Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 0202-0001

Development and Analysis of a Window Manager Concept for Consolidated 3D Rendering on an Embedded Platform

Han Zhao

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Simon Gansel, Dipl.-Inf. Stephan Schnitzer
Commenced:	19. Jan. 2015
Completed:	21. July 2015
CR-Classification:	I.3.2, C.3, D.4.9

Abstract

Nowadays with the information technology rapidly developing, an increasing number of 2D and 3D graphics are used in automotive displaying systems, to provide vehicle information, driving assistance, etc. With the demand of 3D models interacting with each other, an implementation should have a 3D compositing capability. However, traditional 2D compositing implementations are not capable of 3D models compositing tasks. In order to composite 3D graphics on embedded platform, the 3D compositing implementation is necessary.

Therefore, a concept of window manager is developed aiming to composite 3D graphics with an optimized efficiency for embedded platform. Specially for automotive platforms, a virtualization is made to unify multiple Electronic Control Units (ECUs) into one single ECU platform. On this platform, a server and multiple clients are implemented with dedicated Virtual Machines (VMs). The server is in charge of rendering tasks requested from clients.

Based on this, a 3D compositing concept is implemented. It handles efficiently the multiple 3D applications situation using a method of off-screen rendering. A server-side virtualization is also implemented by replacing certain client-side commands during commands forwarding. With this virtualization implementation, multiple applications run simultaneously with accessing single 3D GPU only. Moreover, due to this implementation, monolithic rendering operations affecting all applications, e.g. uniform lighting operation, are possible.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Recently, with the rapid expansion of electronics and information technology, 2D and 3D computer graphics are becoming widely applied in daily life, e.g. smartphones, animation movies and embedded devices. In the last generation of automotive Instrument Cluster (IC) system, analog gauges are widely used for speedometer, tachometer, etc. Also for the Head Unit (HU) system, integrated electronic navigator, reversing camera video and other kinds of in-car infotainment are displayed.

Currently, there is a tendency to mount more than one displays per car. For each display, a large size screen with high resolution is equipped, to provide rich and vivid information and a good driving experience for drivers. With the development of *Internet of Things*[7] and autonomous car technology, the concept of *Internet of Vehicles*(IoV)[8] comes out. By using inter-vehicle communication and real-time road condition collection, this assists drivers to avoid severe accidents. For the IoV model, multiple displays with accurate and intuitive information and feedback providing are also the essential part.

In latest in-vehicle information and infotainment systems, multiple displaying screens become the main information providers in the car. For example, in the new Mercedes-Benz concept car *F 015 Luxury in Motion*[9], as Figure 1.1 shows, multiple displays are mounted to interact with both the driver and passengers.

However, multiple in-vehicle displays are always connected with different Electronic Control Units (ECUs). With the number of displays increasing, more ECUs have to be mounted in a vehicle accordingly. Moreover, separated ECUs are difficult to communicate and interact with each other. For example, the IC system and HU system are separated with different displays and ECUs respectively, and it is difficult to do a monolithic rendering operation or interaction to applications from different systems, such as a uniform lighting or shadow mapping.

To make the monolithic rendering operations available across multiple displays, a consolidated window manager concept is proposed. In this concept, a virtualization is implemented to unify the IC and HU systems. For an isolation purpose, each system runs on a dedicated Virtual Machine (VM), as a client VM. Both IC system VM and HU system VM forward rendering commands to another VM, which is as a server VM. This server VM operates all forwarded commands from client-side and decide the rendering algorithms. This implementation makes monolithic rendering operations and inter-VM interactions possible, with a unification 3G GPU access by multiple systems.

Figure 1.1: Mercedes-Benz Concept Car F 015 Luxury in Motion [1]

A traditional 2D compositing concept has an obvious disadvantage when it comes to a 3D models intersecting rendering situation. Therefore, a 3D compositing concept is proposed implementing on the basis of the consolidated window manager concept, which makes multiple 3D applications compositing fully supported.

Summing up the above, an implementation on server VM is introduced. This implementation replaces the forwarded commands from client-side applications, and then redirect the render target of each application to a dedicated off-screen buffers. After off-screen renderings complete, the server VM use these rendered off-screen buffers as sources of a second rendering process. During the second rendering, all client-side applications are manipulated by server VM with various operations, such as 3D compositing, uniform lighting, shadowing, etc. With this implementation, further monolithic operations are also possible to add on this platform.

Outline

This thesis is organized as follows:

Chapter 2 – Background: introduces the background knowledge which is referred to among this work, such as graphics libraries and the window manager concept.

Chapter 3 – System Model: gives an overview of the whole system model on which this implementation works, including assumptions about this work.

Chapter 4 – Concepts and Implementations: describes the concept and implementation of this work, including the system architecture, the algorithms, data structures and implementations.

Chapter 5 – Evaluation: presents the evaluation scenarios, results and analysis about this implementation.

Chapter 6 – Related Work: does a brief review of related implementations.

Chapter 7 – Conclusion and Outlook: draws a conclusion and presents further possible work to improve this implementation.

2 Background

In this chapter, related background knowledge is introduced.

The whole work is based on the concept of window manager, which is widely integrated in modern operating systems. Therefore the window manager concept is discussed along with examples from well-known operating systems.

This work is implemented with the graphical library OpenGL ES, which is a multi-platform graphics application programming interface (API)specially for embedded systems. OpenGL ES is in charge of the main drawing tasks in this work.

In order to provide a rendering environment for OpenGL ES on an embedded platform, the library EGL is used. EGL is an intermediate layer between OpenGL ES and the native windowing platform. Namely, EGL prepares the rendering environment for OpenGL ES. First EGL communicates with the native windowing system of an embedded platform to fetch the current connected display and acquire a window for the rendering preparation. And with these EGL builds a rendering container specially for OpenGL ES. Then the OpenGL ES rendering process takes place in EGL containers.

2.1 Window Manager

Window manager is a system software with the functionality of controlling where and how could a window be drawn in a graphical user interface (GUI)[10]. Window managers manage multiple graphical applications displaying on the screen with corresponding order, position, shape, etc. In modern operating systems, a window manager has a significant role especially for desktop environments.

Modern operating systems, such as Windows and Mac OS X, usually have their dedicated window managers. Below are examples to explain the functionality of window managers.

Quartz

Quartz is a window manager on Mac OS X. Figure 2.1 shows the desktop of Quartz window manager, and two applications display simultaneously. One of them is on top of the other, which is decided and drawn by Quartz. Window managers decide and manipulate the way to draw graphics on screen.

Figure 2.1: Quartz Desktop

Windows Aero

From Windows Vista, a new window manager for Windows is introduced with the name of *Aero*. Aero supports plenty of 3D effects and animations, as Figure 2.2. A 3D flip effect is made by Aero for a multi-tasking switch. Aero accomplishes this effect by manipulating display surfaces of each applications and rebuild them with a 3D compositing algorithm. This is how window managers composite 3D graphics.

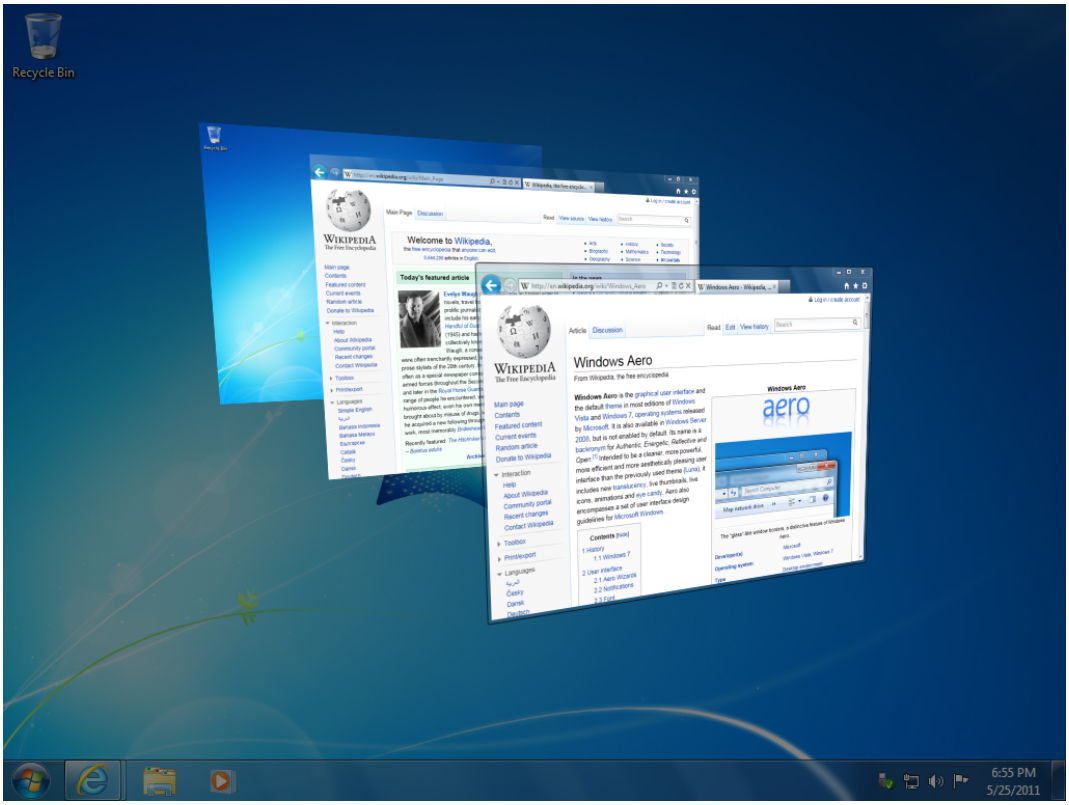


Figure 2.2: Flip 3D on Windows 7 [2]

2.2 OpenGL ES 2.0

OpenGL ES(also called GLES as abbreviation) is a full-function 2D and 3D graphics API specialized for embedded systems. It is a subset of OpenGL, the corresponding API of desktop version. It is cross-platform supported, widely applied on modern embedded operating systems such as Linux, iOS, Android, etc.[11]

It is designed for embedded systems, therefore it aims to maximize the efficiency and usage of hardware, with a relative low power consumption. For this reason, some features are removed from OpenGL to OpenGL ES.

2.2.1 Programable Pipeline

OpenGL ES 1.x(including 1.0, 1.1 and all version ealier than 2.0) is defined for a fixed function rendering pipeline. Programs can benefit from hardware acceleration by call provided functions only. In OpenGL ES 2.0, a programmable rendering pipeline is introduced (Figure 2.3), which is a huge progress making it easier and more flexible to design and implement.

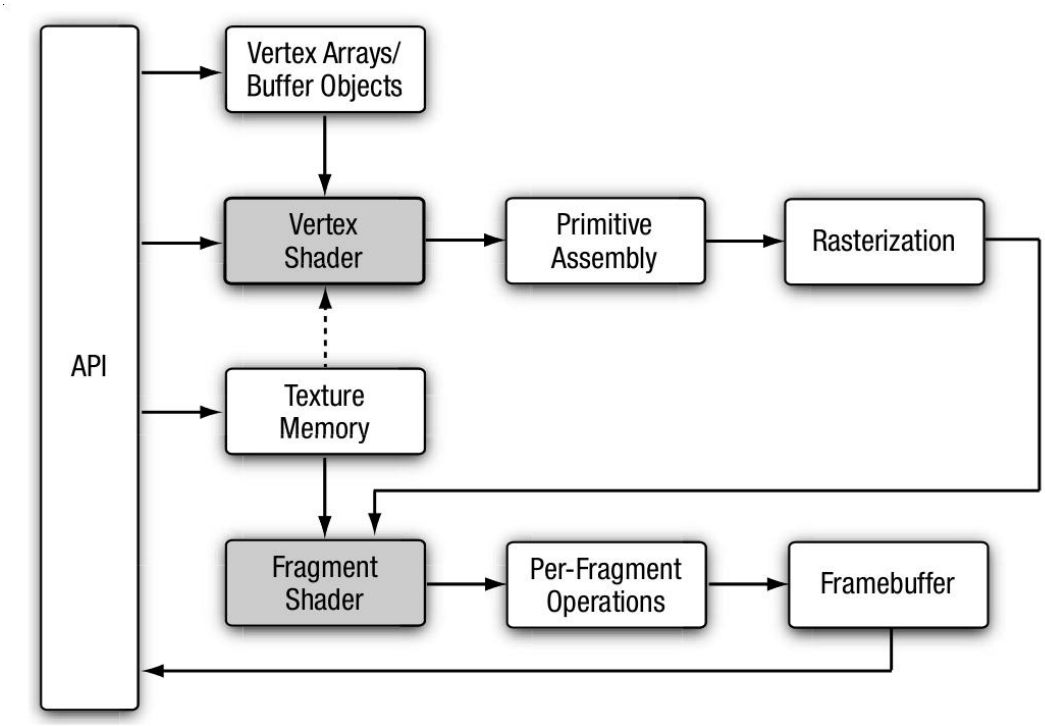


Figure 2.3: OpenGL ES 2.0 Programmable Rendering Pipeline [3]

In the programmable rendering pipeline, *shaders* are introduced. A shader is a program written with OpenGL Shading Language (GLSL), a C-like language with customized types, such as

vectors, matrices, and related operations. These features make GLSL to be convenient of manipulating vertices, coordinate transformations, etc.

There are two types of shaders, vertex shaders and fragment shaders. A vertex shader is to manipulate the vertices with certain operations, such as coordinate transformations, per vertex operations, etc. A fragment shader receives the result of rasterization and operates each fragment. In fragment shader, an operation frequently used is to sample and display a texture. Besides, the fragment shader can manipulate the display content result with its own algorithm.

Shaders provide a highly flexible way to customize OpenGL ES pipeline, to meet the requirements.

2.2.2 Textures

A texture is an OpenGL object that contains one or more images with the same image format. There are 2D and 3D textures. However, in most cases, 2D textures are preferred because of the simplicity and performance. Textures can be used in two ways, as an input source of rendering, or as an output render target of a rendering process.

Applying textures as a rendering source to a surface is one of the most fundamental operations in rendering 3D graphics, which reduces a considerable amount of drawing overhead[4]. For instance, to draw a brick wall (as Figure 2.4) using OpenGL ES commands only, all vertices from each brick, various colors, and even the random spots should be specified and calculated during rendering process. However, if a texture rendering is used, only the vertices coordinates of texture are specified, with sampling texture and mapping to certain position.

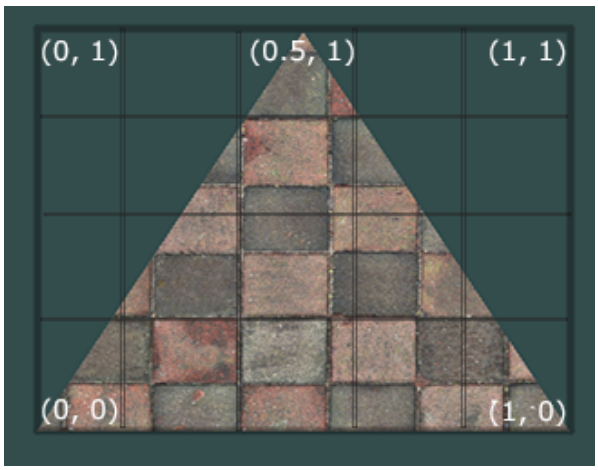


Figure 2.4: Render a Brick Wall with Texture [4]

Textures can also be used as a color or depth buffer for data storage. In this case, a texture becomes to a render target, instead of a render source. Rendering contents, such as depth information, color information and stencil information, can be stored into one or multiple textures respectively. And these textures can be used later by another rendering process as render sources.

2.3 EGL

EGL is an interface between OpenGL ES API and the underlying native windowing systems[12], providing OpenGL ES a platform-independent rendering container.

A series of EGL commands should be called in certain order to build an OpenGL ES rendering environment. The code snippet 2.1 depicts a basic process to establish an rendering environment.

```
1         EGLDisplay display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
2         eglInitialize(display, NULL, NULL);
3         eglChooseConfig(display, attribute_list, &config, 1, &num_config);
4         Window_Type native_window = createNativeWindow();
5         EGLWindowSurface surface = eglCreateWindowSurface(display, config, native_window, NULL);
6         EGLContext context = eglCreateContext(display, config, EGL_NO_CONTEXT, NULL);
7         eglMakeCurrent(display, surface, surface, context);
```

List of Listings 2.1: EGL Basic Workflow

Following are separated steps of this process according to List 2.1.

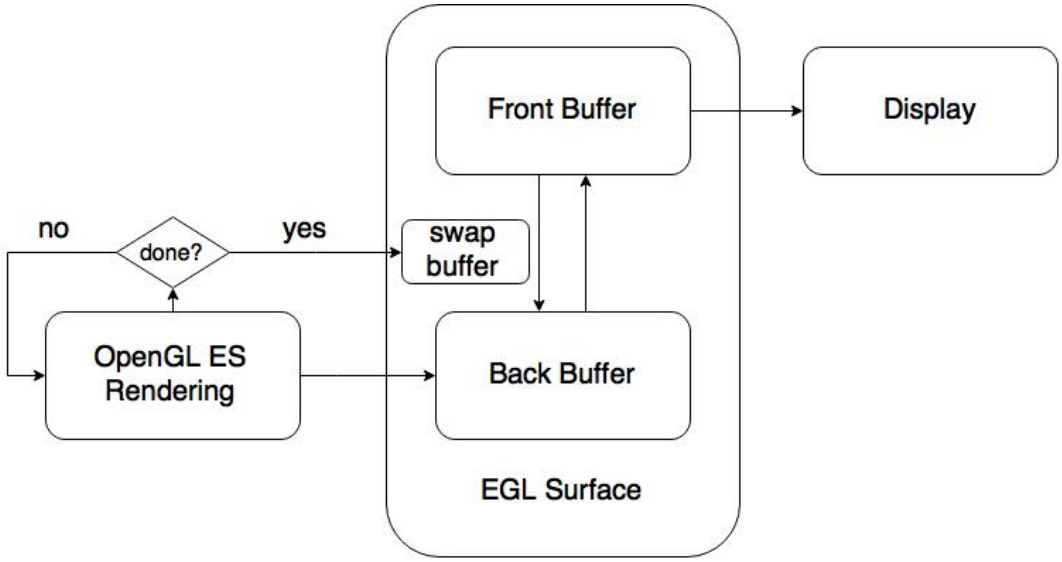
eglGetDisplay

Most EGL functions require a display directly or indirectly, because a display is the final render target where all drawing contents are shown to users. **eglGetDisplay** does a job to bind `EGLDisplay` with a native display. A display could be a framebuffer or other display units. Usually this is the first step of all EGL calls, and the returned `EGLDisplay` is used in the following EGL calls.

After this, the function **eglInitialize** should be called to initialize the display.

eglChooseConfig

This function aims to acquire a customized framebuffer configuration. This includes the framebuffer format, the sizes of color buffer, depth buffer, etc. The returned configuration is used as parameter to create a `EGL Surface`.



H

Figure 2.5: EGL Double Buffering System

eglCreateWindowSurface

This function creates a `EGLWindowSurface`. A `EGLWindowSurface` is a chunk of memory where rendering takes place. A window surface is a surface that can be shown directly on a screen. Besides this, there are also other kinds of EGL surfaces e.g. pixel buffer surface, which is a surface that cannot be shown on a display. In the following chapters we will discuss how and why we use pixel buffer surface in our implementation.

To create a window surface, a window is required. As line 4 of List 2.1, a native window is provided by a native window manager.

There is a double buffering mechanism in EGL surfaces. For each EGL surface creation, there are always two buffers created at the same time, the front buffer and the back buffer. The back buffer behaves as the render target of client-side renderings, while the front buffer is the memory chunk used directly by the on-screen framebuffer. After a whole rendering process is finished, **eglSwapBuffers** is called, and the back buffer is swapped to front, and vise versa.(see Figure 2.5) With this system, a complete rendering image is guaranteed for each frame to present to users, and partial updates of certain frames are avoided.

eglCreateContext

eglCreateContext creates a `EGLContext`.

`EGLContext` is a container of OpenGL ES or other client API rendering. A context should be established specifying with a client API and corresponding version, and in our case it is OpenGL ES 2.0. The bound surface, display should also be specified during `EGLContext` creation.

eglMakeCurrent

There could be more than one `EGLContext` for each surface, but rendering commands only draw on the one which is made to be *current*. Therefore before rendering starts, **eglMakeCurrent** is called with specified `EGLContext` and `EGL Surface`.

So far, an OpenGL ES rendering environment is established, and OpenGL ES commands start executing. Every time when a frame is rendered completely, **eglSwapBuffers** would be called to swap the rendered content from back buffer to on-screen buffer. Then next frame starts rendering, and swapping again to front buffer when it is completed. This is the OpenGL ES rendering loop.

3 System Model

This chapter presents a general system view on which this implementation works. Assumptions about this work will be proposed regarding to the system.

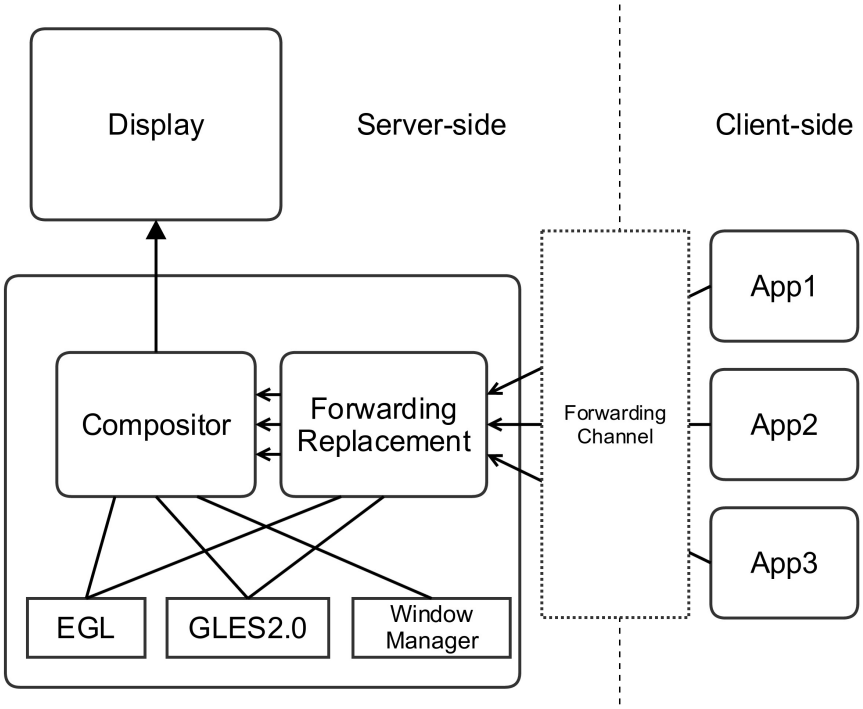


Figure 3.1: System Model

Figure 3.1 illustrates the system and its working flow. Rendering takes place on this system in two stages, client-side requests and server-side responses. On client-side, applications firstly send EGL/GLES commands to server. On server-side, commands will be forwarded to compositor.

On client-side, EGL and OpenGL ES commands are called in turn to accomplish a rendering process. Also, a shared memory channel between server-side and client-side has been established before rendering for communication.

System Virtualization

The server-client model is running on the same host machine with separated virtual machines (VMs) respectively. Figure 3.2 illustrates the hierarchy of a hypervisor-based virtualization model.

A hypervisor is a system software in charge of creating and running VMs. The hypervisor manages all VMs to share and access the hardware of host machine, such as 3D GPU, displays, etc. The forwarding replacement part between client-side applications and server-side is implemented via the shared memory channel.

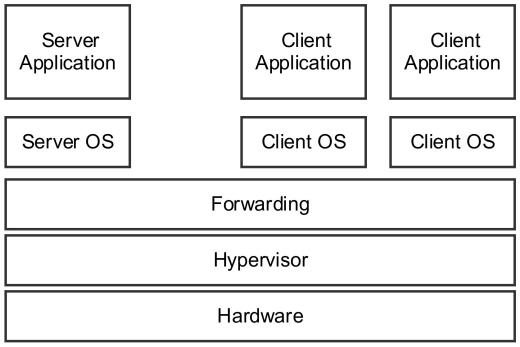


Figure 3.2: System Virtualization

Server-side and client-side applications are running on their dedicated operating systems (OSes) separately on top of the hypervisor. Multiple client VMs are deployed for dedicated systems, for instance, the IC system and the HU system. Different systems can be isolated with separated client VMs implementation, whereas the rendering processes of all client VMs are gathered at the server VM.

With this implementation, a server VM and multiple client VMs are running separately for an isolation purpose. Moreover, by gathering all rendering works to the server VM, monolithic rendering operations are possible on server-side to manipulate all client-side renderings with a unifying operation. Also, an efficient usage of hardware resources, e.g. 3G GPU, is another advantage of this implementation.

Forwarding Replacement

In the forwarding replacement process, commands from client-side applications are assumed to be handled in different methods. For certain EGL commands, appropriate replacement

schemes are applied, in order to build a rendering container on an off-screen buffer for each application to execute the following OpenGL ES commands. All rendering related commands, are forwarded directly without changes, to ensure the integrity of rendering content. This builds a transparency layer to client-side, which is virtualized and makes applications on client-side executing just as working naively.

In this part all client-side renderings take place in off-screen buffers, which store the rendered contents of each client-side application, as the render sources of 3D compositing implementation.

Compositor

On compositor part, rendering contents passed from the forwarding replacement part are composited according to the position of each window surface and, if multiple applications exist at the same z position, depth information. Namely, multiple applications are assumed to be rendered at the same window according their depth information, achieving a 3D compositing implementation. At the end the composited graphics are displayed on monitor.

Window Manager

The Window manager is implemented to provide a native window for rendering processes. According to the Section 2.3, a native window is required during the EGL working process. Each client-side application requests a window during initialization, and these requests are handled by the window manager.

4 Concepts and Implementations

In this chapter, design concepts and implementations are described.

This work uses off-screen rendering for client-side applications. A 3D compositing via fragment shader is implemented with the rendered off-screen buffers as render sources. A virtualization concept is implemented via forwarding replacement, making multiple client-side applications running on server with accessing single 3G GPU.

Section 4.1 presents the basic structure and the rendering process. And in Section 4.2 the concept off-screen rendering is introduced for accomplishing this work. About this concept, different schemes are compared and selected. Section 4.3 describes the selected off-screen rendering scheme in detail about how all things work. Section 4.4 presents the forwarding replacement methodology, to visualize the server-side for client-side applications. In Section 4.5 the main rendering thread is introduced for the server-side system. Section 4.6 presents a uniform lighting implementation.

4.1 Architecture

Figure 4.1 is the architecture of the 3D compositing concept.

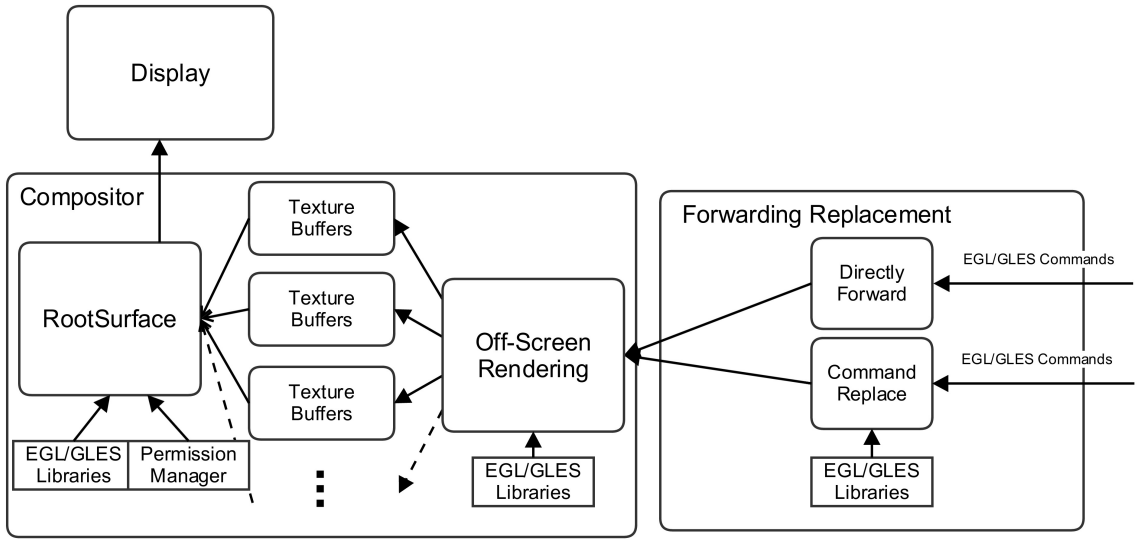


Figure 4.1: Architecture

Figure 4.1 illustrates the basic concept of two key parts in this work, the compositor part and the forwarding replacement part.

On the forwarding replacement part, the inputs are EGL and GLES commands forwarded from client-sides. Forwarding replacement algorithms vary, however, from different commands. For most commands, a simple forwarding operation is implemented, with exactly the same command passing through, being as a transparent channel. For some commands, instead of forwarding the original data from client-sides, this part replaces it with new EGL or OpenGL ES commands, according to the rendering algorithms on compositor part. In Section **4.4 Forwarding Replacement**, the details about commands replacement with reasons are described.

On the compositor part, in order to render multiple 3D applications compositing according to depth data at the same window, a concept of off-screen rendering is raised to manage multiple graphics rendering. Namely, the application from client-side renders first not to the on-screen framebuffer, but to an off-screen framebuffer. And until all applications finish rendering, data from all these off-screen buffers are gathered and composited, finally rendering to the on-screen framebuffer with considering all depth information from each application. Section **4.2 Off-Screen Rendering** and **4.3 Framebuffer Objects** present the details about this algorithm, including rendering schemes comparison, final implementation, etc. Section

4.5 Root Surface Rendering describes the concept and implementation for rendering from the off-screen buffers to the on-screen framebuffer.

Monolithic rendering operations to all applications are also a key feature of this implementation due to the consolidated window manager concept. Various operations can be implemented to client-side applications, such as uniform lighting, shadowing, etc. In Section **4.6 Uniform Operations** a uniform lighting is introduced along with related algorithms.

4.2 Off-Screen Rendering

Speaking of rendering processes, a rendering target has to be set, to which the finally full-rendered graphic is stored and displayed. It could be a on-screen framebuffer, which is the most common situation, and after rendering process completed, the screen used will display the graphic content of this on-screen framebuffer immediately. This should be the final step of all rendering processes, if these rendering processes are aimed to display some content on a screen.

There is another rendering method, instead of on-screen framebuffer, it renders to off-screen "framebuffer" or other kinds of buffers. There are various reasons for doing this. In our case, this is used for temporarily storing rendered content of each application. These stored contents from all client-side applications will be used during the *RootSurface* rendering, for multiple 3D graphics rendering. And then the render target is set to screen, completing a on-screen rendering process.

Several possible off-screen rendering schemes are presented here for comparison and selection with demands of this project. Because of the limitation of embedded system, a scheme with less resource consumption and more efficiency is important and necessary.

Pixel Buffer

In EGL library there is a kind of surface named pixel buffer surface (pbuffer for short, mentioned in 2.3), which is an off-screen surface. Unlike on-screen surfaces, it allows and only allows client-side rendering to a off-screen buffer. Thus there is no associated native framebuffer along with it[13].

To create a pbuffer surface, an EGL command **eglCreatePbufferSurface** is called. The creation procedure is just similar with that of an EGL window surface. After creation, client-side applications can render directly to this surface as a normal render target. Pbuffer can take full advantage of any hardware acceleration available to GLES 2.0, just as a window surface[3] does.

Framebuffer Object

A framebuffer object (FBO), as its name expresses, is an object which behaves just like a framebuffer. OpenGL ES 2.0 treats a FBO just the same as a native framebuffer. When a FBO is established, a GLES function **glBindFramebuffer** can be called passing the name of this created FBO as a parameter, to bind this as a current rendering framebuffer. By default, a

native framebuffer of current screen is bound. The default name of native framebuffer is set to '0'.

Comparison and Selection

- FBOs are more efficient than pbuffers[3]. Thus in terms of this one, in the situation where both can be applied and used, the solution with FBOs is more preferable.
- Sharing of depth buffers between FBOs or with other `eglContext` is possible with framebuffer objects usage, while this is usually not true with pbuffers[3]. Accessing the depth data of each FBO is very important in our case because we need to use this information for multiple 3D applications compositing.
- There is a double buffering system in pbuffer surface, but not in FBOs. Namely, using pbuffers can benefit from this. However, a solution to setup this double buffering system manually by building a double FBOs for each client-side application, which makes similar result as the double buffering system in EGL surface does.
- pbuffer surfaces are EGL surfaces while framebuffer objects are OpenGL ES objects. Besides the efficiency mentioned in the first point, another point is that, during the forwarding replacement, a GLES data structure is likely easier to handle as well as to share with each other than a EGL data structure. This can also make the replacement logic less complex.

To sum up, an off-screen rendering scheme with framebuffer objects are more suitable to our requirements than the EGL pbuffer surfaces scheme, in consideration of efficiency, implementation difficulty and logic complexity.

4.3 Framebuffer Objects

A concept of FBO off-screen rendering solution is described in this section.

According to this concept, a *Render to Texture* method is implemented. And a FBO double buffering system is implemented to simulate and implement the *swap buffer* function.

4.3.1 FBO Workflow

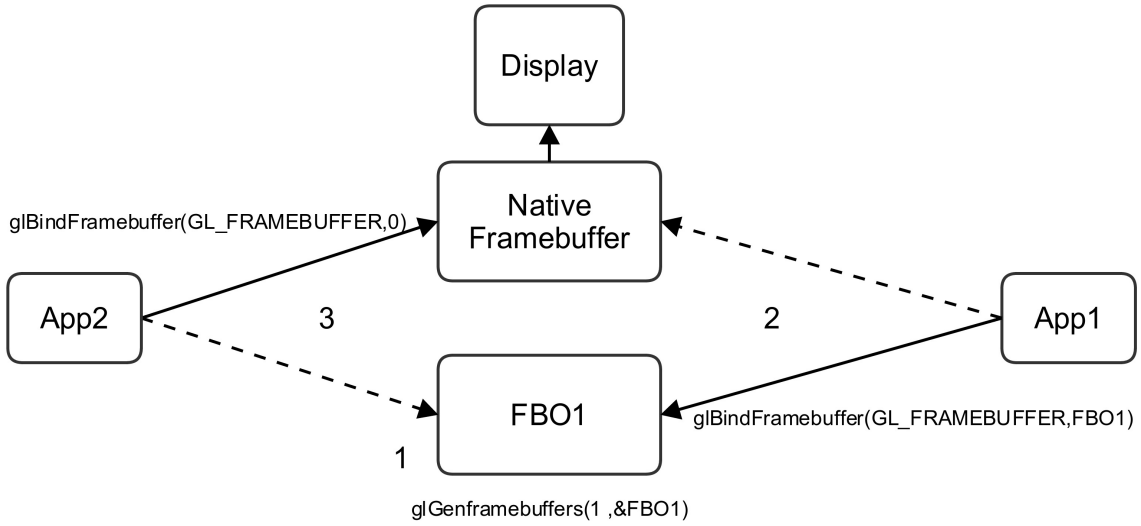


Figure 4.2: Framebuffer Object Workflow

The workflow (Figure 4.2) has three steps:

First Step 1 generates a framebuffer object by calling the function **glGenFramebuffers** with a specific name `FBO1`.

Second At step 2, a client-side application named `App1` is supposed to render to `FBO1`. However, by default the render target of OpenGL ES commands is the native framebuffer. Hence at this time a function **glBindFramebuffer** is called with parameter `FBO1` to rebind the render target from native framebuffer to this framebuffer object, namely `FBO1`. In the following rendering commands from GLES, the render target is `FBO1`, which means, at this moment, all renderings affect noting with the screen display any more, but only the framebuffer object, `FBO1`. All rendering and displaying contents are stored inside `FBO1`, ready for later use.

Third After `App1` all renderings complete, another application `App2` is planned to render directly on this native framebuffer. Thus again **glBindFramebuffer** is called with parameter `0`,

which means the default (native) framebuffer. After this function call, `App2` can start to render on the native framebuffer to expect a rendering result displaying on screen.

This workflow depicts that with this FBO off-screen rendering solution, an application is able to choose its render target freely, which could be an on-screen framebuffer, or a FBO. This feature makes this work possible to manipulate multiple applications by using a FBO off-screen rendering.

4.3.2 Render to Texture

In the last section the basic FBOs implementation structure is described. In this section the deep inside of a FBO is explored and discussed how to store and load the color and depth information.

FBO Attachments

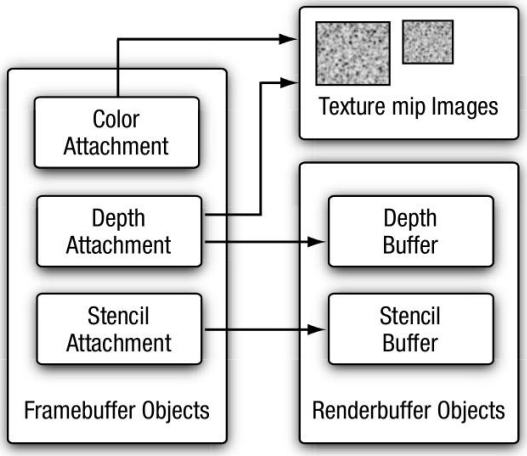


Figure 4.3: Attachments for Framebuffer Objects [3]

Figure 4.3 illustrates the available attachments of a FBO and types. Before describing this figure, some new concepts should be introduced first.

A *renderbuffer object*, unlike a framebuffer object, is a 2D image buffer allocated for storing color data, depth data or stencil data of a framebuffer object[3].

Renderbuffer Objects versus Textures

According to Figure 4.3, there are three kinds of attachments for FBOs, one color attachment, one depth attachment and one stencil attachment.

- In general using renderbuffer objects as attachments of a FBO has efficiency advantages over a texture, and renderbuffer has a wider support to be an attachment of a FBO than a texture[3].
- The disadvantage of using renderbuffer objects as color or depth attachments is that, during the root surface rendering process, renderbuffer objects cannot be used and sampled in a fragment shader, which makes the following rendering steps difficult to proceed.

According to the archtecture, the color and depth information stored inside FBO attachments are fundamental inputs for the root surface rendering. Hence a *Render to Texture* solution is chosen in our case, for both color buffer and depth buffer implementations.

Textures Rendering Workflow

A textures based FBO attachments for our work is chosen with reasons mentioned above, and Figure 4.4 shows the workflow of textures rendering from client-side applications and then as inputs to a root surface rendering process.

First Step 1 shows two applications `App1` `App2` send OpenGL ES rendering commands to their corresponding FBO respectively. Each FBO has one color buffer and one depth buffer, both of which are implemented with textures.

Second At step 2, each FBO has received all rendering commands from its client-side application and rendered completely. The rendering color and depth results are stored into these attached buffers respectively.

Third On root surface side, a new thread is allocated and running from the beginning. It detects every new color and depth buffers and composites all these buffers together to the native framebuffer. And at this time the final rendering image displays on screen.

Render to Texture is one of the key steps in this work. The details about root surface rendering will be discussed in Section 4.5.

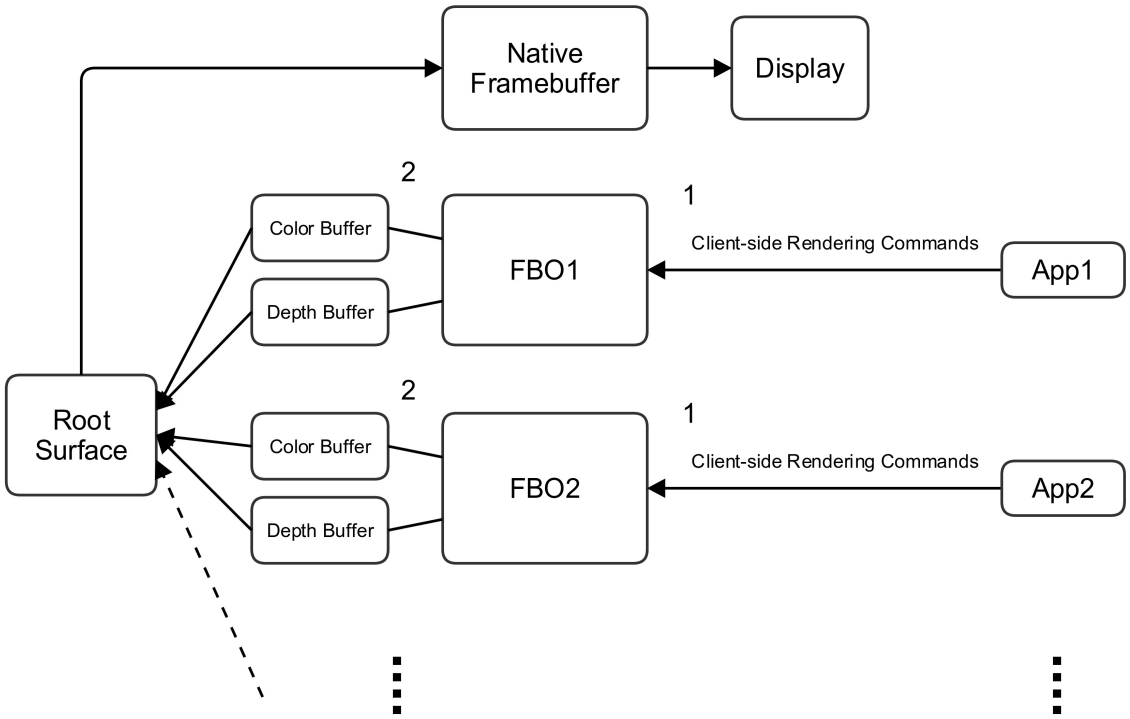


Figure 4.4: Textures Rendering Workflow

4.3.3 Double Buffering in FBOs

As we claimed on Section 4.2, a disadvantage of using FBO is lack of double buffering system in comparison of EGL pbuffer surface, which leads to a unfinished rendering image displaying on screen. To avoid this situation, we imitate this system and build a double FBOs system for each client-side application.

Figure 4.5 illustrates the double buffering FBOs implementation, which behaves similar as EGL double buffering systems does.

First In step 1 client-side application `APP1` renders on `FB01_back` as described in Figure 4.4. Instead of one FBO per application, here two FBOs (front and back respectively) are created for each application.

Second After all rendering completed, a function **texSwapTexture()** is called for this FBO. At this time, front FBO swaps with back FBO, along with all attachments together. This makes the rendered content swapped from the back FBO to the front one.

Third On root surface side, color buffer and depth buffer of the front FBO are used as inputs.

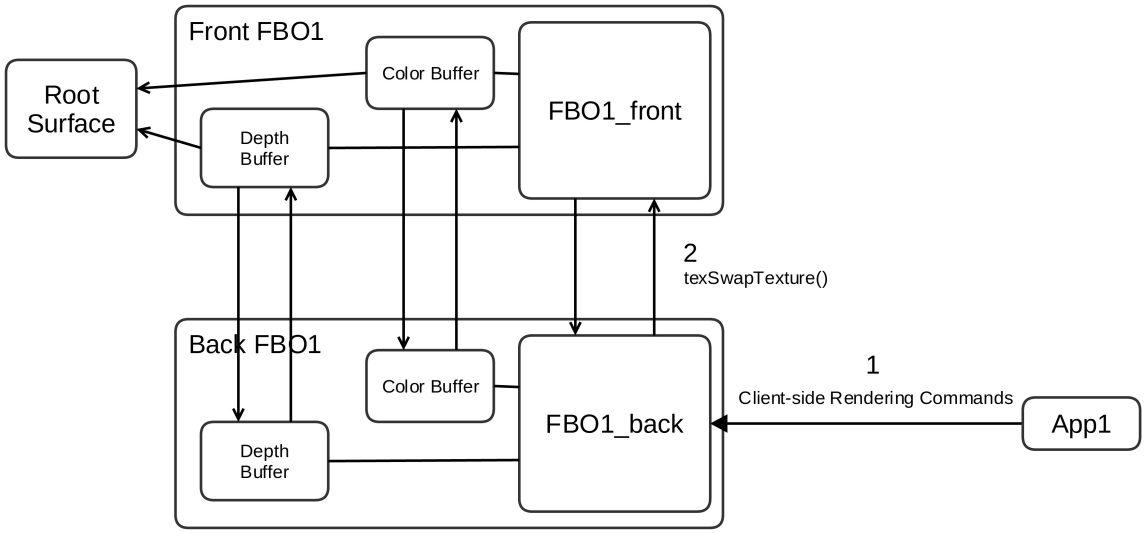


Figure 4.5: Double Buffering Implementation in Framebuffer Objects

In the workflow, back FBOs are always set as render targets of client-side application, and front FBOs always provide their attachments to root surface as rendering inputs. By using this design we ensure that all renderings on screen are complete.

4.4 Forwarding Replacement

The goal of this work is to make client-side applications work in the same way as they work on a native machine, without notifying any underlying implementation. Therefore a server-side virtualization is implemented with rendering commands replacement during the forwarding stage.

Specifically, for each client-side application, a pair of corresponding FBOs and attached textures should be created and bound. This process is different from a normal rendering process with single application owning one window. Therefore most EGL commands, such as **eglGetDisplay eglCreateWindowSurface**, are not necessary any more. Instead, we introduce new EGL and GLES commands helping to build and manage off-screen renderings to replace these old ones. Thus a forwarding replacement concept is a crucial part of this implementation.

We use a *C struct* `Application` to define and represent each client-side application for root surface to gather and use. The data structure of `Application` is as Figure 4.6. There are various pointers in each `Application` to store the necessary rendering data for each client-side application.

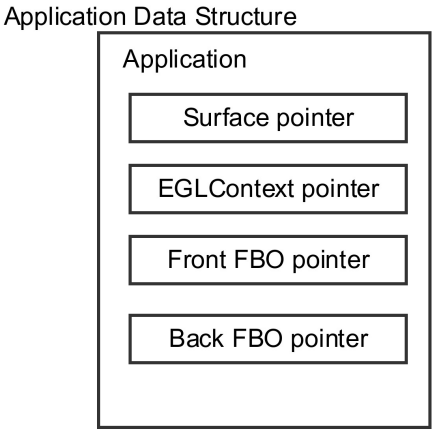


Figure 4.6: `Application` Data Structure

Figure 4.7 depicts the detail replacement strategies and interactions with `Application`, which is according to an order how an application renders.

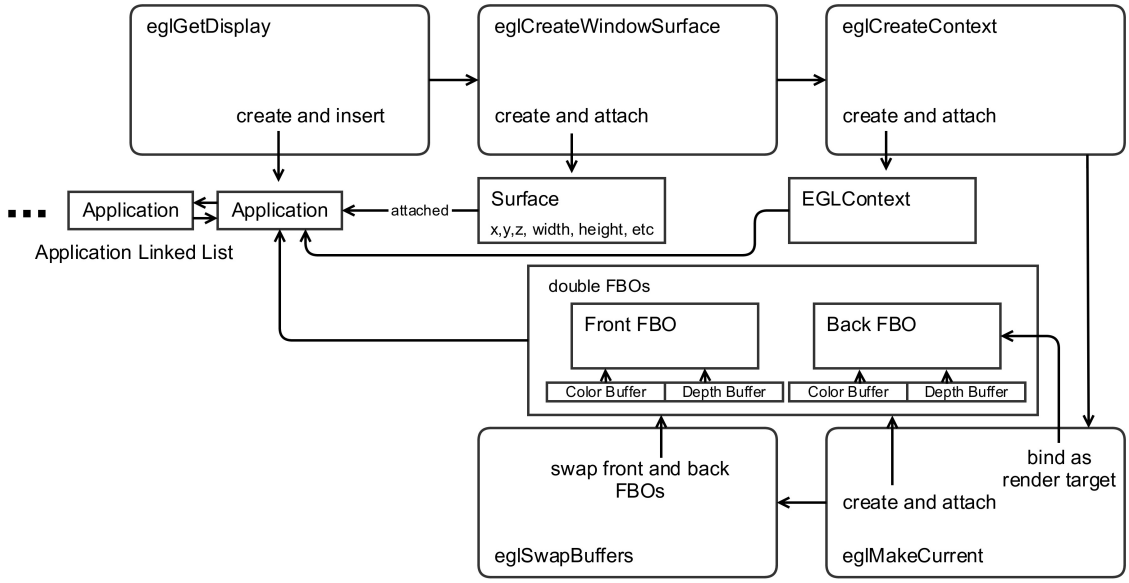


Figure 4.7: Replacement Strategies and Workflow

In the following Figure 4.7 is described in detail for each part.

eglGetDisplay

Originally by calling **eglGetDisplay** a native display can be obtained as the first step of rendering. With this work, a native display is already obtained by the root surface, because the final rendering takes place there. Therefore a second request for the display is not necessary here.

Because this function is always the first call of a client-side application, a new instance of `Application` is created and inserted to a linked list.

eglCreateWindowSurface

An EGL window surface is requested to be created when the function **eglCreateWindowSurface** is called. However, EGL surfaces are not necessary for off-screen rendering.

Instead, the data about the requested window surface, for instance, top-left corner coordinates (relative coordinates to root surface window), width and height, are passed to this `Application` built in **eglGetDisplay**. These are stored inside and used as position ans size parameters during rendering.

eglCreateContext

An `EGLContext` is created for OpenGL ES rendering via this function. And in this work an `EGLContext` is also necessary for rendering. `EGLContexts` for all applications are created during root surface initialization, as 4.8 describes. This is because in order to make root surface available to access rendered textures in each `EGLContext`, the root surface context should be passed as a *share_context* during `EGLContext` creation. Whereas an access to an `EGLContext` in another thread as a *share_context* is not possible, an `EGLContext` is created during root surface initialization (see details in Section 4.5) but used here.

Besides `EGLContexts` creation, **eglMakeCurrent** is also called here, to make corresponding `EGLContext` current for responding rendering commands.An EGL surface is necessary for making an `EGLContext` current, and a 1*1 size pbuffer is created and used for **eglMakeCurrent** surface parameter. After these operations, the current `EGLContext` are ready to handle rendering commands.

eglMakeCurrent

Here the corresponding FBOs along with attachments of a certain client-side application are created and bound for replacement. A pair of FBOs (front and back ones), color and depth buffer textures are created, and textures are attached with their FBOs respectively. At last **glBindFramebuffer** is called to bind the back FBO, being as the current target framebuffer.

So far all preparations are completed, and when client-side rendering commands are forwarded, this would handle them and render to textures of the back FBO.

eglSwapBuffers

This function is called when all rendering operations are finished and a full rendered image in back buffer is ready to be swapped to front buffer, according to Figure 2.5.

Similarly, a function **texSwapTexture** replaces the original one, swapping the back FBO with the front FBO including buffer attachments, according to step 2 in Figure 4.5. This ensures the front FBO attachments are always ready to be used for root surface rendering.

Also a flag is set here to notify the root surface that new rendering updates come for this FBO.

Above EGL function replacements virtualize the forwarding part, making itself transparent to client-side applications.

4.5 Root Surface Rendering

The root surface rendering part plays a role to composite all rendered textures and render to screen. Root surface runs in a separated thread, which is started at the beginning of the server program.

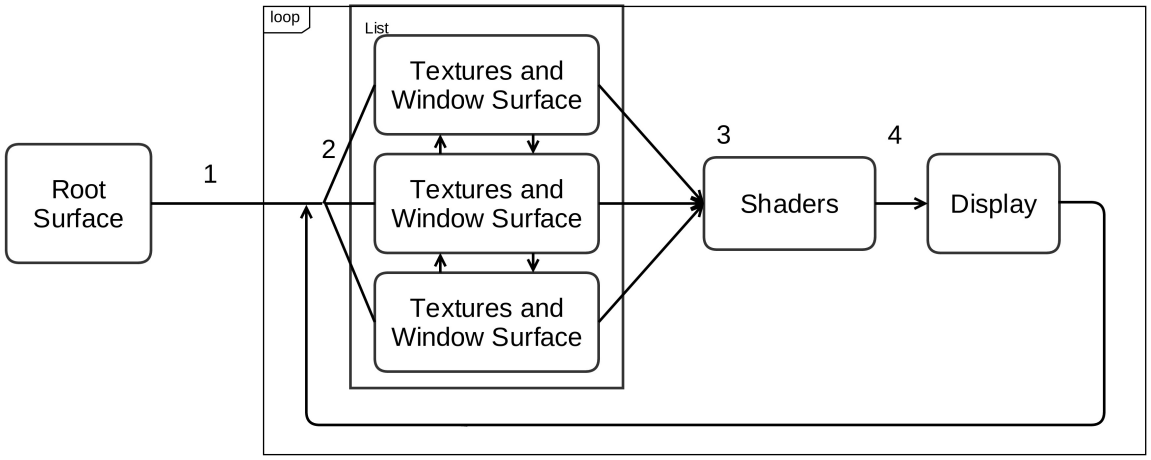


Figure 4.8: Root Surface Rendering

Figure 4.8 depicts the root surface rendering steps.

First At step 1 the root surface thread is created and joined. Then `rootSurfaceContext` is created as a type `ESContext` (a `C struct` defined in esUtil.h[14]), initialized. Furthermore, a native display is connected and an EGL window surface and an `EGLContext` are created for rendering. After initialization the `rootSurfaceContext` is ready to render.

Moreover, a number of `EGLContexts` are created for client-side applications off-screen rendering, according to the **eglCreateContext** replacement strategy in Section 4.4. The number of this depends on the maximal supported number of applications rendering at the same z level window surfaces. This is discussed in details in Subsection 4.5.2.

Second At step 2, a doubly linked list filled with texture contexts is accessed by `rootSurfaceContext`. A texture context includes rendering information of this corresponding client-side application, such as size and position of rendering window surfaces (talked in **eglCreateWindowSurface** replacement strategy in Section 4.4), color and depth textures which are ready to render. New texture contexts can be inserted into this list when the first swap buffer is called.

The window surface information of a texture context presents not only the window position in 2D plane, but also the value of z-axis position. Therefore a strategy to composite multiple application with different z value window surface is to draw these

surfaces one by one from far to near (relatively to the view point). For applications whose window surfaces have the same z value, a fragment shader is involved for a 3D compositing strategy (details are described in Subsection 4.5.2).

Third At step 3 a pair of appropriate vertex and fragment shaders are selected according to the number of surfaces rendering at the same z level.

Fourth By using shaders selected from last step, `rootSurfaceContext` renders all these textures to screen. And then it goes back to step 2 and continues looping.

Step 2 to 4 compose the `draw` function of this rendering loop. This is the on-screen rendering process of this work.

4.5.1 Non-Intersection Rendering Situation

As step 2 of 4.8 describes, if all window surfaces of client-side applications are at a different z level from each other, there would be no intersection at any surface. For this situation, the color buffer texture is rendered on corresponding window surface directly.

First the window surfaces list would be sorted according to z value. Second, `rootSurfaceContext` would sample each color texture and render them in the turn from bottom to top, according to z value. This ensures that window surfaces which are closer to the view point can override the covered part, when overlapping happens.

4.5.2 Intersection Rendering Situation

For a situation where multiple applications are rendered at the same z level of window surfaces, a concept of comparing depth texture value per fragment is a solution for displaying the intersection rendering result.

Fragment Shaders

In this situation, a fragment shader is used for comparing the depth information per fragment of all textures, displaying the color with a smallest depth value (which means it is nearest to the view point among all rendering contents). This operation is applied for each fragment.

Listing 4.1 is a code snippet for 3 applications rendering at the same window. `v_texCoord` is a `GLSL varying` passed from the vertex shader, as the texture coordinate. The function **texture2D** returns each texel of texture bound sampler according to `v_texCoord`.

```
1           depth0 = texture2D(s_depth0, v_texCoord);
2           depth1 = texture2D(s_depth1, v_texCoord);
3           depth2 = texture2D(s_depth2, v_texCoord);
4
5           tex0 = texture2D(s_texture0, v_texCoord);
6           tex1 = texture2D(s_texture1, v_texCoord);
7           tex2 = texture2D(s_texture2, v_texCoord);
8
9           if (depth0.r < depth1.r) {
10              if (depth0.r < depth2.r) {
11                  gl_FragColor = tex0;
12              } else {
13                  gl_FragColor = tex2;
14              }
15          } else {
16              if (depth1.r < depth2.r) {
17                  gl_FragColor = tex1;
18              } else {
19                  gl_FragColor = tex2;
20              }
21          }
```

List of Listings 4.1: Fragment Shader for 3D Compositing with 3 Applications

In line 9-21 of Listing 4.1, a selection algorithm is implemented to select the visible texture part for each fragment. The built-in variable `gl_FragColor` is assigned with color value outputted by this algorithm, and it finally displays on screen.

4.5.3 Maximal Textures Limitation

Theoretically, the implementation is supposed to handle unlimited number of applications to composite and display on screen simultaneously. However, plenty of limitations exist in practice, making this assumption difficult to realize. For instance, a maximal number of textures support is usually restricted by the hardware implementation and driver settings. At this situation, a new root surface rendering concept is raised, to solve the maximal textures limitation.

Suppose `m` is the maximal supported number of applications rendering at the same fragment shader. If the number of applications is now `m+1`, textures of the `(m+1)`th application cannot be sampled into the corresponding fragment shader due to exceeding the fragment shader capability. A solution to render the first `m` applications to a FBO, and then render the last application together with this FBO to the screen, regarding this FBO also as an application.

We call it *Double Render to Texture* as Figure 4.9 shows. For simplicity, in this figure each `Application` contains the rendered depth and color textures from previous steps. And the `Temp FBO` is a temporary FBO with depth and color textures attached. These attachments are render targets of *First Render* and render sources of *Second Render*.

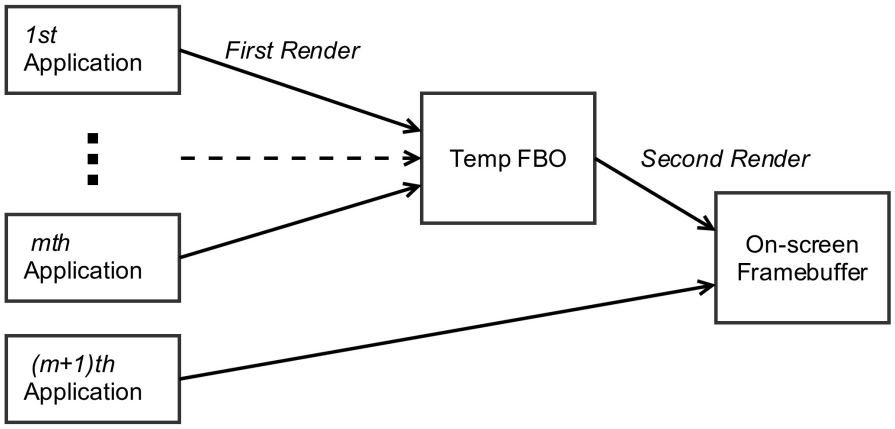


Figure 4.9: `m+1` Applications Rendering Strategy

For the situation with `2m` applications, similarly, a *Triple Render to Texture* is implemented, with two intermediate FBOs and three times rendering.

With this solution, regardless of performance and overhead, theoretically, there is not a maximal supported number of applications for this implementation.

4.6 Uniform Operations

Section 4.5 describes how the root surface implementation solves 3D compositing problem. Besides, another advantage by using this is to do uniform operations for all client-side applications.

Various uniform operations can be implemented with vertex and fragment shader, for instance, uniform lighting, shadowing, etc. Here a uniform directional lighting operation is implemented.

4.6.1 Phong Reflection Model

A Phong reflect model is used.for this lighting operation.

In this model, the final reflection is composed of three parts, ambient, diffuse and specular, as Figure 4.10 shows.

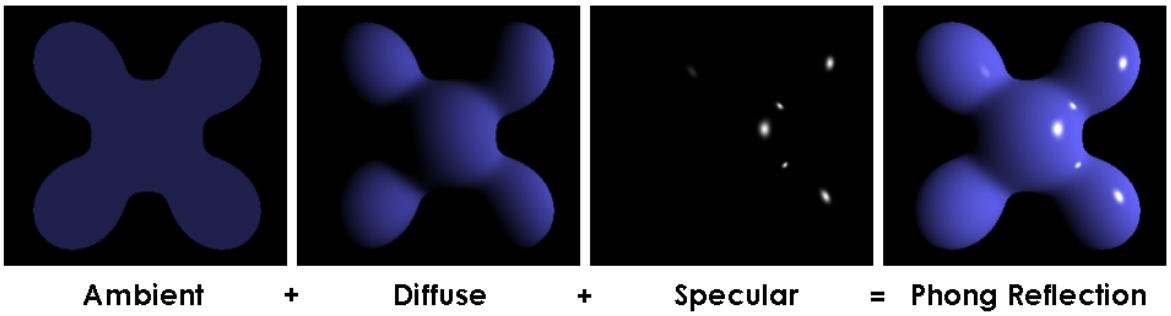


Figure 4.10: Phong Lighting Model [5]

- Ambient lighting refers to light reflections on the object surface which are not from the light source directly, but lights from environments or reflections. By using this a part that is sheltered totally from the light source would not be full dark, but with a low brightness, which makes the lighting simulation more lifelike.
- Diffuse lighting approximates the light reflections from the light source on the real world surface, which is not perfectly flat. Hence this describes reflections to other directions rather than a single reflection on a flat surface.
- Specular lighting simulates the lighting reflection which is reflected directly by the surface from a light source. This is usually shiny and localized.

4.6.2 Implementation

The lighting algorithm is mainly implemented in fragment shader. A number of `uniform` variables are used to pass input data to the fragment shader, such as normal vectors for each application, light position, view position, etc. A normal vector is a vector which is perpendicular to the surface of a vertex[6]. A `varying` variable `v_FragPos` with a coordinate value of each vertex is passed from vertex shader to fragment shader for calculation.

For simplification, the light color is set to be white.

Ambient Lighting

Ambient lighting is calculated with the original color multiplied by a constant factor. This factor could be set in the range (0, 1) according to the needs. Here the ambient factor is assigned to 0.1.

```
1      float ambient = 0.1;
2
3      gl_FragColor = ambient * original_color;
```

List of Listings 4.2: Ambient Lighting Sample Code

Listing 4.2 shows a simple way to simulate a ambient lighting effect with a preset ambient factor.

Diffuse Lighting

Diffuse lighting is impacted by the angle between the light direction vector and the normal vector of a vertex. As Figure 4.11, if the light ray is perpendicular to the surface of a vertex, a vertex has a maximal diffuse lighting value from this light source.

Therefore a dot product of the normalized normal vector and light direction vector could represent the diffuse lighting factor.

```
1      vec3 light_direction = normalize(light_position - v_FragPos);
2      float diffuse = max(dot(normal, light_direction), 0.0);
3
4      gl_FragColor = diffuse * original_color;
```

List of Listings 4.3: Diffuse Lighting Sample Code

This code (Listing 4.3) calculates the light direction for each fragment. Original color is multiplied by the calculated diffuse factor `diffuse`, resulting to a diffuse lighting.

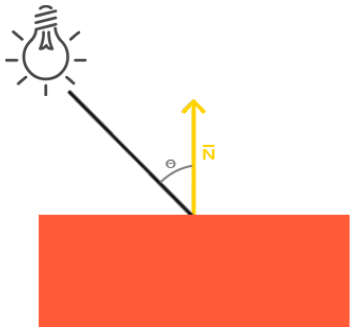


Figure 4.11: Diffuse Lighting [6]

Specular Lighting

A specular lighting calculation is similar with the diffuse lighting. As Figure 4.12, the specular lighting is impacted with the angle between a view direction and the reflected light direction. Here the reflected light refers to the light source reflected by the flat surface, with a single reflection only.

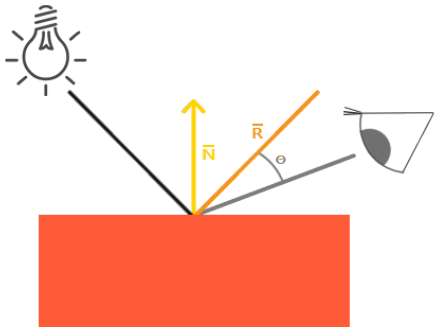


Figure 4.12: Specular Lighting [6]

Also, we use a dot product of the normalized reflection direction vector and the view direction vector to represent the specular factor.

```
1     vec3 view_direction = normalize(view_position - v_FragPos);
2     vec3 reflect_direction = reflect(-light_direction, normal);
3     float specular = pow(max(dot(view_direction, reflect_direction), 0.0), 32.0);
4
5     gl_FragColor = specular * original_color;
```

List of Listings 4.4: Specular Lighting Sample Code

During the specular factor `specular` calculation in Listing 4.4, a power of 32 is calculated to the result of dot product, for making the specular lighting highlighted to certain point.

Code Example

Listing 4.5 is a code snippet of a fragment shader for 2 applications situation.

```
1   vec4 original_color;
2   vec3 normal;
3
4   float ambient = 0.1;
5
6   vec3 light_direction = normalize(light_position - v_FragPos);
7   vec3 view_direction = normalize(view_position - v_FragPos);
8
9   vec4 depth0 = texture2D(s_depth0, v_texCoord);
10  vec4 depth1 = texture2D(s_depth1, v_texCoord);
11  vec4 tex0 = texture2D(s_texture0, v_texCoord);
12  vec4 tex1 = texture2D(s_texture1, v_texCoord);
13
14  if (depth0.r < depth1.r) {
15      original_color = tex0;
16      normal = normal_tex0;
17  } else{
18      original_color = tex1;
19      normal = normal_tex1;
20  }
21
22  float diffuse = max(dot(normal, light_direction), 0.0);
23  vec3 reflect_direction = reflect(-light_direction, normal);
24  float specular = pow(max(dot(view_direction, reflect_direction), 0.0), 32.0);
25
26  gl_FragColor = (ambient + diffuse + specular) * original_color;
```

List of Listings 4.5: Uniform Phong Lighting Model Implementation

The 3D compositing algorithm is modified to adapt this operation. For normal vectors selection, it is similar with the solution mentioned in Subsection 4.5.2, which is according to the depth comparison. Comparing line 14-20 with the selection algorithm in Listing 4.1, the normal vector selection is implemented with the same method. At the end all three factors are summed up and affect on the original color.

5 Evaluation

In this chapter evaluation about this work is elaborated. First an overview of the hardware and software setups is presented for a better understanding of the evaluation environment. In the following parts there are various scenarios to evaluate and verify the concept of this work.

The 3D compositing capability is verified first, with exploring the performance variation tendency according to the increasing number of applications. Then the result will be compared with those of other scenarios, to analyze the performance.

To verify the monolithic rendering, a uniform lighting operation is evaluated at the end with a Phong lighting model implemented on a 2-triangle intersection scenario.

5.1 Evaluation Setup

5.1.1 Hardware Setup

The 3D compositing concept is implemented on Freescale i.MX6 Quad Automotive board, which is an embedded board specifically for automotive and infotainment applications. The main hardware configuration is as below[15]:

- **CPU:** ARM Cortex A9 Quad-Core 4 x 1.2 GHz
- **RAM:** 2 GB
- **GPU 3D:** Vivante GC2000
- **GPU 2D(Vector Graphics):** Vivante GC355
- **GPU 2D(Composition):** Vivante GC320

Specially, the 3D GPU, Vivante GC2000, has a OpenGL ES 2.0 graphics accelerator support[16]. Namely, calling OpenGL ES 2.0 commands directly can benefit from GPU 3D acceleration. Freescale i.MX6 graphic hardware has a full `GL_OES_depth_texture` support, which makes it possible to use depth texture as a FBO depth buffer attachment.

On Freescale i.MX6 board, a maximal supported number of textures in one fragment shader is 8. Because for each application, there are at least two textures representing, color buffer texture and depth buffer texture. Hence for this board, 4 (8 divided by 2) applications are capable to render at the same time. With the solution in 4.5.3, a performance degradation is expected when the number of applications is large enough.

Besides, up to four displays can be connected to this board at the same time, including parallel display, HDMI1.4, MIPI display, and LVDS display[17]. During the evaluation the HDMI port is used to output the display content to a monitor.

5.1.2 Software Setup

A real-time system, PikeOS, is installed as the operating system on the i.MX6 board, on which three virtual machines are running. A server virtual machine behaves as the server-side for in Chapter 3. The other two virtual machines behave as client-side.

5.1.3 Test Applications

For performance evaluation, `glmark2` is used. `glmark2` is an OpenGL 2.0 and OpenGL ES 2.0 benchmark[18].

`glmark2` has numerous built-in 3D models. Options can be given to select and switch on and off features about `glmark2` during execution. For example, the following code means to render the built-in model horse for benchmark by rendering 10,000 frames (default value).

```
./glmark2-es2 -b build:model=horse
```

The result of each `glmark2` execution is a `glmark2` score based on the average Frames per Second(FPS) of the whole rendering process. This result reflects an overview of the current rendering performance.

By using `glmark2` built-in models a relatively complex rendering situation is simulated. In addition, a uniform benchmark is also a good measurement to compare the performance between different implementations. Therefore, `glmark2` is used to evaluate this work.

5.2 3D Compositing Verification

A fundamental feature of this work is the compositing for 3D applications. The first evaluation scenario is set to measure the performance variation tendency by increasing the number of applications.

In this scenario, `glmark2` is used with different built-in 3D models in different applications.

Input details of this scenario is as Table 5.1. `glmark2-es2` with a number appended at the end is the client-side `glmark2-es2` application. Each of them is with a native window requested for 540x540 resolution, same z position at the same area.

Application	Command	Window Size	z Position
1	`./glmark2-es2-1 -b build:model=horse`	540x540	1
2	`./glmark2-es2-2 -b build:model=dragon`	540x540	1
3	`./glmark2-es2-3 -b build:model=cat`	540x540	1
4	`./glmark2-es2-4 -b build:model=bunny`	540x540	1
5	`./glmark2-es2-5 -b build:model=buddha`	540x540	1
6	`./glmark2-es2-6 -b build:model=horse`	540x540	1
7	`./glmark2-es2-7 -b build:model=dragon`	540x540	1
8	`./glmark2-es2-8 -b build:model=cat`	540x540	1
9	`./glmark2-es2-9 -b build:model=bunny`	540x540	1
10	`./glmark2-es2-10 -b build:model=buddha`	540x540	1

Table 5.1: 3D Compositing Verification Scenario

5.2.1 Results and Analysis

Rendering Results

The 3D compositing result is as Figure 5.1.

Figure 5.1: Horse Model and Dragon Model Intersection Scene

Two applications render horse and dragon models respectively simultaneously, depicted in Figure 5.1. At the part where the front legs of the horse model intersect with the horn of the dragon model, one leg comes out through the gap between the horn and the head of dragon. This proves that the 3D compositing is working as the expectation.

Performance Analysis

The performance results(FPS) according to the number of 3D compositing applications is as Figure 5.2:

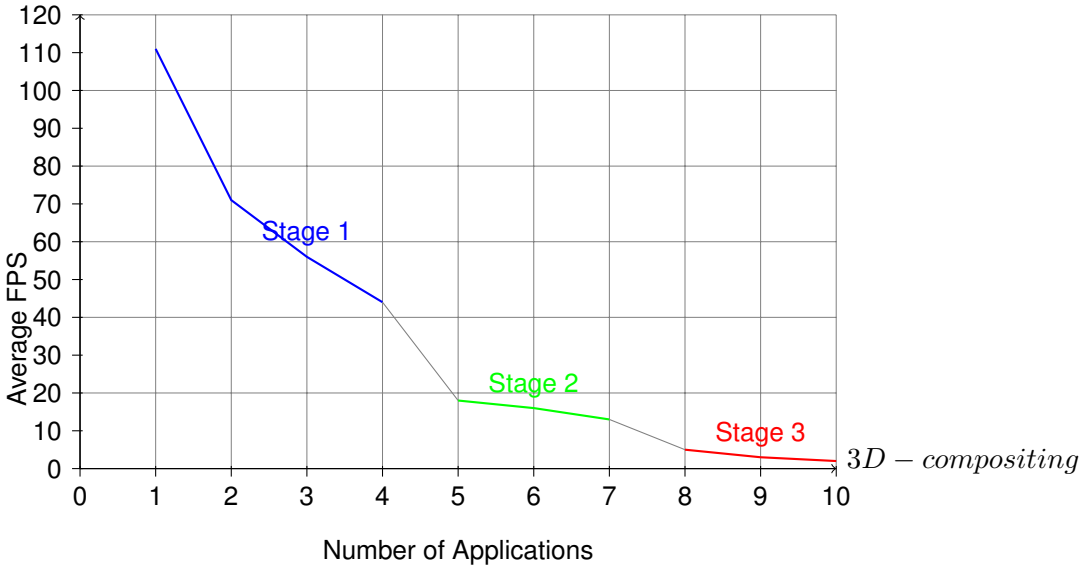


Figure 5.2: Performance Comparison in 3D Compositing Scenario

This line graph shows, in general, the performance decreases with the number of applications increasing.

Specifically, a number of stages are shown on this graph. First, when the number of textures increases from 1 to 2, the performance declines rapidly, comparing to the descending ranges from 2 to 4 in x axis. Same phenomenon happens also from 4 to 5 and from 7 to 8 in x axis. According to this, the whole line graph is divided to four stages, (1), (2,4), (5,7) and (8,10) in x axis.

Stage 1 contains only the situation when there is one application, where 3D compositing is not in use, and compositor just simply forwards the color buffer of this application to on-screen framebuffer. In the situation that 2 applications are rendering simultaneously, 3D compositing is invoked, which has a heavier CPU and GPU overhead than that in stage 1. Therefore a sharp decline happens between these two stages.

On stage 2, 3D compositing is invoked. As the number of applications increases, the maximal number of textures is reached, and performances drop again to stage 3. On stage 3, the *Double Render to Texture* algorithm (mentioned in 4.5.3) is used. With two times *Render to Texture*, a performance degradation is inevitable.

Similarly, on stage 4 a *Triple Render to Texture* is applied, which leads a further performance degradation.

To sum up, the number of *Render to Texture* times has a major influence on performances. Different stages have a distinct performance differences between each other.

5.3 Performance Comparisons with Native Applications

The scenario in this section focuses on comparing the performance between 3D compositing and native application executions. Strictly speaking, this comparison is not with equal conditions and outputs, because native applications execution do not have any compositing related implementation. This comparison is only for evaluating the 3D compositing performance.

5.3.1 Scenario Description

This scenario compares the performances between 3D compositing applications with applications running natively on server-side.

In this scenario, multiple `glmark2` applications executes simultaneously on server-side natively, without compositing. Theriotically, the performance of native execution applications should be the maximal performance that can be ever reached.

Input details of this scenario is as Table 5.2. `glmark2-es2-native` with a number appended at the end is the server-side `glmark2-es2` application. Each of them is with a native window requested for 540x540 resolution, same z position at the same area.

Application	Command	Window Size	z Position
1	`./glmark2-es2-native-1 -b build:model=horse`	540x540	1
2	`./glmark2-es2-native-2 -b build:model=dragon`	540x540	1
3	`./glmark2-es2-native-3 -b build:model=cat`	540x540	1
4	`./glmark2-es2-native-4 -b build:model=bunny`	540x540	1
5	`./glmark2-es2-native-5 -b build:model=buddha`	540x540	1
6	`./glmark2-es2-native-6 -b build:model=horse`	540x540	1
7	`./glmark2-es2-native-7 -b build:model=dragon`	540x540	1
8	`./glmark2-es2-native-8 -b build:model=cat`	540x540	1
9	`./glmark2-es2-native-9 -b build:model=bunny`	540x540	1
10	`./glmark2-es2-native-10 -b build:model=buddha`	540x540	1

Table 5.2: Native Applications Comparison Scenario

5.3.2 Results and Analysis

The rendering result is a continuously flashing screen mixed with several models overlapping due to a lack of compositing implementation.

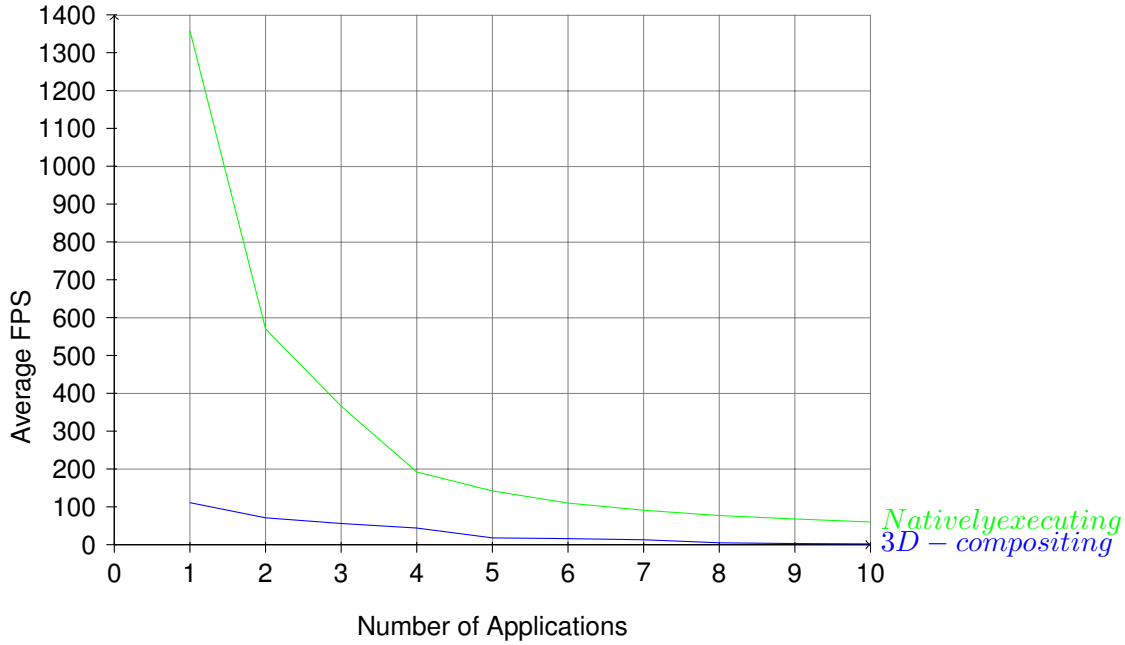


Figure 5.3: Performance Comparison between 3D Compositing and Native Applications

Figure 5.3 illustrates that, native application performance has a large advantage over 3D compositing performance, as we expect. However, this advantage becomes less with the number of applications increasing.

Therefore, a optimization potential for the 3D compositing is possible in theory, due to the large performance differences.

5.4 Performance Comparisons with Non-Intersecting Applications

The scenario in this section focuses on comparing the performance between 3D intersecting applications compositing and non-intersecting applications compositing situations. This comparison is for evaluating the 3D compositing performance with different situations.

5.4.1 Scenario Description

As section 4.8 presents, compositing strategy differs from the z position of window surfaces. 3D compositing algorithm is invoked only when application window surfaces are on the same z level, otherwise all applications are rendered to screen in the turn of z value ascending (which is from far to near in terms of view point). This scenario compares the performances between 3D compositing applications and non 3D compositing applications.

Input details of this scenario is as Table 5.3. `glmark2-es2` with a number appended at the end is the client-side `glmark2-es2` application, which is the same applications with those in Table 5.1. Each of them is with a native window requested for 540x540 resolution at the same area, but with a different z position.

Application	Command	Window Size	z Position
1	`./glmark2-es2-1 -b build:model=horse`	540x540	1
2	`./glmark2-es2-2 -b build:model=dragon`	540x540	2
3	`./glmark2-es2-3 -b build:model=cat`	540x540	3
4	`./glmark2-es2-4 -b build:model=bunny`	540x540	4
5	`./glmark2-es2-5 -b build:model=buddha`	540x540	5
6	`./glmark2-es2-6 -b build:model=horse`	540x540	6
7	`./glmark2-es2-7 -b build:model=dragon`	540x540	7
8	`./glmark2-es2-8 -b build:model=cat`	540x540	8
9	`./glmark2-es2-9 -b build:model=bunny`	540x540	9
10	`./glmark2-es2-10 -b build:model=buddha`	540x540	10

Table 5.3: Non-Intersecting Applications Comparison Scenario

5.4.2 Results and Analysis

Because of different z position is used on different applications, only the application on top of all is displayed, according to 4.5.1. In this case, the first one with a horse model is displayed only.

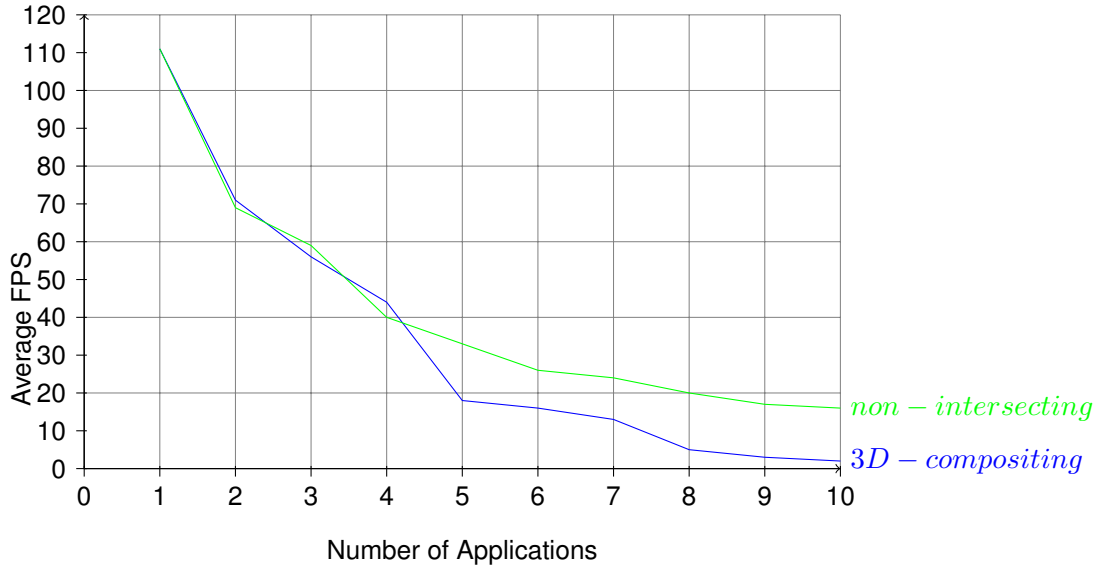


Figure 5.4: Performance Comparison between 3D Compositing and Non-Intersecting Applications

Figure 5.4 depicts the performance comparison result in this scenario.

These two lines have similar trends at the first half as the number of applications increases. However, unlike the distinct stages appearing on 3D compositing performance graph, the performance of non-intersecting applications declines gradually with applications increasing. This is because there is no implementations differences with applications increasing at this situation, and the rendering overhead is directly proportional to the number of applications, without sharp decline points. Hence, at the second half of this graph, the non-compositing situation shows a better performance than the 3D-compositing situation.

Nevertheless, the non-intersecting situation shows an average performance in total. Considering this work is mainly targeted to 3D compositing situation, the non-compositing implementation has also performance sacrifices due to this *Render to Texture* design concepts and implementations.

5.5 Performance Comparisons with 2D Compositor

The scenario in this section focuses on comparing the performance between the 3D compositing and a 2D compositor. This comparison is for evaluating the 3D compositing performance with other related implementations.

5.5.1 Scenario Description

The 2D compositor we use to compare with is implemented mainly using 2D GPU commands with bitblitting algorithms, which is optimized by reducing drawing for the overlapped parts[19].

This scenario compares the 3D compositing implementation with the 2D compositing implementation with both non-intersecting applications rendering situation and 3D intersecting situation.

Input details of this scenario is as Table 5.4. The input data of this scenario is the same with that of non-intersecting scenario.

Application	Command	Window Size	z Position
1	`./glmark2-es2-1 -b build:model=horse`	540x540	1
2	`./glmark2-es2-2 -b build:model=dragon`	540x540	2
3	`./glmark2-es2-3 -b build:model=cat`	540x540	3
4	`./glmark2-es2-4 -b build:model=bunny`	540x540	4
5	`./glmark2-es2-5 -b build:model=buddha`	540x540	5
6	`./glmark2-es2-6 -b build:model=horse`	540x540	6
7	`./glmark2-es2-7 -b build:model=dragon`	540x540	7
8	`./glmark2-es2-8 -b build:model=cat`	540x540	8
9	`./glmark2-es2-9 -b build:model=bunny`	540x540	9
10	`./glmark2-es2-10 -b build:model=buddha`	540x540	10

Table 5.4: 2D Compositing Comparison Scenario

5.5.2 Results and Analysis

As this compositor has only the ability to composite 2D scenario, the testing applications are with different z positions. And the result is the same with non-intersecting scenario, which is only the first model is displayed due to its z position.

Figure 5.5 illustrates a performance comparison between current compositor, including 3D compositing and non-intersecting situations, and previous 2D compositor. The first half of 2D compositor performance graph is nearly level, because a reduce FPS strategy is applied when the FPS is over 60. At the second half, the performance of 2D compositor drops linearly. Nonetheless, in general, the 2D compositor has a better performance than both the 3D

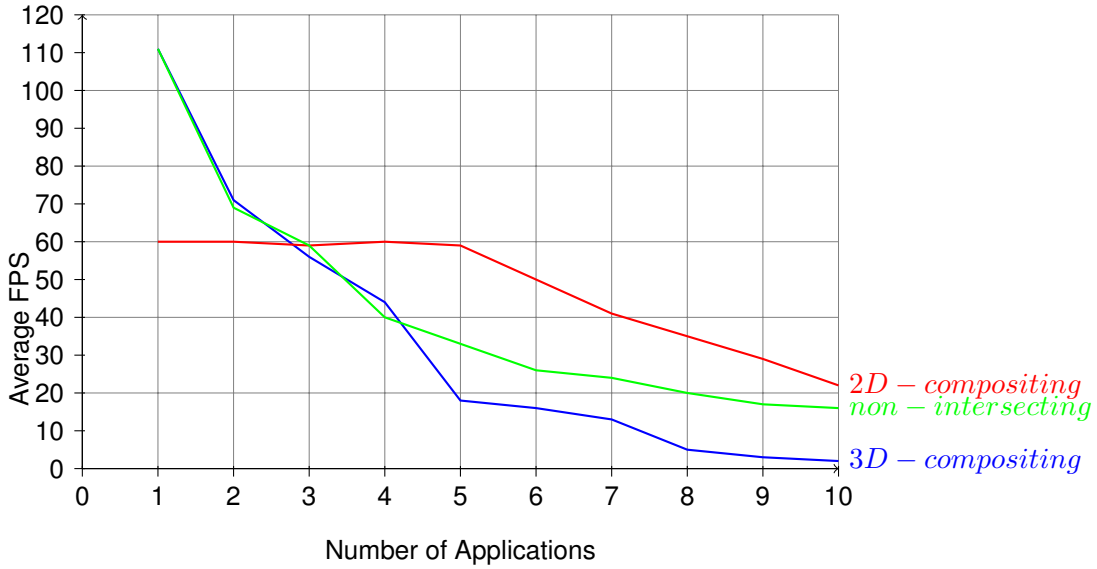


Figure 5.5: Performance Comparison between 3D Compositing, Non-Intersecting Compositing and 2D Compositing

compositing and the non-compositing situations of this work. This could be explained by the overhead of *Render to Texture*.

5.6 Monolithic Rendering Operations

Monolithic rendering operations are available in this work. For an evaluation purpose, one scenario with the uniform lighting is implemented to verify this feature.

In this scenario, a simple uniform lighting operation is implemented on a 2-triangle intersecting scene. These two triangles intersect with each other as Figure 5.6. A lighting source is set between these two in front of the viewing point. The rendering result with lighting effect is as Figure 5.6.

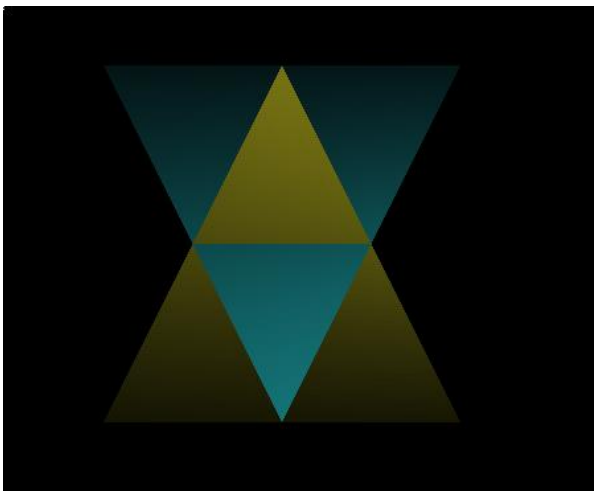


Figure 5.6: Uniform Lighting for 2-Triangle Scenario

Figure 5.6 shows that the lightness varies from the near part to the far part. And at the far end, it becomes nearly black.

We move the lighting source nearer to these two triangles, and this time the rendering result is as Figure 5.7.

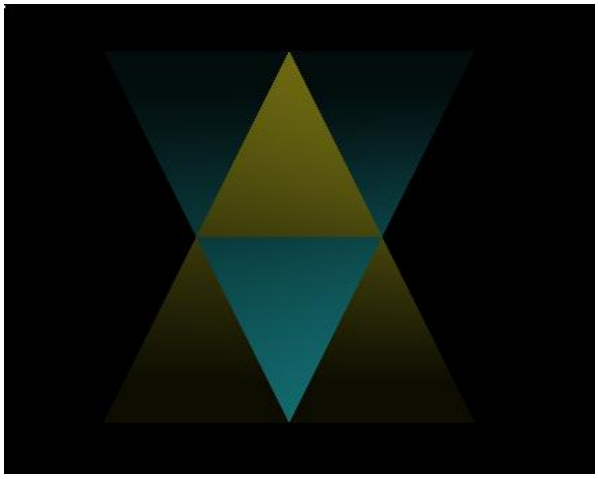


Figure 5.7: Uniform Lighting Effect when Light Source Moves Nearer to Triangles

Figure 5.7 depicts that as the lighting source becomes nearer to the triangles, the diffuse lighting, which is the main composing part of the lighting model, is weaker for the far end points, due to a larger angles between the light direction and the normal vector for those points, referring to Section 4.6. Thus the far end parts is darker than Figure 5.6

Results of this uniform lighting operation is as expected. Other monolithic rendering operations are with similar principles. Therefore, the monolithic rendering feature is proven.

5.7 Summary

The basic goal of this implementation is achieved, for a 3D compositing rendering. It is stable with multiple applications tests. A uniform operation performance also meets the expectation.

After performance comparisons in various scenarios, a conclusion has been made, that the performance of this work still has potential to optimize. In order to composite 3D applications, a performance sacrifice is inevitable. Moreover, the hardware limitation is another obstacle for the 3D compositing performance.

For a 3D compositing situation with the number of applications less than or equal to 4 on this board, the performance is still good and acceptable.

6 Related Work

With a continuously growing demand of graphical applications, various window managers exist to accomplish different tasks. A few instances are referred and described in this chapter.

3D Composting Window Manager by Mayerle

Mayerle's work[20] about a 3D compositing window manager concept is one basis of this work. In Mayerle's work, a 3D compositing concept is implemented with off-screen renderings and command replacement strategies. Based on that, this work adapt the 3D compositing concept to a unifying platform. Furthermore, this work extends the capability with monolithic rendering operations, which makes the system more flexible to manipulate all applications uniformly.

Cache-Hybrid compositing by Gansel

Gansel's work[21] introduces a new compositing strategy named *Cache-Hybrid compositing*. In 2D compositing concepts, *full compositing* and *tile compositing* are two strategies to implement bitblitting. This concept describes a strategy to predict the execution time of each bitblitting operation, with which an appropriate bitblitting strategy is assigned. A cached prediction model is also established to reduce the CPU overhead of prediction execution. This is proven to be an efficient strategy with 2D compositing. However, in case of a 3D model intersection scenario, it is still not capable.

X Window System

The X window system (also as know as X11) is a windowing system widely used on Unix-like operating systems. In X window system, a composite extension *Xcomposite* is a popular extension for 2D compositing[22]. It is implemented with bitblitting and a *full compositing* strategy is chosen. The concepts of *Xcomposite* are aimed at desktop environment mainly. For embedded systems, these concepts are not suitable any more. Besides, 3D compositing concept is also usually ignored by X11 related compositors.

Summary

Most currently existing window manager implementations are aimed at 2D compositing. With a trend of 3D models increasing, window managers with a 3D compositing feature would be necessary. Also, a monolithic rendering concept is another useful feature for embedded systems.

7 Conclusion and Outlook

Currently, 3D graphical applications play an important role in automotive platforms. Traditional 2D compositors have the capability to cope with simple 3D compositing scenarios, whereas for complex 3D compositing scenarios, for example, 3D models intersection, the disadvantage of 2D compositors stands out. For the purpose to handle complex 3D compositing scenarios, a 3D compositing implementation is necessary.

The implementation of this consolidated window manager uses off-screen rendering to store render result of client-side applications into textures, and then render all rendered textures to screen with a 3D compositing implementation. During the rendering to screen process, other implementations besides 3D compositing are also possible, such as uniform lighting operations, shadow mapping, etc, to make a monolithic rendering affecting all rendered applications.

According to various evaluations, this 3D compositing implementation handles multiple intersecting 3D applications with a good performance. And the monolithic rendering performance is also as good as expectation.

Outlook

Possible optimization might be made for further development. The performance of this work could be improved by learning from the existing 2D compositing strategies when it copes with non-intersecting applications compositing scenarios, because in these scenarios this work is not fully optimized, whereas 2D compositors have a better performance due to various compositing strategies.

Moreover, besides the uniform lighting operation, other kinds of monolithic rendering operations could be implemented, to maximize the benefits from this implementation and to fulfill more practical and useful features.

A Glossary of Acronyms

LCD - Liquid-Crystal Display

API - Application Program Interface

GLSL - OpenGL Shading Language

FBO - Framebuffer Object

IoV - Internet of Vehicles

ECU - Electronic Control Unit

GPU - Graphics Processor Unit

GLES - OpenGL ES

GUI - Graphical User Interface

OS - Operating System

CPU - Central Processing Unit

RAM - Random-Access Memory

VM - Virtual Machine

Bibliography

[1] "The mercedes-benz f 015 luxury in motion research vehicle." (Cited on pages 7 and 14)

[2] https://en.wikipedia.org/wiki/Windows_Aero. (Cited on pages 7 and 19)

[3] D. S. Aaftab Munshi, Dan Ginsburg, *OpenGL ES 2.0 Programming Guide*. Addison-Wesley Professional, 2008. (Cited on pages 7, 20, 34, 35, 37 and 38)

[4] J. de Vries, "Learn opengl." http://learnopengl.com/#!Getting-started/Textures. (Cited on pages 7 and 21)

[5] https://en.wikipedia.org/wiki/Phong_reflection_model. (Cited on pages 7 and 48)

[6] J. de Vries, "Learn opengl." http://learnopengl.com/#!Lighting/Basic-Lighting. (Cited on pages 7, 49 and 50)

[7] http://en.wikipedia.org/wiki/Internet_of_Things. (Cited on page 13)

[8] Huawei, "Internet of vehicles: Your next connection." http://en.wikipedia.org/wiki/Internet_of_Things. (Cited on page 13)

[9] https://www.mercedes-benz.com/en/mercedes-benz/innovation/research-vehicle-f-015-luxury-in-motion/. (Cited on page 13)

[10] https://en.wikipedia.org/wiki/Window_manager. (Cited on page 18)

[11] https://en.wikipedia.org/wiki/OpenGL_ES. (Cited on page 20)

[12] https://en.wikipedia.org/wiki/EGL_(API). (Cited on page 23)

[13] https://www.khronos.org/registry/egl/sdk/docs/man/html/eglIntro.xhtml. (Cited on page 34)

[14] http://opengles-book-samples.googlecode.com/svn/trunk/LinuxX11/Common/esUtil.h. (Cited on page 44)

[15] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q. (Cited on page 54)

[16] http://www.vivantecorp.com/en/technology/3d.html. (Cited on page 54)

[17] F. Semiconductor, "i.mx6 dual/6quad automotive and infotainment applications processors data sheet," 2014. (Cited on page 54)

[18] https://github.com/glmark2/glmark2. (Cited on page 55)

[19] R. Cecolin, "Compositing concepts for the presentation of graphical application windows on embedded systems," 2014. (Cited on page 65)

[20] M. Mayerle, "Konzeption und realisierung eines window managers für die darstellung von 3d-inhalten unabhängiger opengl es 2.0 anwendungen," 2012. (Cited on page 71)

[21] R. C. F. D. K. R. C. M. Simon Gansel, Stephan Schnitzer, "Efficient compositing strategies for automotive hmi systems." (Cited on page 71)

[22] http://www.x.org/archive/X11R7.5/doc/man/man3/Xcomposite.3.html. (Cited on page 71)

All links were last followed on July 20, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature