76 shows the source code of VertexPositionColor.

```
1   using System;
2
3   namespace JBBRXG11V2
4   {
5       public struct VertexPositionColor
6       {
7
8           public Color Color;
9
10          public Vector3 Position;
11
12          public VertexPositionColor(Vector3 position, Color color)
13          {
14              Position = position;
15              Color = color;
16          }
17      }
18  }
19
```

Figure.76 Source Code of VertexPositionColor Class

In the Windows XNA framework, there are a number of vertex structures (e.g. VertexPositionNormalTexture) and conventions to allow users to add their own. Each structure provides a run time description in the form of an array of VertexElement structures. This description allows full implementation of the generic DrawUserPrimitives method in GraphicsDevice. For the Pi-XNA implementation, this would be possible, but is left to further work. Instead we demonstrate how one or two fixed vertex structures can be built and used.

Modifying the PrimitiveType Class:

PrimitiveType is an enumeration type. The PrimitiveType class in XNA is shown in Figure.77.

```
using System;

namespace Microsoft.Xna.Framework.Graphics
{
    public enum PrimitiveType
    {
        TriangleList = 0,
        TriangleStrip = 1,
        LineList = 2,
        LineStrip = 3,
    }
}
```

Figure.77 PrimitiveType of XNA

Compared with XNA, the primitive types are declared as macros in OpenGL ES 2.0. Figure.78 shows the declarations of primitive types in OpenGL ES 2.0.

```
/* BeginMode */
#define GL_POINTS                       0x0000
#define GL_LINES                        0x0001
#define GL_LINE_LOOP                    0x0002
#define GL_LINE_STRIP                   0x0003
#define GL_TRIANGLES                    0x0004
#define GL_TRIANGLE_STRIP               0x0005
#define GL_TRIANGLE_FAN                 0x0006
```

Figure.78 Primitive Types in OpenGL ES 2.0

In this project, the values of primitive types defined in OpenGL ES were copied to define the XNA primitive types. Figure.79 shows the modified PrimitiveType class. These values will be used in GraphicsDevice to decide how to draw primitives.

```
1   using System;
2
3   namespace JBBRXG11V2
4   {
5       public enum PrimitiveType
6       {
7           // 摘要: ...
11          TriangleList = 0x0004, //GL_TRIANGLES
12          // ...
17          TriangleStrip = 0x0005, //GL_TRIANGLE_STRIP
18          // ...
22          LineList = 0x0001, //GL_LINES
23          // ...
28          LineStrip = 0x0003, //GL_LINE_STRIP
29      }
30  }
31
```

Figure.79 Modified PrimitiveType Class

Modifying the GraphicsDevice Class:

The DrawUserPrimitives method is defined in the GraphicsDevice class. In the XNA program, BasicEffect allows users to use the XNA built-in shader, rather than writing shaders themselves. In previous programs created for the Raspberry Pi, the shaders were written in the CreateProgram function. At this stage, as the shaders cannot be loaded to the program from separate files, most of the code in CreateProgram was directly copied into the DrawUserPrimitives method.

What is more, the routine in the Render function that handles binding vertex buffers, indicating attribute locations in the vertex shader and drawing primitives was also moved to the DrawUserPrimitives method.

In the current DrawUserPrimitives method, the triangle primitives are actually drawn by the glDrawArrays function called from OpenGL ES. This function takes three parameters, the first parameter specifies the primitive type to render, the second one specifies the starting vertex index in the enabled vertex arrays, and the last parameter specifies the number of vertices to be drawn.

In the DrawUserPrimitives method, the first parameter of glDrawArrays uses the value of the first parameter of DrawUserPrimitives, which indicates the primitive type. The second parameter of glDrawArrays uses the third parameter of DrawUserPrimitives, which represents the vertex offset (the value is 0 in this program). The value of the last parameter of glDrawArrays, the number of vertices, depends on the primitive type. When the program draws Triangle List, the number of indices three times the number of the primitives, as each triangle contains three vertices. If the primitive type is Triangle Strip, then the number of the vertices equals that of the primitives plus two, as each two conjoint triangles share two vertices. Figure.80 shows the source code of calculating the number of vertices (the variable named "count").

```
int count=0;//it is used as a parameter in glDrawArrays()
uint type = 0;

if (primitiveType.ToString() == "TriangleList")
{
    count = primitiveCount * 3;
    type = GL_TRIANGLES;
    Console.WriteLine("type = " + type);

}
else if (primitiveType.ToString() == "TriangleStrip")
{
    count = primitiveCount + 2;
    type = GL_TRIANGLE_STRIP;
}
else if (primitiveType.ToString() == "LineList")
{
    count = primitiveCount * 2;
    type = GL_LINES;
}
else if (primitiveType.ToString() == "LineStrip")
{
    count = primitiveCount + 1;
    type = GL_LINE_STRIP;
}
```

Figure.80 Calculating the Number of Vertices for glDrawArrays

The process of the DrawUserPrimitives method is shown in Figure.81.

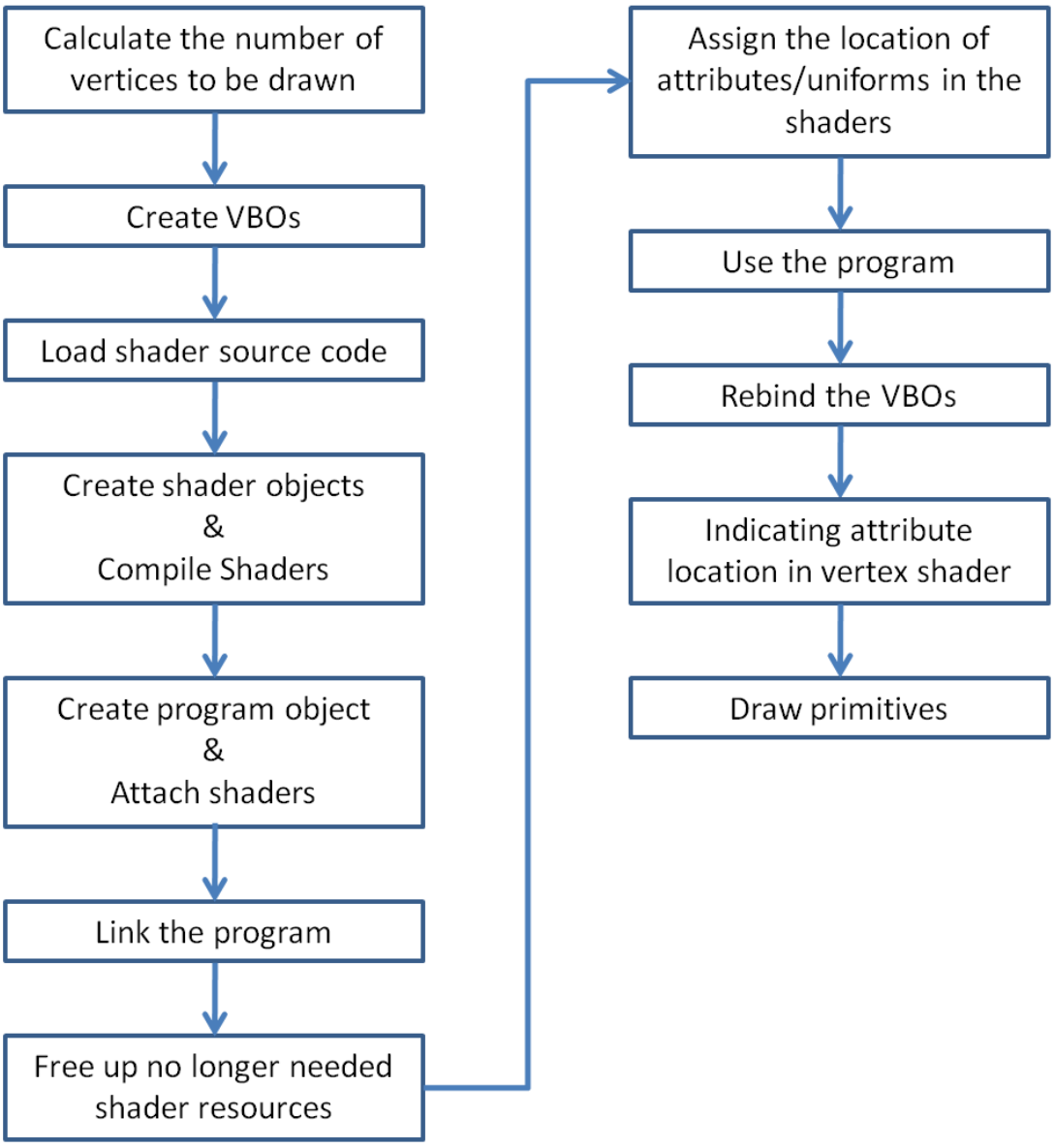


Figure.81 Process of the DrawUserPrimitives Method

The current DrawUserPrimitives method still has two problems.

First of all, the way of storing vertex data between XNA and the current program is different. The vertices are declared as VertexPositionColor type. The way the vertex data is stored in this structure is shown as Figure.82. Each vertex contains two parts, the vertex position and the vertex color, and they are stored alternately.

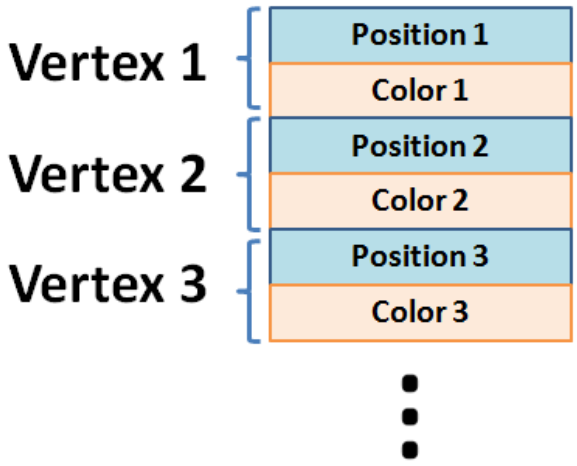


Figure.82 VertexPositionColor Data Structure

By contrast, in the current program, vertex position and vertex color are stored in two separate arrays, and the program uses two VBOs to cache the vertex data.

The way of solving this problem is quite easy. The glVertexAttribPointer function takes six parameters, the first parameter stands for the attribute location in vertex shader, the second parameter represents the number of elements in the attribute data, the third and the forth parameters represents the type of the element and whether the data value should be normalized to 0 - 1 respectively. The fifth parameter holds the spacing of attribute data items, and the last one indicates the offset of the first element. Thus, OpenGL ES allows a vertex buffer to be a series of values interleaved with other data.

The problem can be solved by simply modifying the value of the last two parameters of glVertexAttribPointer. Figure.83 shows the modified source code of the program. A vertex position is a Vector3 data that make up of three float numbers. Therefore, the number of its elements is 3. Vertex color data consist of four byte numbers, so the number of color elements is 4. The size of 4 bytes equals the size of 1 float, so the extra data between each vertex address is 4 * sizeof(float). The data structure starts with a vertex position data, so the offset value of the vertex position is 0. By contrast, the first vertex color data follows the first vertex position, so the offset value of the vertex color should be 3 * sizeof(float). The second call to glBindBuffer is not needed as both position and color data are accessed from the same physical buffer.

```
glVertexAttribPointer(vertLoc, 3, GL_FLOAT, GL_FALSE, 4  * sizeof(float), 0);
glEnableVertexAttribArray(vertLoc);


//glBindBuffer(GL_ARRAY_BUFFER, vbo_colors);
glVertexAttribPointer(
                      attribute_v_color, // attribute
                      4,                 // number of elements per vertex, here (r,g,b,a)
                      GL_UNSIGNED_BYTE,          // the type of each element
                      GL_TRUE,           // should it be normalized
                      4 * sizeof(float),                 // extra data between each vertex
                      3 * sizeof(float)                  // offset of first element
                      );
glEnableVertexAttribArray((uint)attribute_v_color);
```

Figure.83 Using Offset in glVertexAttribPointer

After using the offset parameter of glVertexAttribPointer, the program only needs to use one vertex buffer object to hold the vertex data.

The second problem is how to use generic types on Raspberry Pi (as the generic type "<T>" is used in the DrawUserPrimitives method on XNA). Note that this is not quite the same problem as fully managing variable vertex types. The issue here is passing data coming in as a generic array to the OpenGL ES code. C# does not provide an automatic way of passing generic arrays to native code, nor does it provide a way of mapping to C++ generics. In order to learn how it could be solved, a test program was created. Figure.84 shows the source code of the test program. This test program accesses data in an array of a structure named VertexXX from the Main function. In order to achieve the goal, pointers are needed. Pointers are considered as unsafe code in C# programs, so the class has to be declared as unsafe.

Test3 is the function that takes a generic type array as its parameter. The Main function uses the structure that contains the data as parameter to call Test3. Test3 uses `Marshal.UnsafeAddrOfPinnedArrayElement(data, 0);` command to set an pointer point to the actual data.
`GCHandle pinhandle = GCHandle.Alloc(data, GCHandleType.Pinned);` ensures a more correct way of doing the access to the data. The program will work without this command, but adding it should mark the memory as in use and fix its memory location in case of concurrent garbage collection while test runs. For the purposes of the experiment, Test3 calls Test (simulating a call to an external (native) function). Test simply copes the data into an array of floats and returns that array for display, to

show that the passing mechanism worked correctly.

```
using System;
using System.Runtime.InteropServices;

namespace Test
{
    unsafe class Program
    {
        struct VertexXX{
                 public float a,b,c;
                 public VertexXX(float pa, float pb, float pc){a = pa; b = pb; c = pc;}
                }

        static VertexXX[] vertices = new VertexXX[]{new VertexXX(1, 2, 3), new VertexXX(55, 44, 33)};

        static float[] testtwo(float* f)
        {
        float[] array2 = new float[6];
            for(int i = 0; i < 6; i++)
        //Console.WriteLine("Here is one: "+ (*(f + i)).ToString());
        array2[i] = *(f + i);
        return array2;
        }

        static float[] test(IntPtr f)
        {
        float[] array1 = new float[6];
            array1 = testtwo((float *)f);
        return array1;
        }

        static void test3<T>(T[] data)
        {
        float[] array = new float[6];

        GCHandle pinhandle = GCHandle.Alloc(data, GCHandleType.Pinned);
        IntPtr ptr = Marshal.UnsafeAddrOfPinnedArrayElement(data, 0);
        array = test(ptr);
        for(int i = 0; i < 6; i++)
        Console.WriteLine(array[i]);
        pinhandle.Free();
        }

        static void Main(string[] args)
        {
            test3(vertices);
        }
    }
}
```

Figure.84 Read data with generic type

Figure.85 shows the running result of this program. The data in the structure can be successfully read with generic type.

```
Here is one: 1
Here is one: 2
Here is one: 3
Here is one: 55
Here is one: 44
Here is one: 33
```

Figure.85 Result of the Test Program

The last step of modifying the GraphicsDevice class is to call

```
GCHandle pinhandle = GCHandle.Alloc(vertices, GCHandleType.Pinned);
IntPtr ptr = Marshal.UnsafeAddrOfPinnedArrayElement(vertices, 0);
```

commands to access the vertex data, and declare the Graphics Device as an unsafe public class.

Modifying the Game1 Class:

The source code in Game1 is quite similar to the XNA program. However, as there are no built-in shaders in the modified classes, the BasicEffect was not used in Game1. The source code of the modified Game1 class is shown as Figure.86.

```
using System;
using JBBRXG11V2;

namespace TestGame
{
    public class Game1 : JBBRXG11V2.Game
    {
        GraphicsDeviceManager graphics;

        VertexPositionColor[] vertices;
        //BasicEffect effect;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            graphics.PreferredBackBufferHeight = 800;
            graphics.PreferredBackBufferWidth = 800;
        }

        protected override void Initialize()
        {
            base.Initialize();
        }

        protected override void LoadContent()
        {
            vertices = new VertexPositionColor[3];
            vertices[0].Position = new Vector3(0, 0.5f, 0);
            vertices[1].Position = new Vector3(-0.5f, -0.5f, 0);
            vertices[2].Position = new Vector3(0.5f, -0.5f, 0);
            vertices[0].Color = Color.Red;
            vertices[1].Color = Color.Green;
            vertices[2].Color = Color.Blue;
            //effect = new BasicEffect(GraphicsDevice);
            //effect.VertexColorEnabled = true;
        }

        protected override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.PaleGreen);

            //effect.CurrentTechnique.Passes[0].Apply();

            GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleList, vertices, 0, 1);


            base.Draw(gameTime);
        }
    }
}
```

Figure.86 Modified Game1 Class for Drawing a Colored Triangle

Figure.87 shows the running effect on Raspberry Pi.

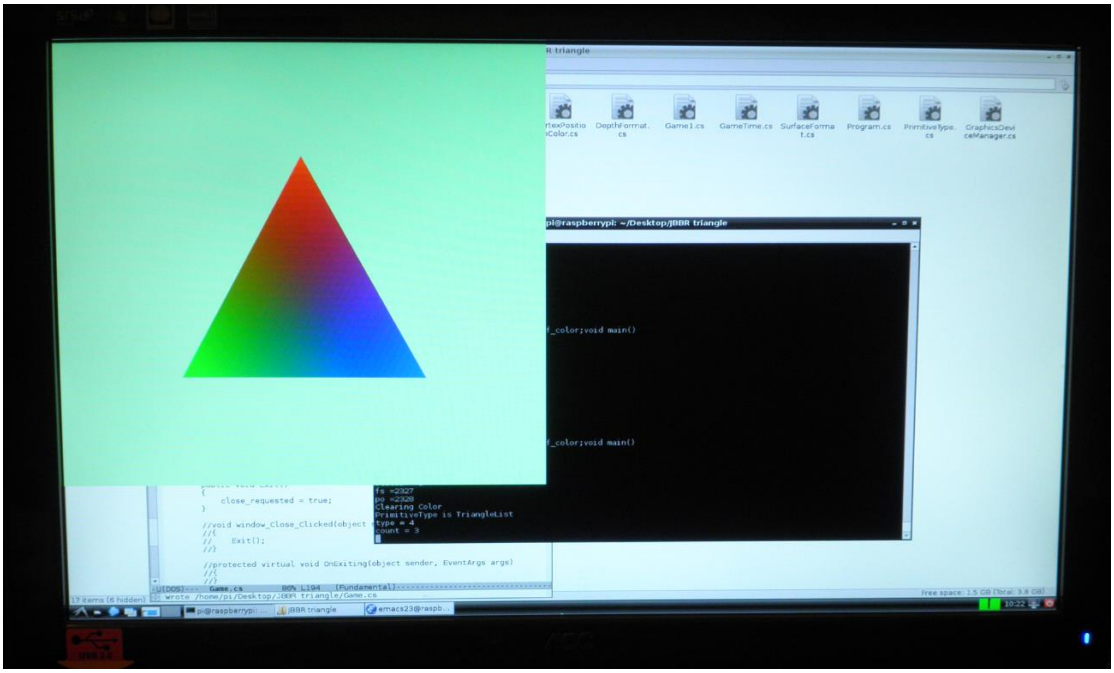


Figure.87Colored Triangle Displayed by Modified XNA Classes

4.4.3 Display a Colored Rotating Cube with Modified XNA Classes

Once the colored triangle can be successfully drawn in Game1 in a similar way with programming XNA on Windows, the next step is try to program a colored rotating cube with modified classes.

The first step is again writing an XNA program on Windows to display a colored rotating cube. In addition to the XNA triangle program, a short type array was declared to store the indices (as shown in Figure.88).

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;

VertexPositionColor[] vertices;
short[] indices;

BasicEffect effect;
```

Figure.88 Declaration of the XNA Colored Rotating Cube

In LoadContent method, vertex data and index data are given (note the vertices of each primitive should be drawn in clock-wise direction in XNA code). Figure.89 shows the source code of the LoadContent method.

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    vertices = new VertexPositionColor[8];

    vertices[0].Position = new Vector3(0.5f, 0.5f, 0.5f);
    vertices[1].Position = new Vector3(-0.5f, 0.5f, 0.5f);
    vertices[2].Position = new Vector3(-0.5f, -0.5f, 0.5f);
    vertices[3].Position = new Vector3(0.5f, -0.5f, 0.5f);
    vertices[4].Position = new Vector3(0.5f, -0.5f, -0.5f);
    vertices[5].Position = new Vector3(0.5f, 0.5f, -0.5f);
    vertices[6].Position = new Vector3(-0.5f, 0.5f, -0.5f);
    vertices[7].Position = new Vector3(-0.5f, -0.5f, -0.5f);

    //                    6--------5
    //    Y              /|       /|
    //    |             / |      / |
    //    |            1--------0  |
    //    +----X       |  7-----|--4
    //   /             | /      | /
    //  Z              |/       |/
    //                 2--------3

    vertices[0].Color = Color.Cyan;
    vertices[1].Color = Color.White;
    vertices[2].Color = Color.Blue;
    vertices[3].Color = Color.Magenta;
    vertices[4].Color = Color.Green;
    vertices[5].Color = Color.Yellow;
    vertices[6].Color = Color.Black;
    vertices[7].Color = Color.Red;

    indices = new short[]{ 1, 0, 2, 0, 3, 2,
                           3, 0, 4, 4, 0, 5,
                           5, 0, 6, 6, 0, 1,
                           6, 1, 7, 7, 1, 2,
                           4, 5, 7, 5, 6, 7,
                           2, 3, 4, 2, 4, 7 };

    effect = new BasicEffect(GraphicsDevice);
    effect.VertexColorEnabled = true;

}
```

Figure.89 LoadContent Method for Drawing a Colored Rotating Cube with XNA on Windows

In the Draw method, the World, View and Projection Matrices are provided to BasicEffect to specify the rotation, view position and projection; and finally the DrawUserIndexedPrimitives method (declared in GraphicsDevice) is called to draw

the cube. Figure.90 and Figure.91 show the source code of Draw method and the running effect respectively.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    //Matrix world, view, projection;

    effect.World = Matrix.CreateRotationX((float)gameTime.TotalGameTime.TotalSeconds);
    effect.View = Matrix.CreateLookAt(new Vector3(3, 2, 1.77f), new Vector3(0, 0, 0), Vector3.Up);
    effect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                                (float)Window.ClientBounds.Width / (float)Window.ClientBounds.Height,
                                                0.1f, 100.0f);

    foreach (EffectPass pass in effect.CurrentTechnique.Passes)
    {
        pass.Apply();
        GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionColor>
                (PrimitiveType.TriangleList, vertices, 0, 8, indices, 0, 12);
    }

    base.Draw(gameTime);
}
```

Figure.90 Draw Method for Displaying a Colored Rotating Cube with XNA on Windows

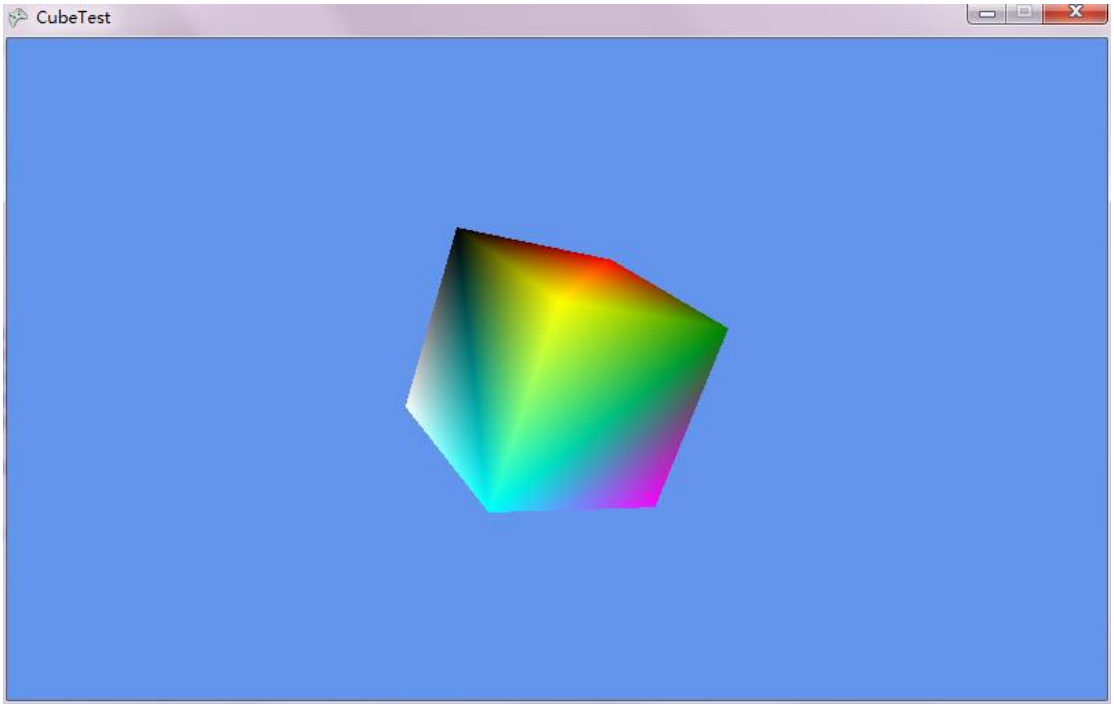


Figure.91 XNA Colored Rotating Cube on Windows

In the XNA program, an array of indices was used to indicate the drawing sequence of the vertices. In OpenGL ES, glDrawElements should be used to draw primitives with indices, as it takes an array of indices as one of its parameters (glDrawArrays does not).

The glDrawElements function takes four parameters, the first parameter specifies the primitive type, the second and the third parameters specify the number of indices and the data type of the indices respectively. The last parameter specifies the location where the indices are stored. On Raspbian, the indices for glDrawElements should be declared as an array of bytes, and the data type of indices should be specified as GL_UNSIGNED_BYTE. In GraphicsDevice, glDrawElements is declared as

```
static extern void glDrawElements (uint mode, int count, uint type, byte[] indices);
```

Modifying the Game1 Class:

In order to make the program in Game1 similar to that of the Windows XNA program, the modification starts with the Game1 class.

As users need to write shader programs themselves when programming with the Pi-XNA classes, an Effect object was declared, rather than the original BasicEffect object. In the LoadContent method, the declaration of vertex data remains the same as in the Windows XNA program. The array of indices was declared as a byte array because the program on Raspberry Pi eventually calls glDrawElements to draw primitives, and the sequence of drawing vertices of the primitives was changed to anti-clockwise to match the different default cull mode between XNA and OpenGL ES. After the vertices and indices have all been declared, an Effect object was created by calling the constructor of the Effect class. In the colored triangle program, the process of loading and compiling shaders were moved into the DrawUserPrimitive method, which is a method called in the game loop (i.e. the shader source code will be loaded and compiled on every iteration of the game loop). However, the loading and compilation of shaders should only be performed once during the initialization part of the program. Therefore, the constructor of the Effect class is a suitable method to place the process of loading and compiling shaders, because this method is called in LoadContent, which is executed once as part of the initialization. So when creating the Effect object, in addition to a GraphicsDevice object, the constructor also takes two strings that representing the names of the shader files as its parameters. This is a change from Windows XNA. This project does not attempt to reconstruct the XNA content management system, so the change cannot be avoided. Finally in

LoadContent, the value of VertexColorEnabled was set to true. This can be achieved by adding a property named VertexColorEnabled that returns a bool value. Figure.92 shows the modified LoadContent method.

```
protected override void LoadContent()
{
    vertices = new VertexPositionColor[8];

    vertices[0].Position = new Vector3(0.5f, 0.5f, 0.5f);
    vertices[1].Position = new Vector3(-0.5f, 0.5f, 0.5f);
    vertices[2].Position = new Vector3(-0.5f, -0.5f, 0.5f);
    vertices[3].Position = new Vector3(0.5f, -0.5f, 0.5f);
    vertices[4].Position = new Vector3(0.5f, -0.5f, -0.5f);
    vertices[5].Position = new Vector3(0.5f, 0.5f, -0.5f);
    vertices[6].Position = new Vector3(-0.5f, 0.5f, -0.5f);
    vertices[7].Position = new Vector3(-0.5f, -0.5f, -0.5f);

    vertices[0].Color = Color.Red;
    vertices[1].Color = Color.Green;
    vertices[2].Color = Color.Blue;
    vertices[3].Color = Color.Red;
    vertices[4].Color = Color.Green;
    vertices[5].Color = Color.Blue;
    vertices[6].Color = Color.Red;
    vertices[7].Color = Color.Green;


    indices = new byte[]{ 0,1,2,0,2,3,
                          0,3,4,0,4,5,
                          0,5,6,0,6,1,
                          1,6,7,1,7,2,
                          5,4,7,5,7,6,
                          2,4,3,2,7,4
                        };

    effect = new Effect(GraphicsDevice, "VertexShader", "PixelShader");
    effect.VertexColorEnabled = true;
}
```

Figure.92 Modified LoadContent for Drawing a Colored Rotating Cube

In the Draw method, as in the program in Windows XNA, the World, View and Projection matrices need to be defined. In the XNA program, these three matrices were managed by BasicEffect. So in the modified Draw method, the matrices should be handled by Effect. This can be done by adding three read and write Matrix type properties in the Effect class. This does not provide the full generality required to handle uniforms in shaders, but that can be added later. For the moment, uniforms are at least being managed in the correct place.

Finally, an Apply method can be created in Effect to bind the shader program for execution.

At the end of the modified Draw method, the DrawUserIndexedPrimitives method is called to draw primitives. Figure.93 shows the modified Draw method.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.PaleGreen);

    effect.World = Matrix.CreateRotationY((float)gameTime.TotalGameTime.TotalSeconds);
    effect.View = Matrix.CreateLookAt(new Vector3(0, 5, 5),
                                      new Vector3(0, 0, 0),
                                      new Vector3(0, 1, 0));
    effect.Projection = Matrix.CreatePerspectiveFieldOfView(3.1415926f/4, //MathHelper.PiOver4,
                            (float)graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight,
                            1.0f, 1000.0f);

    effect.Apply();
    GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionColor>(PrimitiveType.TriangleList,
                                                                  vertices, 0, 8,
                                                                  indices, 0, 12);


    base.Draw(gameTime);
}
```

Figure.93 Modified Draw Method for Drawing a Colored Rotating Cube

Modifying the GraphicsDevice Class:

As the Draw method in Game1 calls DrawUserIndexedPrimitives to draw primitives, a DrawUserIndexedPrimitives method was required in GraphicsDevice. This method was declared as

```
public void DrawUserIndexedPrimitives<T>(PrimitiveType primitiveType,
                                         T[] vertices, int vertexOffset, int numVertices,
                                         byte[] indices, int indexOffset, int primitiveCount)
```

Similar to the DrawUserPrimitives method, it firstly calculates the number of vertices according to the primitive type, then sets a pointer point to the vertex data, and then use vertex buffer objects to hold the data of the vertices, and finally calls glDrawElements, rather than glDrawArrays, to draw primitives.

Modifying the Effect Class:

The modified Effect class contains a constructor, four properties, and a method named Apply.

As described in the introduction to the modified Game1 class, the constructor of Effect contains the instructions for loading and compiling shader source code. In this program, the vertex shader and the pixel shader were programmed in two separate files. The parameters taken by the constructor are the names of the shader files, and it uses

```
System.IO.File.ReadAllText("./" + file name + ".txt");
```

command to read the shader source text from their paths.

The four properties are VertexColorEnabled, World, View and Projection. In the Apply method, the value of the World, View and Projection properties were read and used to calculate the transformation matrix. The data of the transformation matrix is then sent to the mvp uniform in vertex shader.

Modifying the Color Class:

As shown in Figure.94, more colors were used to define the vertex colors. The three colors added in the Color class are Red, Green and Blue. Figure.94 shows the definition of these three colors.

```
public static Color Red { get { return new Color(255, 0, 0, 255); } }
public static Color Green { get { return new Color(0, 255, 0, 255); } }
public static Color Blue { get { return new Color(0, 0, 255, 255); } }
```

Figure.94 Definition of Colors

After doing the modifications described above, a colored rotating cube can be displayed. Figure.95 shows the effect of running the modified program.

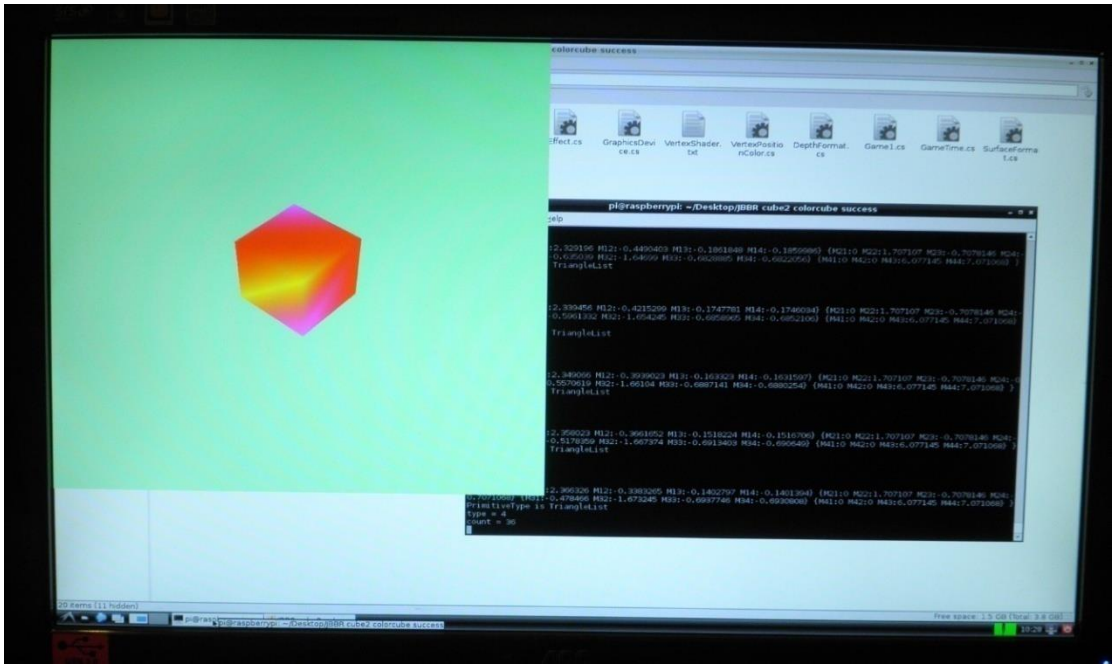


Figure.95 Colored Rotating Cube Displayed by Modified XNA Classes

4.4.4 Display a Textured Lighting Cube with Modified XNA Classes

The last job is to draw a textured rotating cube with modified classes on the Raspberry Pi. In order to show more shading effect, lighting was also implemented in the last program.

Creating a VertexPositionNormalTexture Structure:

As the texture effect and the lighting effect were to be implemented in this program, each vertex needed a texture coordinate and a normal vector to indicate the location in texture space and calculate the illumination intensity. Therefore, a new data structure, the VertexPositionNormalTexture, was created.

Compared with the VertexPositionColor structure, the vertex color is replaced by normal vectors and texture coordinates. The VertexPositionNormalTexture structure declares a Vector3 type variable representing vertex position, another Vector3 variable that represents vertex normal, and a Vector2 type variable stores the data of vertex texture coordinates. The constructor of this structure takes three parameters, and gives the values to the three variables respectively. Figure.96 shows the source code of this structure.

```
using System;

namespace JBBRXG11V2
{
    public struct VertexPositionNormalTexture
    {

        public Vector3 Position;

        public Vector3 Normal;

        public Vector2 TextureCoordinate;

        public VertexPositionNormalTexture(Vector3 position, Vector3 normal, Vector2 textureCoordinate)
        {
            Position = position;
            Normal = normal;
            TextureCoordinate = textureCoordinate;
        }
    }
}
```

Figure.96 VertexPositionNormalTexture

Figure.97 shows the data structure of VertexPositionNormalTexture.

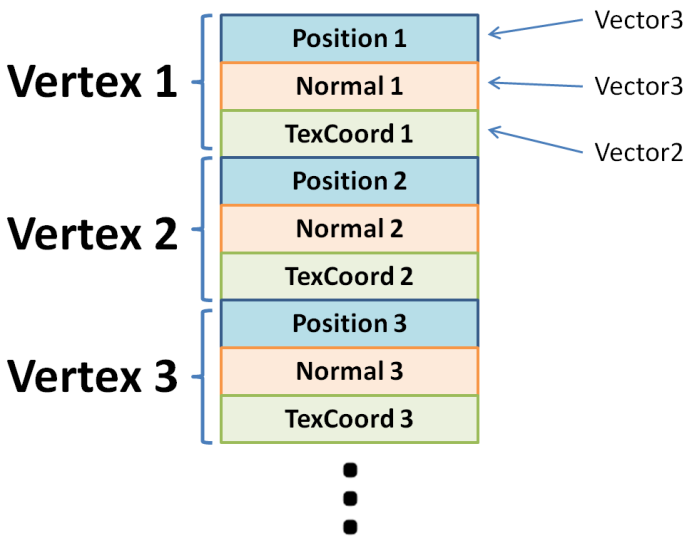


Figure.97 Data Structure of VertexPositionNormalTexture

Modifying the Game1 Class:

In the XNA library, a class named "Texture2D" handles the loading of texture data. When programming textured objects with XNA, a Texture2D object should be declared first, and in LoadContent method, `Content.Load<Texture2D>("file name")` command can be used to pass texture data to the Texture2D variable. Again a file name is needed because an XNA content system is not available.

Therefore, a Texture2D class should be created (the detail of Texture2D class will be explained later), and in the modified Game1 class, a Texture2D variable was declared. Also, the vertices of the cube were declared as VertexPositionNormalTexture type. Figure.98 shows the declaration part in modified Game1.

```
GraphicsDeviceManager graphics;

VertexPositionNormalTexture[] vertices;
byte[] indices;
Effect effect;

Texture2D tex;
```

Figure.98 Declarations in Game1 for Drawing a Textured Cube with Lighting Effect

In the LoadContent method, a total of 24 vertices data were defined. This is because each vertex on a cube is shared by three surfaces. As the normal directions of a vertex on the three connected surfaces are different, one vertex actually has three normal vectors, so 8 vertices have 24 normal vectors. Figure.99 shows the declaration of the vertices.

```
vertices = new VertexPositionNormalTexture[24];

vertices[00].Position = new Vector3(-0.5f,  0.5f,  0.5f);      vertices[00].Normal = new Vector3(0,  0, +1);
vertices[01].Position = new Vector3( 0.5f,  0.5f,  0.5f);      vertices[01].Normal = new Vector3(0,  0, +1);
vertices[02].Position = new Vector3(-0.5f, -0.5f,  0.5f);      vertices[02].Normal = new Vector3(0,  0, +1);
vertices[03].Position = new Vector3( 0.5f, -0.5f,  0.5f);      vertices[03].Normal = new Vector3(0,  0, +1);
vertices[04].Position = new Vector3( 0.5f,  0.5f,  0.5f);      vertices[04].Normal = new Vector3(+1, 0,  0);
vertices[05].Position = new Vector3( 0.5f,  0.5f, -0.5f);      vertices[05].Normal = new Vector3(+1, 0,  0);
vertices[06].Position = new Vector3( 0.5f, -0.5f,  0.5f);      vertices[06].Normal = new Vector3(+1, 0,  0);
vertices[07].Position = new Vector3( 0.5f, -0.5f, -0.5f);      vertices[07].Normal = new Vector3(+1, 0,  0);
vertices[08].Position = new Vector3( 0.5f,  0.5f, -0.5f);      vertices[08].Normal = new Vector3(0,  0, -1);
vertices[09].Position = new Vector3(-0.5f,  0.5f, -0.5f);      vertices[09].Normal = new Vector3(0,  0, -1);
vertices[10].Position = new Vector3( 0.5f, -0.5f, -0.5f);      vertices[10].Normal = new Vector3(0,  0, -1);
vertices[11].Position = new Vector3(-0.5f, -0.5f, -0.5f);      vertices[11].Normal = new Vector3(0,  0, -1);
vertices[12].Position = new Vector3(-0.5f,  0.5f, -0.5f);      vertices[12].Normal = new Vector3(-1, 0,  0);
vertices[13].Position = new Vector3(-0.5f,  0.5f,  0.5f);      vertices[13].Normal = new Vector3(-1, 0,  0);
vertices[14].Position = new Vector3(-0.5f, -0.5f, -0.5f);      vertices[14].Normal = new Vector3(-1, 0,  0);
vertices[15].Position = new Vector3(-0.5f, -0.5f,  0.5f);      vertices[15].Normal = new Vector3(-1, 0,  0);
vertices[16].Position = new Vector3(-0.5f,  0.5f, -0.5f);      vertices[16].Normal = new Vector3(0, +1,  0);
vertices[17].Position = new Vector3( 0.5f,  0.5f, -0.5f);      vertices[17].Normal = new Vector3(0, +1,  0);
vertices[18].Position = new Vector3(-0.5f,  0.5f,  0.5f);      vertices[18].Normal = new Vector3(0, +1,  0);
vertices[19].Position = new Vector3( 0.5f,  0.5f,  0.5f);      vertices[19].Normal = new Vector3(0, +1,  0);
vertices[20].Position = new Vector3(-0.5f, -0.5f,  0.5f);      vertices[20].Normal = new Vector3(0, -1,  0);
vertices[21].Position = new Vector3( 0.5f, -0.5f,  0.5f);      vertices[21].Normal = new Vector3(0, -1,  0);
vertices[22].Position = new Vector3(-0.5f, -0.5f, -0.5f);      vertices[22].Normal = new Vector3(0, -1,  0);
vertices[23].Position = new Vector3( 0.5f, -0.5f, -0.5f);      vertices[23].Normal = new Vector3(0, -1,  0);


const float xlo = 0f, xhi = 1f, ylo = 1f, yhi = 0f;

vertices[00].TextureCoordinate = new Vector2(xlo, ylo);
vertices[01].TextureCoordinate = new Vector2(xhi, ylo);
vertices[02].TextureCoordinate = new Vector2(xlo, yhi);
vertices[03].TextureCoordinate = new Vector2(xhi, yhi);          //                16------17
vertices[04].TextureCoordinate = new Vector2(xlo, ylo);          //               /       /
vertices[05].TextureCoordinate = new Vector2(xhi, ylo);          //              /       /
vertices[06].TextureCoordinate = new Vector2(xlo, yhi);          //             18------19
vertices[07].TextureCoordinate = new Vector2(xhi, yhi);          //                9--------8
vertices[08].TextureCoordinate = new Vector2(xlo, ylo);          //         12     |        |     5
vertices[09].TextureCoordinate = new Vector2(xhi, ylo);          //        /|      |        |    /|
vertices[10].TextureCoordinate = new Vector2(xlo, yhi);          //       / |      |        |   / |
vertices[11].TextureCoordinate = new Vector2(xhi, yhi);          //      13 |      11------10  4  |
vertices[12].TextureCoordinate = new Vector2(xlo, ylo);          //       | 14   0--------1     |  7
vertices[13].TextureCoordinate = new Vector2(xhi, ylo);          //       | /    |        |     | /      Y
vertices[14].TextureCoordinate = new Vector2(xlo, yhi);          //       |/     |        |     |/       |
vertices[15].TextureCoordinate = new Vector2(xhi, yhi);          //      15      |        |     6        |
vertices[16].TextureCoordinate = new Vector2(xlo, ylo);          //              2--------3              +---X
vertices[17].TextureCoordinate = new Vector2(xhi, ylo);          //                 22------23          /
vertices[18].TextureCoordinate = new Vector2(xlo, yhi);          //                /       /          Z
vertices[19].TextureCoordinate = new Vector2(xhi, yhi);          //               /       /
vertices[20].TextureCoordinate = new Vector2(xlo, ylo);          //              20------21
vertices[21].TextureCoordinate = new Vector2(xhi, ylo);
vertices[22].TextureCoordinate = new Vector2(xlo, yhi);
vertices[23].TextureCoordinate = new Vector2(xhi, yhi);
```

Figure.99 Vertex Declarations for Drawing a Textured Cube with Lighting Effect

Once the vertex data has been defined, the array of indices was given. The Effect object and the Texture2D object were instantiated to load shaders and the texture (as the XNA ContentManager class was not included in this project, the constructor of modified Texture2D handles the loading of textures). Figure.100 shows the remainder of LoadContent.

```
indices = new byte[]{ 0,2,1,1,2,3,
                      4,6,5,5,6,7,
                      8,10,9,9,10,11,
                      12,14,13,13,14,15,
                      16,18,17,17,18,19,
                      20,22,21,21,22,23
                     };

effect = new Effect(GraphicsDevice, "VertexShader", "PixelShader");

tex = new Texture2D(GraphicsDevice, "./TulipSquare.bmp");
```

Figure.100 Declaring Indices and Loading Contents

In a typical Windows XNA project, transformation matrices are passed to and used in the vertex shader. The World, View and Projection matrices are all sent to the vertex shader, and it may multiply these three matrices to get a composite transformation matrix. By contrast, as explained in section 4.4.3, the current modified program calculates a combined transformation matrix in the Apply method of Effect class. However, the Effect class is transparent to the users, and this causes some problems. Programmers have to use the calculated transformation matrix in vertex shader, even if they do not want to do so (in shaders, if an attribute or uniform is given a value from the program, then it has to be used for calculating the positions or colors of the primitives, otherwise nothing can be displayed on the window. Probably the lack of output is due to silent failure in some part of the shaders driver code).

In order to solve this problem, a new overload method named "SetParameters" was added in the Effect class (the detail of this method will be explained later). This method takes two parameters, the first is a string variable, which is the name of the attribute or uniform in vertex shader. The other parameter takes a variable holding the data that suppose to be sent to the corresponding attribute or uniform. In this way, users are able to decide which data should be sent to the shaders according to the needs of their shader programs. For example, as shown in Figure.101, the transformation matrix "mvp" was calculated in the Draw method in Game1, then the SetParameters("mvpMatrix", mvp); command could be used to send the matrix data to a uniform named "mvpMatrix" in the vertex shader. This provides the same flexibility as the Windows implementation of the XNA Effect class. In the Windows version, parameters are set up as an array property of Effect, so the syntax is slightly different. However, usage is similar.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.PaleGreen);

    effect.World = Matrix.CreateRotationY((float)gameTime.TotalGameTime.TotalSeconds);
    effect.View = Matrix.CreateLookAt(new Vector3(0, 5, 5),
                                      new Vector3(0, 0, 0),
                                      new Vector3(0, 1, 0));
    effect.Projection = Matrix.CreatePerspectiveFieldOfView(3.1415926f/4, //MathHelper.PiOver4,
                                (float)graphics.PreferredBackBufferWidth / (float)graphics.PreferredBackBufferHeight,
                                    1.0f, 1000.0f);

    Matrix mvp = effect.World * effect.View * effect.Projection;


    effect.Apply();
    effect.SetParameters("mvpMatrix", mvp);
    effect.SetParameters("worldMatrix", effect.World);
    effect.SetParameters("textureUnit0", tex);

    GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionNormalTexture>(PrimitiveType.TriangleList, vertices,
                                                                          0, 24, indices, 0, 12);


    base.Draw(gameTime);
}
```

Figure.101 Draw Method for Drawing a Textured Cube with Lighting Effect

As the lighting effect was implemented in this program, the data of the World matrix had also to be sent to the vertex shader to allow the shader to transform normal vector into World space to get the correct lighting effect.

Creating a Texture2D Class:

The modified Texture2D class contains a constructor and the LoadTexture method used in the previous C# program. The constructor takes a GraphicsDevice object and the file path of the texture as its parameters, and calls the LoadTexture method to load texture data.

Modifying the Effect Class:

Firstly, the overload method named "SetParameters" as mentioned above, was created to send uniform data to the uniforms in shaders. The glGetUniformLocation function was moved from the constructor of Effect to SetParameters.

Secondly, the routine for calculating the transformation matrix was removed from the Apply method as this should either be done in the Game1 class by the user or be done in the shader itself. The current Apply method only handles the use of program and sets some GL states.

Figure.102 shows the modified code in Effect class.

```
public void Apply()
{
    Console.WriteLine("Apply");

        glUseProgram(programObject);// (ctx.rs.po);

        // set state
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
}


public void SetParameters(string attribName, Matrix matrix)
{
    matrixLoc = glGetUniformLocation(programObject, attribName);

    glUniformMatrix4fv(matrixLoc, 1, GL_FALSE, ref matrix);
}

public void SetParameters(string attribName, Texture2D tex)
{
    texUnitLoc = glGetUniformLocation(programObject, attribName);

    SetTexture(tex);
}

public void SetTexture(Texture2D texture)
{
    uint textureUnit = texture.tex[0];

        glUniform1i(texUnitLoc, 0);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textureUnit);
}
```

Figure.102 Modified Methods in Effect Class for Drawing a Textured Cube with Lighting

Modifying the GraphicsDevice Class:

The only change in GraphicsDevice class was the vertex buffer objects used in DrawUserIndexedPrimitives method. As in Game1, a new data structure was used to declare the vertex data, when calling DrawUserIndexedPrimitives in Draw method, the generic type of vertices actually used VertexPositionNormalTexture structure. Therefore, instead of holding vertex color data, the vertex buffer object allocates buffers for vertex positions, vertex normals and texture coordinates.

The type of vertex position and that of vertex normal are both Vector3, which consists of three floats. The type of texture coordinate is Vector2, which contains only two floats. Therefore, the number of vertex position elements and that of vertex normal elements are all 3 (3 floats), the number of texture coordinate elements is only 2. The extra data between each vertex data address is (3 + 3 + 2) * sizeof(float), and their offset of the first element are 0, 3 * sizeof(float), and 6 * sizeof(float) respectively. Figure.103 shows the modified VBOs in DrawUserIndexedPrimitives.

```
// Load the vertex data

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(attribute_v_position, 3, GL_FLOAT, GL_FALSE, 8  * sizeof(float), 0);
glEnableVertexAttribArray((uint)attribute_v_position);

glVertexAttribPointer(attribute_v_normal, 3, GL_FLOAT, GL_FALSE, 8  * sizeof(float), 3 * sizeof(float));
glEnableVertexAttribArray((uint)attribute_v_normal);

glVertexAttribPointer(texcoordLoc, 2, GL_FLOAT, GL_FALSE, 8  * sizeof(float), 6 * sizeof(float));
glEnableVertexAttribArray((uint)texcoordLoc);
```

Figure.103 VBOs for VertexPositionNormalTexture

Modifying the Shaders:

To implement the lighting effect, the shader source code was modified as well. In vertex shader, the light direction was defined, and a float varying "lightlevel" was declared to represent the illumination intensity. This is calculated by multiplying the light direction with the vertex normal, and clamping the result into 0 to 1.

The light level was sent to the fragment shader to be multiplied with the colors, and the lighting effect was successfully achieved.

Figure.104 and Figure.105 show the not textured cube with lighting effect and the textured cube with lighting effect respectively.

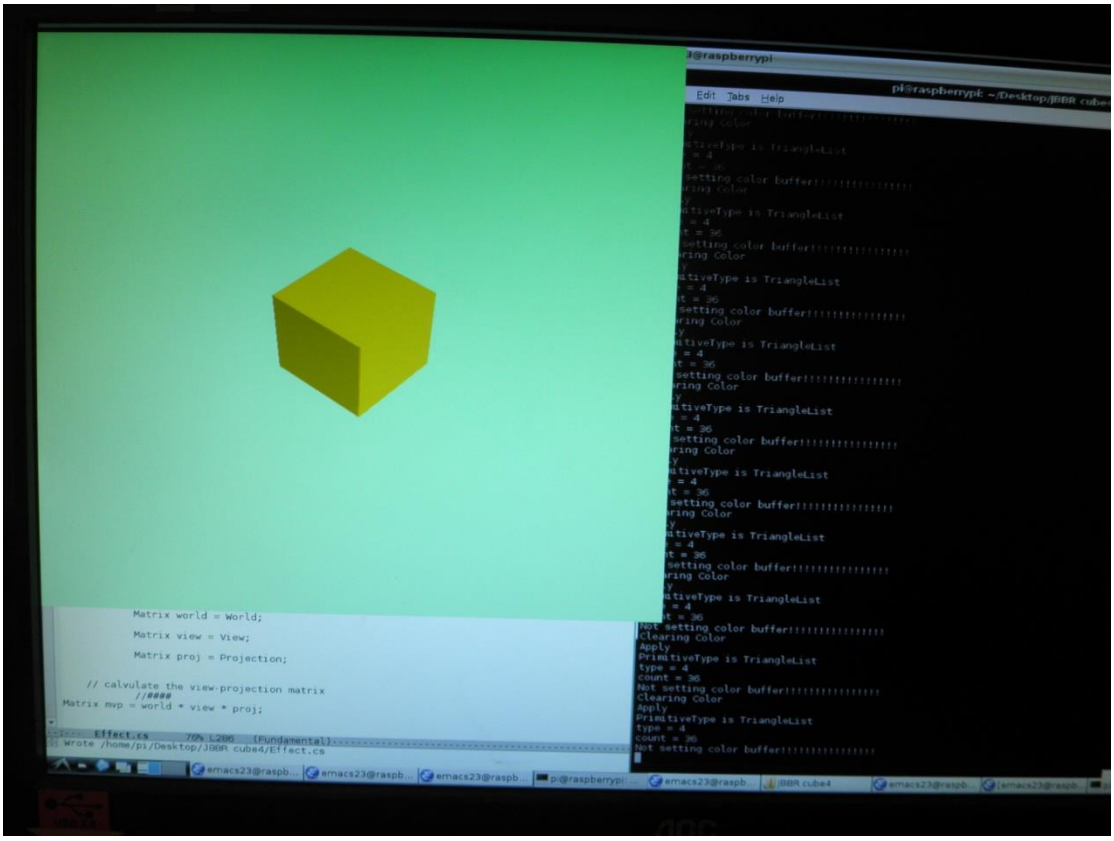


Figure.104 Cube with Lighting Effect

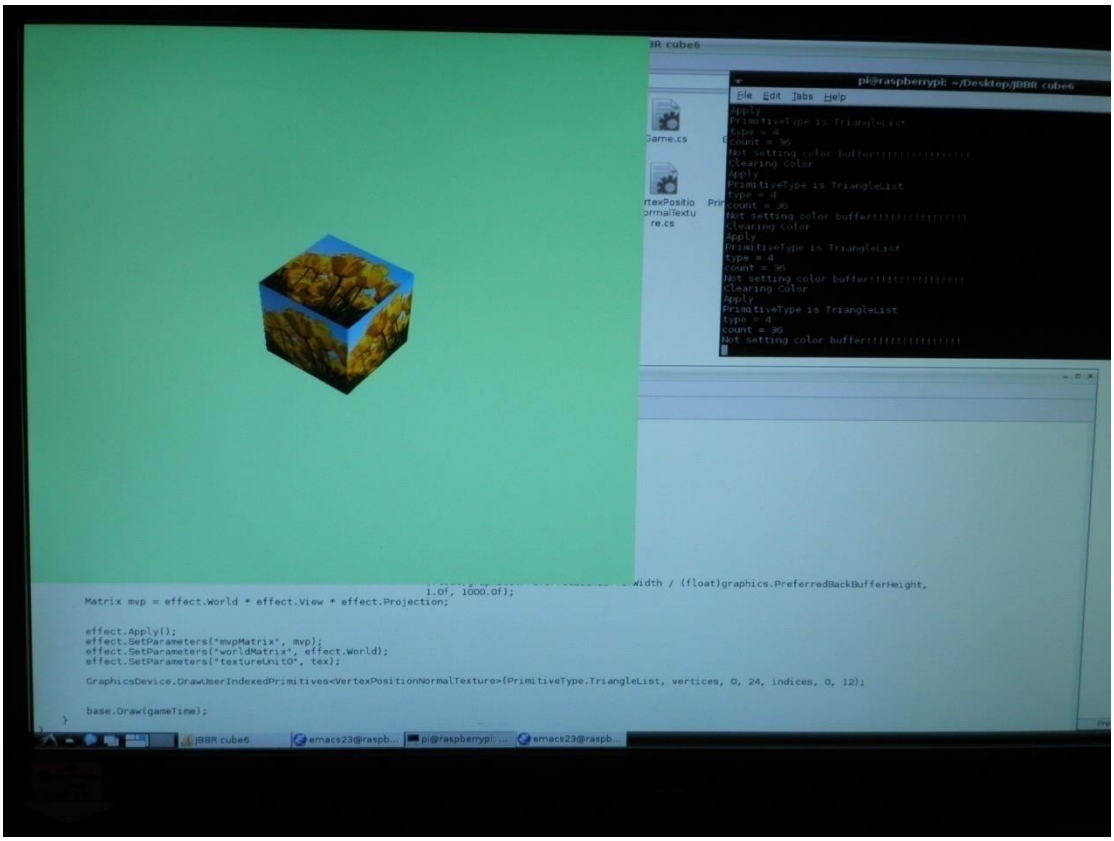


Figure.105 Textured Cube with Lighting Effect

At this stage, there are enough modified classes for programming a textured3D object with lighting effect on Raspberry Pi, and the code in Game1 is quite similar to the XNA program on Windows. However, there is still a part in the program which cheats.

When modifying the GraphicsDevice class to draw the textured cube with lighting effect, we knew that the generic type used in DrawUserIndexedPrimitives for declaring the vertices would be VertexPositionNormalTexture. So the way of using VBOs in DrawUserIndexedPrimitives was manually modified and hard coded to allocate buffers for VertexPositionNormalTexture type vertex data.

However, vertex data may be declared with other data structures, like the sample of drawing a colored rotating cube with the modified classes. It also calls the DrawUserIndexedPrimitives method to draw primitives, but its vertex data was of type VertexPositionColor.

In the Windows XNA library, there is a class named "VertexDeclaration", which contains an array of VertexElements. VertexElement is a structure that stores the information for all the possible vertex data items (e.g. offset, format, index etc.). When calling DrawUserIndexedPrimitives in the Draw method, information about the vertex data type can be queried from VertexDeclaration. For example, if the vertex data is declared as VertexPositionColor, one vertex data consists of 3 floats representing the XYZ value of the position, followed by 4 byte values representing the color. By querying the VertexDeclaration, the program knows that a vertex position occupies 12 byte of the buffer, and its offset is 0 byte. Similarly, the buffer size for a vertex color should be 4 bytes, with a 12 byte offset for the first vertex color data. When a Windows XNA program is running, this information will be automatically queried and used to allocate buffers for the vertex data.

A VertexElement structure can be created in this project to store information on vertex data types. A new class named VertexDeclaration can also be created, holding a VertexElement array to store the information about different vertex data types in use. When calling the drawing methods declared in GraphicsDevice, the type of the vertex

data could be looked up, and buffers automatically allocated for the vertices. Because of the time limitation, using VBOs to automatically allocate buffers for the vertex data is not achieved in this project, and this work is left for the future.

Another difficulty also arises with vertex data structures. In HLSL, the package of information constituting a vertex is usually referenced as a structure, and each data item has an associated "semantic" which identifies its role in the shader. For example, POSITION and NORMAL are HLSL semantics. This makes it possible to map vertex items to shader attributes automatically. An OpenGL ES shader lacks this detail. To overcome the problem, we adopted the convention of using standard names for vertex shader attributes in our final shader programs. This should permit the final step of automating vertex buffer management.

Chapter 5: Conclusion and Future Work

This thesis has shown the possibility of developing XNA like programs directly on the Raspberry Pi. It described the way of interacting with the graphics system through the OpenGL ES 2.0 library by creating some sample programs, both in C++ language and in C# language. It also explained the data structures and algorithms of some of the XNA classes, the relationships between these classes, and how to combine the routines of the C# sample programs with the XNA classes to acquire the modified classes that replace DirectX with the OpenGL ES library.

At the end of this project, users are able to program in quite a similar way to programming XNA on Windows to draw a textured rotating cube with lighting effects. Shaders and the texture are loaded from separate files. This result shows that the modified system is able to create a suitable rendering environment for drawing 2D and 3D objects with different data structures. It is also capable of implementing shading effects (e.g. texturing and lighting) and mathematical calculations (e.g rotation) to the objects.

Therefore, this project clearly shows that it is absolutely possible to write XNA like programs on Raspberry Pi.

Pi-XNA programs are not identical to Windows XNA programs. However the differences are small and appropriate to Raspberry Pi development. Pi-XNA programs use GL shader language. There are differences in the way assets are loaded. But, the level of detail required of implementers and the level of abstraction of the liabrary matches that of XNA.

The goal of this project was not to fully rewriting the XNA library. XNA contains a large number of classes and structures. Rewriting all parts of the XNA library to make it work with OpenGL ES 2.0 involves a huge amount of work. Because of time limitations, this project only modified the most important and needful classes and structures, which declared enough fuctionalities of the system and proved the possibility of XNA programming on the Raspberry Pi.

More work can be done in the future to make the system become a fully featured library on Raspberry Pi. However, building an entire XNA like system still leaves much work to do. There are two parts can be firstly done in the near future.

Users have to write their own shaders to implement shading effects with the current modified classes. Therefore, one work that worth to do is to create a BasicEffect class that contains built-in shader programs, which allows programmers to use a number of different effects in the Game1 class without writing any shader programs themselves.

The other work can be done in the future is to keep modifying the GraphicsDevice class. Compare with the modified classes, the GraphicsDevice in XNA contains more properties and drawing methods, like the DrawInstancedPrimitives method, and some overload declarations of the methods that already existed in current modified GraphicsDevice class. Fully modifying this class may provide users more flexibility for their programs. In particular implementing VertexDeclaration and using it to automatically bind different vertex buffer types would be very useful.

Appendix

Setting-up the Raspberry Pi

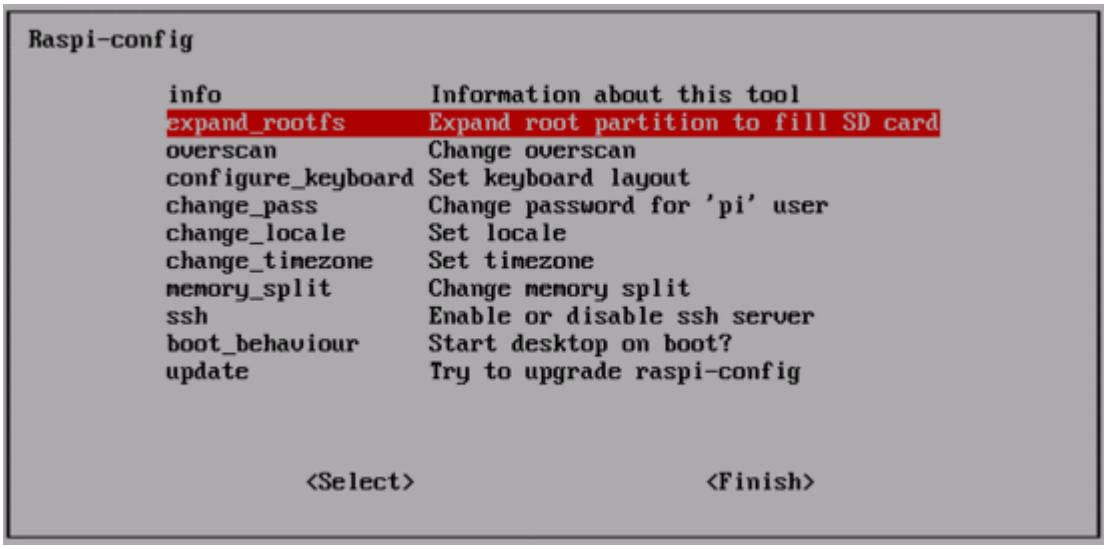
Raspberry Pi uses an SD card as its ram, so the first thing to do is to write the disk image of Raspbian on to an SD card (4GB or larger). Raspbian images can be downloaded from http://www.raspberrypi.org/downloads

For this project the Hard Float version of Raspbian (release dated 26/7/2013) was used. This version uses the hardware floating point capabilities of the Raspberry Pi CPU (instead of using floating point emulation software).

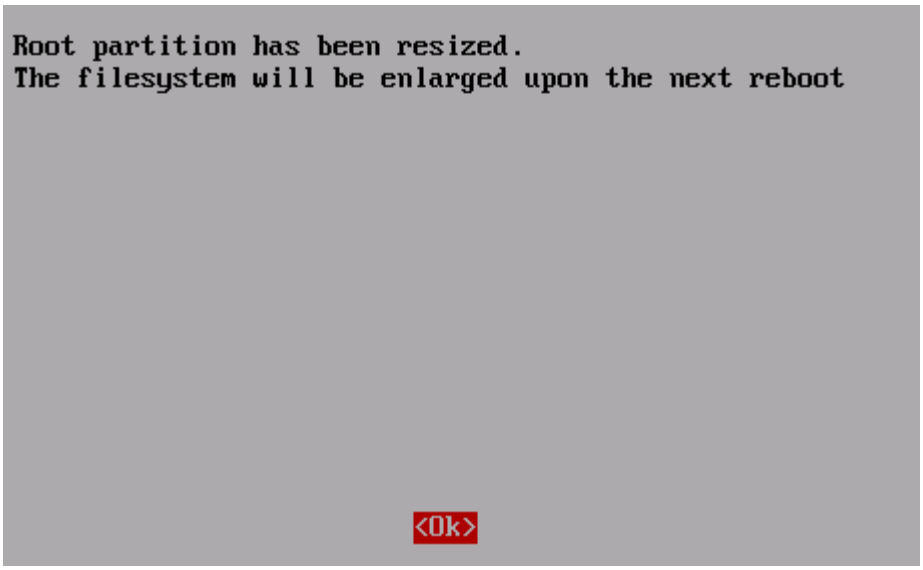
Please note that not all kinds of SD cards can be used with Raspberry Pi. The available and unavailable SD cards can be checked on
http://elinux.org/RPi_SD_cards

Raspbian images can be written into SD cards from a Windows computer by using software named "Win32diskimager". Instructions for its use can be found on
http://rpi.tnet.com/project/faqs/backups/backingup

The first time a Raspberry Pi boots up with Raspbian, the Raspi-config will be automatically shown to help setting up the configuration of Pi. Later users may type "raspi-config" ata terminal window to use this configuration tool again.



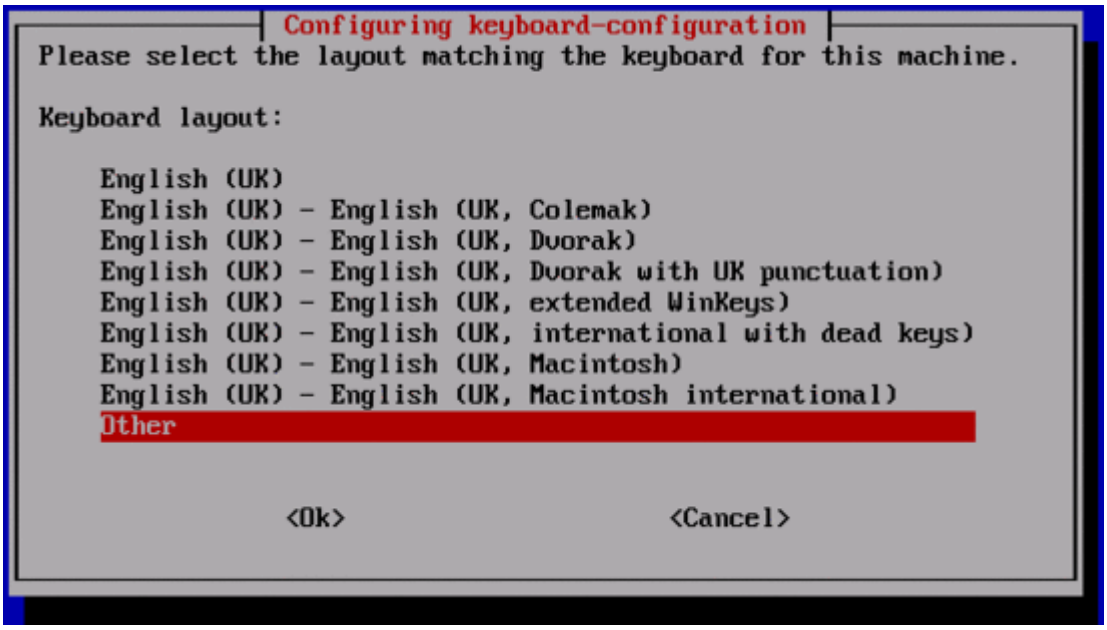
Firstly, the expand_rootfs should be enabled to fully use the SD card's memory space on Raspberry Pi. The expand_rootfs is disabled by default, and there will be only approximately 2GB available, no matter how big is the memory space of the SD card.

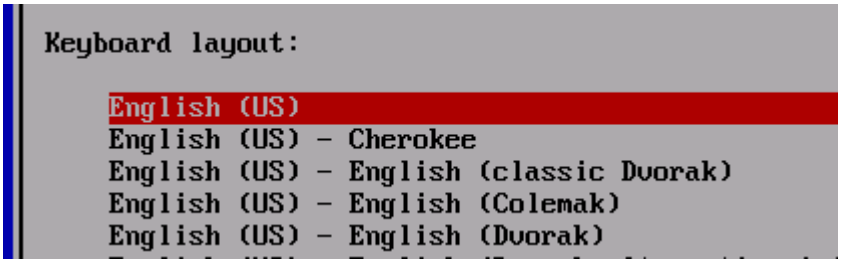


The screens used with Raspberry Pi may have different resolutions. Disabling the overscan option allows the screen to be fully used to display images on the Pi.

The next step is to select a layout for the thekeyboard. In the configure_keyboard option, users may firstly choose the type of the keyboard used, and then a list of keyboard layouts will be offered.

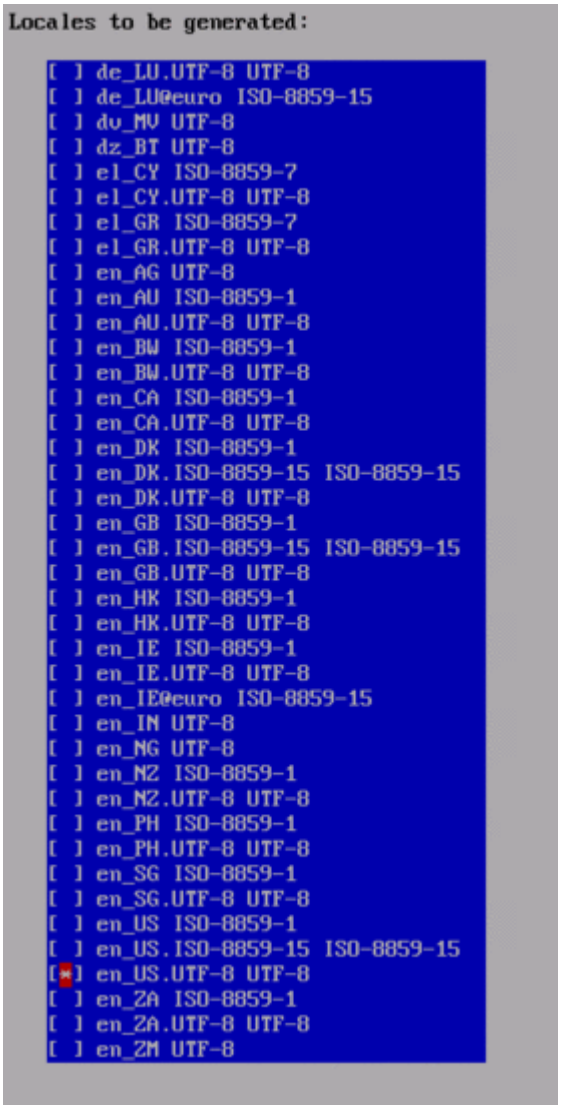
In this project, the English(US) was used as the keyboard layout.



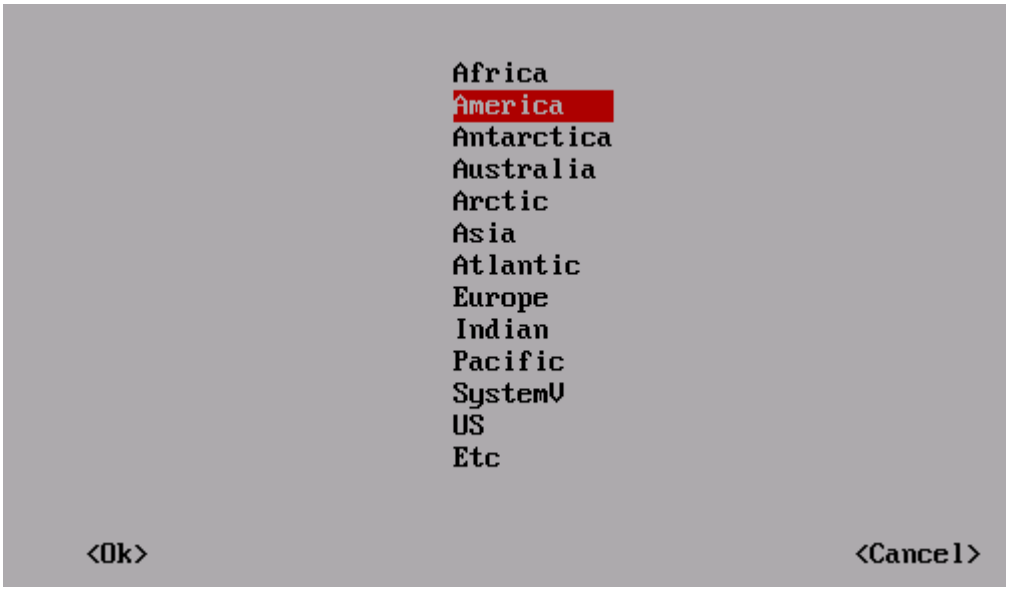


The change_pass option in the main menu handles the modifying of password.

The next step is to set up the region. This will decide the symbols and languages used on Raspberry Pi.



The last step is to set the time zone by the chage_timezone option.



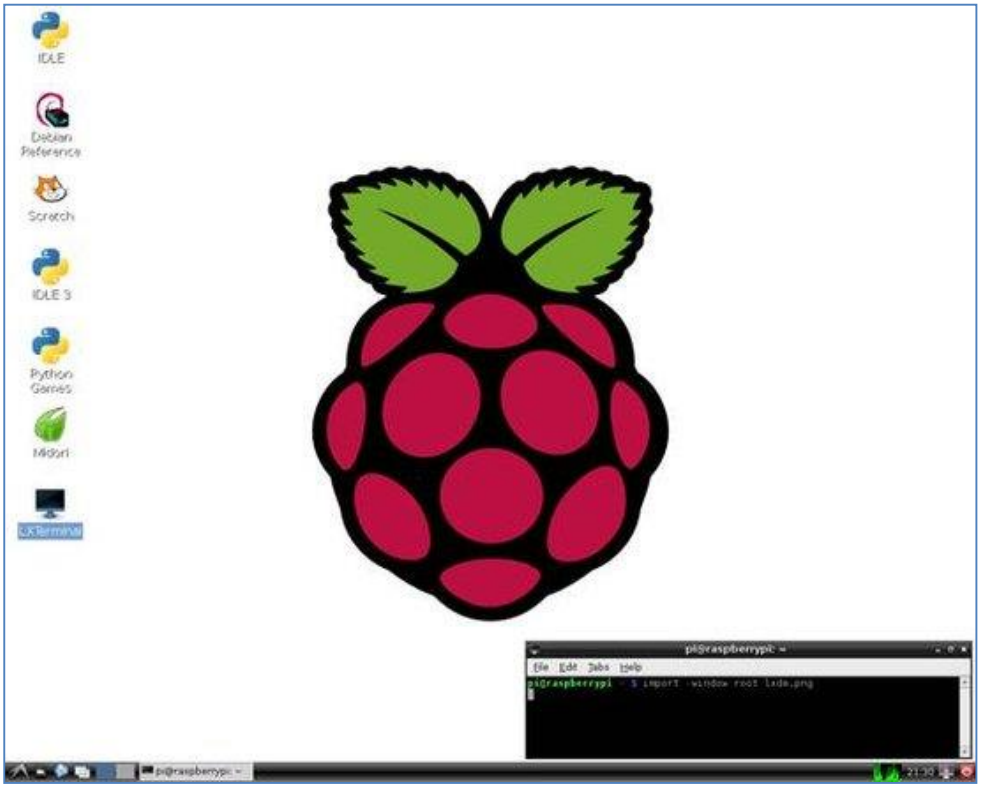
At this stage, the set up of Raspberry Pi is complete. Reboot the Pi, and this time users may login with the modified password.

```
My IP address is 192.168.11.22

Debian GNU/Linux wheezy/sid raspberrypi tty1

raspberrypi login: _
```

After logging in, users may type the "startx" command to launch a graphical session, and the desktop will be displayed.



Installing Mono

To run the code in this project it is necessary to install Mono – the open source implementation of C#. At the time of writing there was an issue with Mono on the Raspberry Pi – it had errors in handling hardware floating point. Instead of using the standard version of Mono therefore, code from an experimental branch of the Mono project in which the floating point handling had been repaired was used.

```
cd ~
wget https://www.dropbox.com/s/sask17flot3zqlg/mono_2_11_4_armv6hf_binary.tgz
cd /
sudo tar zxvf ~/mono_2_11_4_armv6hf_binary.tgz
sudoldconfig
sudo apt-get install libgdiplus
```

List of EGL and OpenGL ES Functions Used in This Project

EGL Procedures:

GLenum**glGetError**(void)

EGLDisplay**eglGetDisplay**(EGLNativeDisplayTypedisplay_id)

EGLBoolean**eglInitialize**(EGLDisplay display, EGLint *majorVersion,
EGLint *minorVersion)

EGLBoolean**eglTerminate**(EGLDisplay display)

EGLBoolean**eglGetConfigs**(EGLDisplay display, EGLConfig *configs,
EGLintmaxReturnConfigs,
EGLint *numConfigs)

EGLBoolean**eglChooseChofig**(EGLDispay display,
constEGLint *attribList,
EGLConfig *config,
EGLintmaxReturnConfigs,
ELGint *numConfigs)

EGLBoolean**eglGetConfigAttrib**(EGLDisplay display, EGLConfigconfig,
EGLint attribute, EGLint *value)

EGLSurface**eglCreateWindowSurface**(EGLDisplay display,
EGLConfigconfig,
EGLNatvieWindowType window,
constEGLint *attribList)

EGLBoolean**eglDestroySurface**(EGLDisplaydisplay,EGLSurface surface)

EGLContext**eglCreateContext**(EGLDisplay display, EGLConfigconfig,
EGLContextshareContext,
constEGLint* attribList)

EGLBoolean**eglDestroyContext**(EGLDisplaydisplay,EGLContext context)

EGLBoolean**eglmakeCurrent**(EGLDisplay display, EGLSurface draw,
EGLSurface read, EGLContext context)

EGLBoolean**eglSwapBuffers**(EGLDisplaydisplay,EGLSurface surface)

OpenGL ES 2.0 Procedures:

```
voidglClear(GLbitfield mask)
voidglClearColor(GLclampf red, GLclampf green,
GLclampf blue, GLclampf alpha)
voidglViewport(GLint x, GLint y, GLsizei w, GLsizei h)
GLuintglCreateShader(GLenum type)
voidglShaderSource(GLuintshader, GLsizei count,
const char** string,
constGLint* length)
voidglCompileShader(GLuintshader)
voidglGetShaderiv(GLuintshader, GLenumpname,
GLint *params)
voidglGetShaderInfoLog(GLuintshader, GLsizeimaxLength,
GLsizei *length, GLchar *infoLog)
voidglDeleteShader(GLuintshader)
GLuintglCreateProgram(void)
voidglAttachShader(GLuint program, GLuintshader)
voidglLinkProgram(GLuint program)
voidglGetProgramiv(GLuint program, GLenumpname,
GLint *params)
voidglDeleteProgram(GLuint program)
GLintglGetAttribLocation(GLuint program,
constGLchar *name)
voidglUseProgram(GLuint program)
voidglVertexAttribPointer(GLuint index, GLint size,
GLenum type,GLboolean normalized,
GLsizei stride,const void *ptr)
voidglEnableVertexAttribArray(GLuint index)
voidglDrawArrays(GLenum mode, GLint first, GLsizei count)
voidglGenBuffers(GLsizei n, GLuint *buffers)
voidglBindBuffer(GLenum target, GLuint buffer)
voidglBufferData(GLenum target, GLsizeiptr size,
const void *data, GLenum usage)
GLintglGetUniformLocation(GLuint program,const char* name)
voidglUniformMatrix4fv(GLint location, GLsizei count,
```

```
GLboolean transpose,
constGLfloat* value)
voidglEnable(GLenum cap)
voidglDisable(GLenum cap)
voidglDepthFunc(GLenumfunc)
voidglDisableVertexAttribArray(GLuint index)
voidglTexParameteri(GLenum target, GLenumpname,GLintparam)
voidglTexImage2D(GLenum target, GLint level,
GLenuminternalFormat, GLsizei width,
GLsizei height, GLint border,
GLenum format, GLenum type,
const void* pixels)
voidglUniform1i(GLint location, GLint x)
voidglActiveTexture(GLenum texture)
voidglBindTexture(GLenum target, GLuint texture)
voidglGenTextures(GLsizei n, GLuint *textures)
```

References

[1]. http://readwrite.com/2014/01/20/
raspberry-pi-everything-you-need-to-know#awesm=~otPxAqvcxEKveu

[2]. http://www.hackthings.com/raspberry-pi-model-a-and-b/

[3]. http://en.wikipedia.org/wiki/Atmel

[4]. http://en.wikipedia.org/wiki/Microsoft_XNA

[5]. http://www.mono-project.com

[6]. http://en.wikipedia.org/wiki/MonoGame

[7]. http://jbbrxg11.codeplex.com/

[8]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.game.initialize.aspx

[9]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.game.update.aspx

[10]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.game.isfixedtimestep.aspx

[11]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.game.targetelapsedtime.aspx

[12]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.game.draw.aspx

[13]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.graphics.aspx

[14]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.graphicsdevicemanager.aspx

[15]. http://msdn.microsoft.com/en-us/library/
microsoft.xna.framework.graphics.graphicsdevice.aspx

[16]. http://rbwhitaker.wikidot.com/intro-to-shaders

[17]. Addison, Wesley (2008) OpenGL ES 2.0 Programming Guide

[18]. http://en.wikipedia.org/wiki/EGL_(API)

[19]. James Boud (2012) Extending SlimDXna to Use XNA 4 and DirectX 11

[20]. http://jbbrxg11.codeplex.com/

[21]. http://www.opengl.org/documentation/specs/version1.1/glspec1.1/
node93.html#SECTION00710000000000000000

[22]. http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/jb/jsr239/javax/microedition/khronos/egl/EGL10.html

[23]. http://www.opengl.org/wiki/Face_Culling

[24]. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.game.run.aspx