國立中山大學資訊工程學系

碩士論文

Department of Computer Science and Engineering

National Sun Yat-sen University

Master Thesis

自動產生與使用人形看板來加速物件繪製之方法

A Method for Automatically Creating and Using billboards to Increase the Speed of Object Rendering

研究生：林君勵

Chin-Li Lin

指導教授：Dr. Steve.W.Haga

中華民國 103 年 12 月

December 2014

國立中山大學研究生學位論文審定書

本校資訊工程學系碩士班

研究生林君勵（學號：M013040022）所提論文

自動產生與使用人形看板來加速物件繪製之方法

A method for automatically creating and using billboards to increase the speed of object rendering

於中華民國 103 年 11 月 7 日經本委員會審查並舉行口試，符合碩士學位論文標準。

學位考試委員簽章：

召集人 黃英哲	黃英哲	委　員 希家史提夫	[illegible]
委　員 張雲南	張雲南	委　員 謝東佑	謝東佑
委　員		委　員	

指導教授(希家史提夫) [illegible] (簽名)

摘要

雖然圖形處理器的繪圖速度已有長足的進步，在及時繪圖與互動繪圖的應用上還是有不足的地方，對於嵌入式的繪圖處理器而言，更是一大挑戰。在繪圖速度的限制下，程式設計師們無不想方設法地減少繪圖時的運算量，impostor 的使用就是一個例子，impostor 是一個軟體的機制，物件首先被繪入貼圖中，往後便利用此貼圖來造成繪製原物件的效果，impostor 的確大量地減少了計算量。

在這篇論文中，我們看到 impostor 的顯著效果，也看到了其背後繁複的實作細節，於是我們提出一套硬體的機制，目地是要讓遊戲設計者得到 impostor 帶來的效能提升，卻不用實作繁複的程式細節，另外，由於我們在硬體上實作，我們可以得到在軟體上無從得知的有用資訊，這使得我們在記憶體的使用上更有效率，也使得此方法可以減少更多的計算量，最後，本篇論文可使整體指令的計算量減少 10%以上。

關鍵字: 繪圖晶片，硬體加速，impostor，billboard。

Abstract

Although the rendering speed of modern GPUs is dramatically improved, it is still not fast enough for some applications such as real time rendering and 3D interactive rendering. Many game developers figure out many methods to reduce the computations of GPUs. One of them is impostor. The impostor method first draw the object into a texture and then apply the texture on a quad or two triangles to generate the illusion of the object. Since the two triangles replace the thousands of triangles, we can reduce a lot of computations.

In this paper, we try to acquire all the benefits of impostor but hide the complex implementation details. Therefore, we propose a hardware mechanism to implement the impostor inside the hardware. With this, game developers do not need to worry about the implementations, because the hardware apply the method automatically. Moreover, since we implement the impostor in the hardware level, we can get some useful data which cannot see in the software. The data help us to apply the impostor technique and use the memory space more efficiently. After all, we can reduce more than 10% number of instructions of the whole GPU system.

Keywords: GPU, hardware, impostor, billboard.

Contents

Lists of figures

A Method for Automatically Creating and Using Impostors to Increase the Speed of Object Rendering

Author: Chin-Li Lin
Advisor: Dr. Steve.W.Haga
National Sun Yat-Sen University

1. Introduction

Handhold devices are now very widely used, such as smartphone and pad. These products are small, light, short battery life and lacking for computing power. Most of the time people run video games on these devices. Since it requires lots of graphic processing when running a game, many modern handhold devices have graphics processing unit (GPU). Although modern GPUs improve the performance of rendering scenes dramatically, it is still not enough for running fancy games, especially on embedded systems. In many games, in order to make the scenes look fancy, game developers elaborate many delicate but complicated objects that have enormous number of polygons. Games rely on real time rendering. That means if it cannot keep a high frame rate (at least 20 frames per second) rendering, the users will be disturbed by the gaps between two frames and are probably not willing to play this game anymore. There are also other applications suffer from limited computing power such as interactive 3D graphics system. Under the premise that the limited computing power of embedded system and high frame rate requirement, not only game developers but also researchers propose methods to speed up the scene rendering.

In the real world, a life-sized cardboard cut-out of a politicians are commonly seen in governments. It provides the illusion that the person is there, but the really thing there is the cardboard cut-out of the person. In the same way, in a game, cardboard cut-out is a common technique called impostor used to speed up the rendering process. An imposters is made by an objects is rendered into a texture and of course, the actually object is not rectangular so that the areas which beyond the outline of the object should be transparent. We got an impostor now, we then need to apply the impostor on a quad or two triangles. After that, object may appear to be in the three dimensions, but in fact actually is in the two dimensional rendering on a quad. There are some situations lead the illusion be ruined, firstly, when the camera gets too close to the impostor of an object, it causes a pixelation problem. Secondly, when either the object rotates or the camera changes its orientation, the quad which the imposter was applied on become thinner and pixels of the imposter are messed up. Thirdly, when panning the camera, some objects get more and more pixels. That means it is going to show pixels that we did not hold. Fourthly, when strafing the camera, the third situation is shown again. Moreover, the player is going to see the side of the object which the impostor did not hold.

Now, we know how people create illusions. On some game development web sides [2] [3], we can clearly see the steps we need to follow to generate imposters and apply it on a quad. Here are steps to follow: (a) to generate imposters, we need to introduce a technique called frame buffer object (FBO). This technique allows us to set up a new rendering target which is a texture instead of frame buffer for an object who wants to generate an imposter. To use FBO, game developers should search the API calls and call them correctly. (b) To prevent using imposter from previous situations (pixelation, rotation, panning and strafing), game developers and researchers propose some tests to detect those situations. These tests are described in [3]. (c) To apply imposter texture on a quad or two triangles, there are a series of API functions to call. In addition, it requires game developers figure out where to put the imposters and because the imposter has transparent areas, game developers need to be careful when applying the imposters. (d) To generate and using imposters on the fly for multiple objects, the game developers must create a data structure to manage each imposter for each object. (e) To avoid memory waste, game developers need to allocate and free the memory space carefully.

In actually games, it is common to have objects that sometimes do not change or change slightly from frame to frame, for example, the game player may walk down the hall and objects get closer but then he stops walking and objects stop changing. So, there will be times when the objects are not changing or slightly changing. In addition, the player may sometimes straight spin (turn his view or turn around). In this case, even though the objects move, the actual pixels that are drawn may not change much. In other cases, the objects may change but not very quickly, for instance, if he is walking down the hall slowly, objects are coming closer but slowly. That is why game developers may acquire a significant improvement in rendering scenes as long as they follow all above steps and do them correctly.

It seems game developers have already been familiar with imposter technique, but there are still some problems and inefficiency of this method. First of all, even though this method has been used for many years, game developers still need to study a bunch of API functions for generating and using imposters. Second of all, it is not easy to figure out and fix the bugs in the series of API function calls, because it usually gives you wired scenes when you do not do it right. Third of all, in the software level, it is impossible to know the width and height values of the imposter texture so game developers may just guesstimate a number. To ensure to allocate an enough space for an imposter, game developers cannot give a tight number. So, there should be some wasteful spaces been created. Last but not least, in terms of pixelated test, since game developers do not have ability to know the width of imposters, they need to calculate angles instead of only comparing current width with last one for detecting the pixelation.

In this thesis, we propose a hardware method to automatically generate imposters, detect the imposter errors and apply the imposter on a quad or two triangles. We take all the advantages of the software level method such as generating and using imposters on the fly, detecting impostor errors to avoid using impostors from those situations, and acquiring a significant improvement on scene rendering. Moreover, the game developers do not need to micro manage the impostor technique but take all the benefits from the technique. The whole details and implementations are hidden by the hardware. That saves game developers a lot of time to survey the API functions and write and debug their program to make the technique work. Since we implement the imposter technique in the hardware level, we can get the useful information that game developers cannot acquire to save some memory spaces and do more efficient imposter error testing. Besides, the memory spaces in embedded system are very constrained, so we not only adopt RGB565 image format but also do a run-length encoding based image compression to shrink the size of the impostor texture. In addition, the game developers probably could not be doing imposters on the times that the object we discovered, because we detect for all objects on the scene while game developers only pick the objects they notice.

This thesis proposes several hardware modifications to make the impostors technique automated and they are based on an existing hardware. The existing hardware is developed by a collaborative group at our university. The group is dedicated to developing a fully 3D graphics system for embedded system that includes the API, GPU hardware and benchmarks. It gives us an entire view of a 3D system and we focus on GPU hardware especially. In the hardware point of view, to increase the flexibility in rendering an image, many modern graphics cards have two shaders, vertex shader and pixel shader. The vertex shader is in charge of putting an object on the right place, while the pixel shader is responsible for coloring an object. Reasonably, our GPU has two shaders but the stable version of our GPU is a single core vertex shader and single core pixel shader while modern GPUs have at most eight cores. The advantage is it is easier for us to explain the modifications we propose for hardware based on the relatively simple GPU. But it must be mentioned that this method should not be only worked for this specific hardware, it can actually apply to any modern GPUs. This method is now not implemented in a real embedded system GPU, but the collaborative project gives us an opportunity to implement the method in it in the future.

There are some situations that the impostor should not be used. For example, two objects interact each other and other special effects. Imaging that when we handshake with others, the two hands interact each other. Because the impostor is a piece of paper, it only holds one depth value. Therefore, when objects are overlapping, the scene would be mess up. To solve this problem, we create an extended API function for game developers to turn on the impostor mechanism of an object. If game developers notice an object is not a good candidate to use the impostor, they just do not turn on the technique for an object.

To measure the expected benefits we can get from this method, we use real world benchmark called GLBenchmark. This benchmark is a popular benchmark to test the performance of the embedded system. There are advantages to choose this benchmark: firstly, this benchmark has many kinds of camera motions, for instance, directly go straight, rotate, spin and strafe. It allows us to test whether our impostor testing is operative or not. Secondly, there are many objects in the scene. It gives us opportunities to see could our method really catch the period of time to create the impostor textures and using these impostor textures. Thirdly, because it is a real world benchmark, that makes our method more robust.

The remainder of this thesis will mention several related works. Since GPUs are not powerful in early years, many game developers and researchers were trying to figure out some methods to speed up rendering scenes. Then we will describe the details of our method such as the how to make the impostor technique automated, how to detect the impostor errors and how to deal with multiple objects at same time. After that, comes the rules we follow to measure the expected benefits we could acquire and the result of how many expected benefits that we could get from this method.

2. Previous Work

2.1 Billboarding

The definition of the billboarding technique is describe in [1] as "Orienting a textured polygon based on the view direction is called billboarding, and the polygon is called a billboard". The two common kinds of billboard are screen-aligned billboard and world-oriented billboard. Figure 2.1 (from [1]) shows the major different of these two billboards.

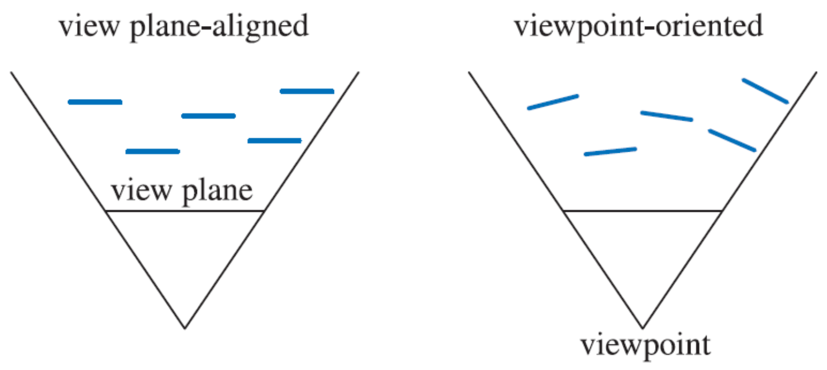


Figure 2.1 A top view of the two billboard alignment techniques. The five billboards face differently, depending on the method. (Figure references from [1].)

There are few ways to generate a billboard such as a picture of an actual object, a pre-rendered image that was done offline, an artist-drawn object, and so on. And the billboarding technique allows us to apply a billboards on a quad.

2.2 Impostors

2.2.1 What is impostor

The definition of impostor is also described in [1] as “An impostor is a billboard that is created on the fly by rendering a complex object from the current viewpoint into an image texture, which is mapped onto the billboard. Figure 2.2 from [1] shows how is an impostor created. An impostor is actually the image of the object that is sent to the frame buffer. The two parts in that image are opaque part and transparent part. The pixels been touched by the object are opaque, otherwise are transparent. Impostors are created at the run-time and billboarded onto a quad or two triangles to create illusion of the real object. Recording that figure 2.1 shows two kinds of billboards. The impostor is belong to the viewport-oriented billboard because the impostor must distort like the object does.

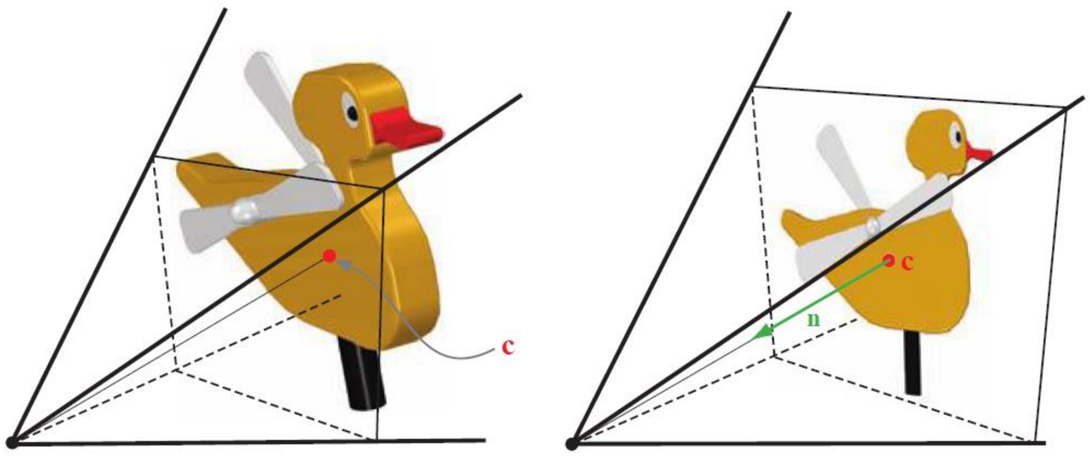


Figure 2.2 At the left, an impostor is created of the object viewed from the side by the viewing frustum. The view direction is toward the center, c, of the object, and an image is rendered and used as an impostor texture. This is shown on the right, where the texture is applied to a quadrilateral. The center of the impostor is equal to the center of the object, and the normal (emanating from the center) points directly toward the viewpoint. *(Figure references from [1].)*

2.2.2 The motivation of using impostors

Although the performance of modern GPUs are dramatically increased, it is still a challenge to keep the frame rate (at least twenty frames per second) because the objects in the game and the effects of these objects or the environment are much more complex. In order to speed up the rendering, the game developers do many tricks. The impostor technique is one of the tricks. In a 3D scene, everything is an approximation of the real object. Not only because the objects are rendering into a 2D screen, but also the 3D models are the approximations in their own. An impostor is an approximation of an object. It has only two triangles (6 vertices) while an object usually has thousands of vertices. So, the motivation of using the impostor technique is to use two triangles to replace thousands of triangles to speed up the rendering and still remain the quality of the scene.

2.2.3 The software implementations of impostors

Normally, in 3D rendering, a finished rendering pixel is sent to frame buffer to be displayed on the screen. In order to use impostor technique, we must create an impostor for the object. This section shows how game developers create an impostor and then how to apply the impostor on a quad or two triangles. Moreover, we also mention that how to apply this technique to multiple objects in last paragraph in the last paragraph.

2.2.3.1 Creating an impostor for an object

Figure 2.3 shows the code that we reference the frame buffer object (FBO) in openGL tutorial on the Internet [2] and implement it ourselves. Although the tutorial on the Internet provides the code for game developers, nonetheless, it is not just easily copying it and pasting it into our code and there are details to take care: firstly, both a new frame buffer and a new depth buffer need to be created. Of course, we must remember to create the new frame buffer, because we are trying to use it. But someone might forget to create the new depth buffer. A black quad would be displayed as the depth buffer was not created. Secondly, to indicate a texture to be the new frame buffer and the other texture to be the depth buffer, we have to change the current binding frame buffer and texture and then turn them back. Making sure to switch correctly, otherwise the scene would be messed up. Thirdly, the parameters should be given properly.

Now, we have two textures, one is used as frame buffer and the other is used as depth buffer. Then we have to draw the object into the texture to get the impostor. Figure 2.4 shows how to draw an object into the texture. After that, we would have the impostor of an object.

```
GLint curTextureID;
GLint curFBO;
glGetIntegerv(GL_TEXTURE_BINDING_2D, &curTextureID);
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &curFBO);
glGenFramebuffers(1, &obj.myFBO);
glBindFramebuffer(GL_FRAMEBUFFER, obj.myFBO);
glGenTextures(1, &obj.texColorBuffer);
glBindTexture(GL_TEXTURE_2D, obj.texColorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, WWW, HHH, 0, GL_RGBA,
                GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                        GL_TEXTURE_2D, obj.texColorBuffer, 0);
glGenTextures(1, &obj.depthBuffer);
glBindTexture(GL_TEXTURE_2D, obj.depthBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, WWW, HHH, 0,
                GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                        GL_TEXTURE_2D, obj.depthBuffer, 0);
glBindTexture(GL_TEXTURE_2D, curTextureID);
glBindFramebuffer(GL_FRAMEBUFFER, curFBO);
```

Figure 2.3 Codes to create a new rendering target.

```
glBindFramebuffer(GL_FRAMEBUFFER, myFBO);
glClear(GL_DEPTH_BUFFER_BIT);
Drawing the object
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Figure 2.4 Codes to render the object into the new rendering target which is the impostor billboard.

2.2.3.2 Applying an impostor on two triangles

Figure 2.5 shows the code that we reference the texturing in openGL tutorial on the Internet and implement it ourselves. Although the website provides the source code to apply the impostor on a quad, it is still not easy to do it right. To achieve this, we need to follow several steps: first, writing shader codes to tell GPUs how to render the impostor. Fortunately, because the impostor must be transformed like the object and the model view projection matrix has been set for the object, we just use it for vertex shader. For fragment shader, it is also easy, since we just try to apply a texture on the quad. Second, we have to indicate the texture coordinate. If we do not do it right, then we might see a black quad on the screen. Third, sending the six vertices of the quad to GPUs. If we do not do it correctly, then we might even not see the quad be displayed on the screen. Fourth, here comes a series of API functions to compile the shader code and link them to tell GPUs what the current program it is. If there are bugs in the series of code, then it might crashes the system. So, making sure to call every functions right. Fifth, because some pixels in the impostor are transparent, remembering to not update the depth buffer and color buffer as the pixel is transparent. Otherwise, the transparent pixels would blend with the pixels that has already been drawn. That is not what we want, because we want to only draw the opaque part of the texture to the screen.

```
const GLchar* vertexSource =
"#version 150 core\n"
"in vec3 position;"
"in vec2 texCoord;"
"out vec2 TexCoord;"
"void main() {"
"   TexCoord = texCoord;"
"   gl_Position = vec4(position, 1.0);"
"}";
const GLchar* fragmentSource =
"#version 150 core\n"
"in vec2 TexCoord;"
"out vec4 outColor;"
"uniform sampler2D texFramebuffer;"
"void main() {"
"   outColor = texture2D(texFramebuffer, TexCoord);"
"}";
GLuint myvbo;
GLint curvbo;
glGetIntegerv(GL_VERTEX_ARRAY_BINDING, &curvbo);
glGenBuffers(1, &myvbo);
obj.p2_min_value.x < 0 ? tminX = (1 - abs(obj.p2_min_value.x)) / 2.0 :
                          tminX = (1 + abs(obj.p2_min_value.x)) / 2.0;
obj.p2_max_value.x < 0 ? tmaxX = (1 - abs(obj.p2_max_value.x)) / 2.0 :
                          tmaxX = (1 + abs(obj.p2  max  value.x)) / 2.0;
obj.p2_min_value.y < 0 ? tminY = (1 - abs(obj.p2_min_value.y)) / 2.0 :
                          tminY = (1 + abs(obj.p2_min_value.y)) / 2.0;
obj.p2_max_value.y < 0 ? tmaxY = (1 - abs(obj.p2_max_value.y)) / 2.0 :
                          tmaxY = (1 + abs(obj.p2_max_value.y)) / 2.0;
GLfloat myvertices[] = { // 6 vertices of the two impostor triangles
obj.p3_min_value.x, obj.p3_min_value.y, obj.p3_min_value.z, tminX, tminY,
obj.p3_min_value.x, obj.p3_max_value.y, obj.p3_min_value.z, tminX, tmaxY,
obj.p3_max_value.x, obj.p3_max_value.y, obj.p3_min_value.z, tmaxX, tmaxY,
obj.p3_max_value.x, obj.p3_max_value.y, obj.p3_min_value.z, tmaxX, tmaxY,
obj.p3_max_value.x, obj.p3_min_value.y, obj.p3_min_value.z, tmaxX, tminY,
obj.p3_min_value.x, obj.p3_min_value.y, obj.p3_min_value.z, tminX, tminY
};
GLint status = GL_FALSE;
glBindBuffer(GL_ARRAY_BUFFER, myvbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(myvertices), myvertices, GL_STATIC_DRAW);
// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
glCompileShader(vertexShader);
// Create and compile the fragment shader
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
```

```
glCompileShader(fragmentShader);
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);
// Link the vertex and fragment shader into a shader program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
GLint current_program;
glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
glUseProgram(shaderProgram);
glUniform1i(glGetUniformLocation(shaderProgram, "texFramebuffer"), 0);
// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), 0);
GLint texAttrib = glGetAttribLocation(shaderProgram, "texCoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat),
                        (void*)(3 * sizeof(GLfloat)));
glViewport(0, 0, WWW, HHH);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
glEnable(GL_DEPTH_TEST);
glDisable(GL_CULL_FACE);
glAlphaFunc(GL_GREATER, 0);
glEnable(GL_ALPHA_TEST);
GLint curTextureID;
glGetIntegerv(GL_TEXTURE_BINDING_2D, &curTextureID);
glBindTexture(GL_TEXTURE_2D, obj.texColorBuffer);
glActiveTexture(GL_TEXTURE0);
glDrawArrays(GL_TRIANGLES, 0, 6);
glUseProgram(current_program);
glBindTexture(GL_TEXTURE_2D, curTextureID);
glBindBuffer(GL_ARRAY_BUFFER, curvbo);
```

Figure 2.5 Codes to render the impostor on a quad or two triangles.

2.3 Existing software approaches to using impostors

2.3.1 Dynamically Generated Impostors

This paper [3] describes some situations to re-generate the impostor of an object. The impostors are expected to be re-used as much as possible. If we over re-use the impostors to replace the object, then the game player might notice the fakes. And there are some situations described in [3] can ruin the illusions. They includes (1) when the resolution of texels (pixels in a texture) are lower than pixels on a screen, (2) when the camera translates and (3) when the camera gets to closer to impostors. The methods to detect these situations are also describe in [3].

Figure 2.6 from [3] shows the first situation. The paper describes that if $\alpha_{tex} > \alpha_{screen}$, then the texels in the texture could be distinguished by viewer. That causes the pixelation problem and it means the impostor need to be regenerated. In addition, the paper also describes that the distant impostor could still be re-used even though the test is failed. That is because the viewer may not notice the low resolution when the impostor is far from the camera.

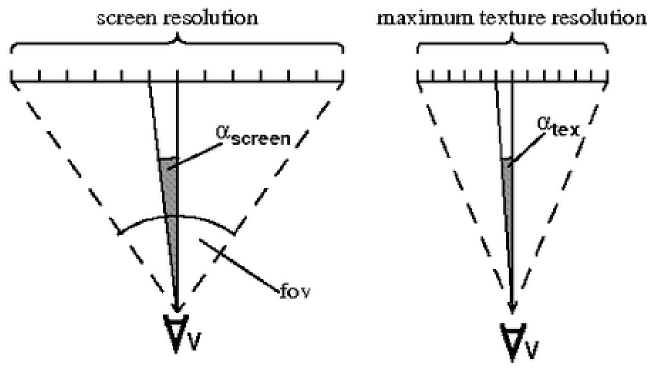


Figure 2.6 Pixel and texel viewing angles. *(Figure references from [3].)*

Figure 2.7 from [3] shows the second situation. When the camera translates from V1 to V2, we can see more left side of the object but see less of the right side. Seeing less causes no problems, but seeing more ruins the illusion. Because the impostor did not hold the pixels of the left side of the object. To prevent using impostor from this situation, the author suggested that if $\alpha_{trans} > \alpha_{screen}$, then the impostor should be re-generated.

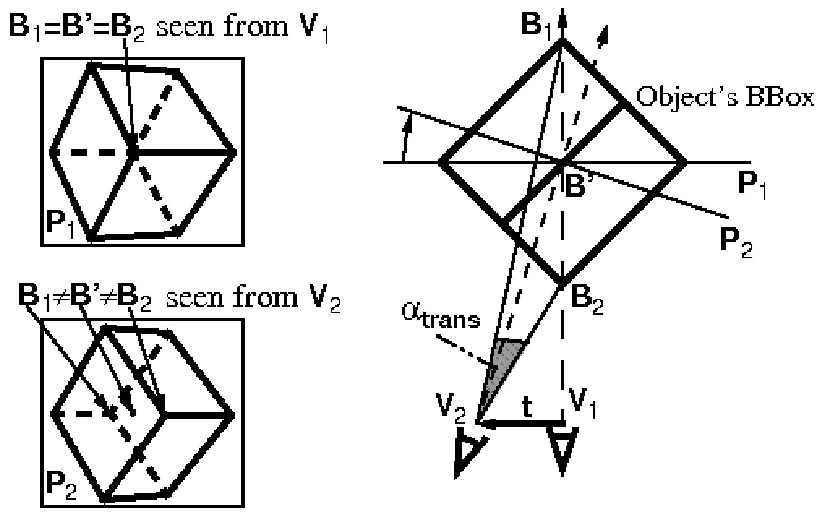


Figure 2.7 Error angle α_{trans} due to translation *(Figure references from [3].)*

Figure 2.8 from [3] shows the third situation. When the camera gets too closer to an object, it causes the pixelation problem. To avoid using impostor from this situation, the author suggested that if $\alpha_{size} > \alpha_{screen}$, then the impostor should be re-generated.

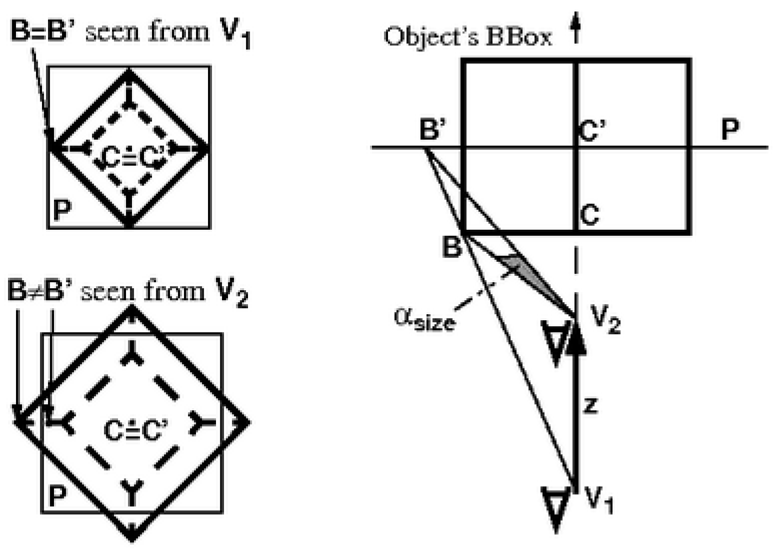


Figure 2.8 Error angle α_{size} due to move-in. *(Figure references from [3].)*

This paper gives situations that the impostor needs to be re-generated. We take all the observations of this paper, but we observe an extra situation that needs to be detected and we also propose different methods to detect the impostor errors. Considering that an object rotates like the earth, the rotation would mess up the pixels in the object. Since this situation might not happen in the benchmark that this paper uses, this paper does not need to consider that. But, in the GLBenchmark, there are times that objects rotate. So, we propose a method in chapter 3 to detect this situation. Moreover, because we have the information (extreme values and indices of the object) that is not acquirable in the software level, we can detect the impostor error by other ways. The methods will be described in chapter 3 as well.

2.3.2 The use of Impostors in Interactive 3D Graphics System

The motivation of this paper is to use impostors to speed up the 3D interactive rendering. The processes of creating and using impostors are similar to the code shown in section 2.2.3.1 and 2.2.3.2. This paper references the paper presented in 2.3.1 a lot. In this paper, it provides similar but easier tests to detect the changes of an object between frames and it also mentions the thresholds of the impostor errors. “In our implementation we decided to generate new imposters when the angle had changed more than 5% (18 degrees) in vertical or horizontal orientation.” mentioned by paper [4]. And they also mentioned: “In our implementation we simple chose to show the imposter when the distance to the object from the camera exceted a fixed number.”

Since this paper only show the impostor when the object is farther than a fixed distance, it might misses some opportunities to get the advantages of the impostors. The reason of their impostors only can display distant impostors is they create a fixed size of impostor. The player would notice the pixelation, as the object is close to the player. So, in order to avoid the pixelation problem, they only re-use the impostors when the object is distant.

In our method, we do not have this constraint, because we hold the width and height of the object. They help us to create a fit size of impostor for the object, so that we can use the impostor as the object is close to the player. But, we do need to avoid the pixelation problem when the player gets closer and closer to the object.

2.3.3 A GPU hardware-based method for automatic occlusion detection and optimization for objects and subobjects

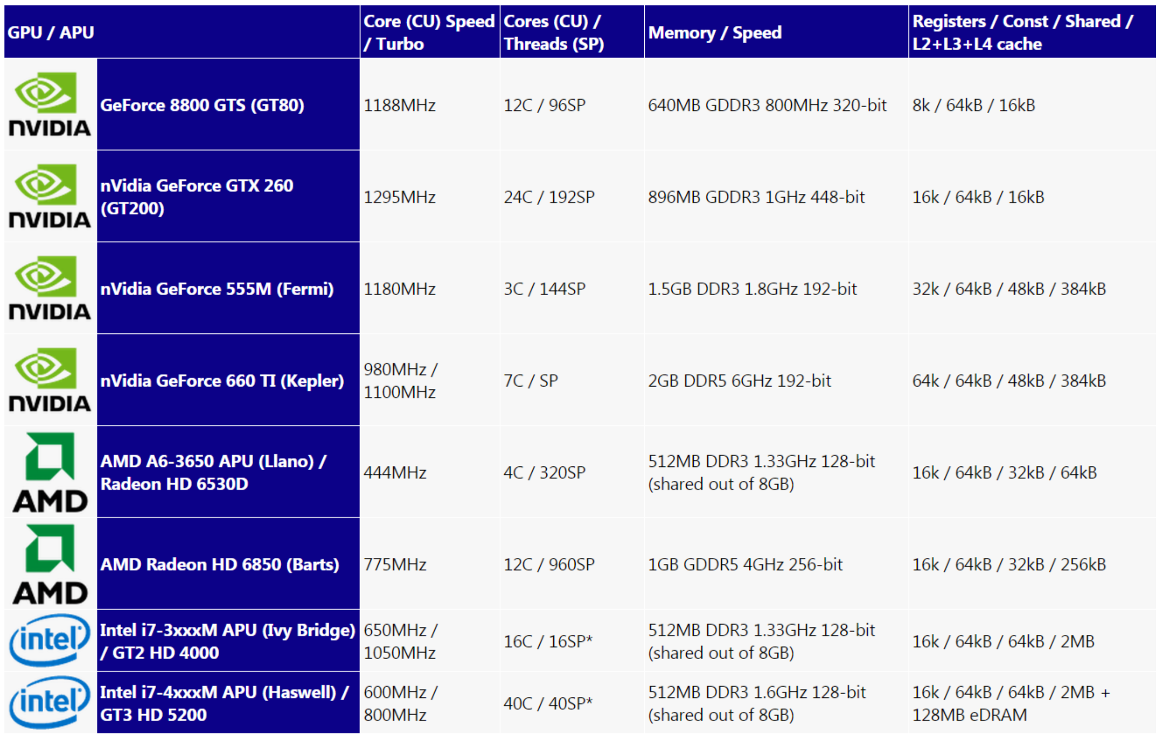
This paper [5] proposes an automated hardware optimization. Game developers figure out ways to speed up the 3D rendering. One of the common methods is occlusion detection. Considering that there are some objects in the scene are fully occluded by other objects. Since the players cannot see these objects, we do not need to draw them at all. The problem is how to know an object is fully occluded and we do not need to draw it. The answer is bounding box which only has 12 triangles and totally wraps the object. The idea is since it is easy enough to draw a bounding box and the bounding box wraps the object, we can utilize the bounding box of the object to do the occlusion test. If the bounding box is fully occluded, then the object is fully occluded, too. Moreover, they propose a method to create subobjects to increase the opportunities of full occlusion of an object.

The paper proposes both software and hardware modifications to make the occlusion detection automated. And the existing games do not need to be modified to acquire benefits from using their method. For us, we also propose both software and hardware modifications to make the existing software method automatically be supported by the hardware and the game needs to do no changes to get benefits from out method.

2.4 The issues of textures

2.4.1 The limitation of the memory space in the embedded system

Figure 2.9 from [6] shows the size of memory space of each GPUs. In the picture, we can see that the memory space in embedded system is really constrain, so we do propose a compression method to reduce the size of IBTs.



GPU / APU		Core (CU) Speed / Turbo	Cores (CU) / Threads (SP)	Memory / Speed	Registers / Const / Shared / L2+L3+L4 cache
NVIDIA	GeForce 8800 GTS (GT80)	1188MHz	12C / 96SP	640MB GDDR3 800MHz 320-bit	8k / 64kB / 16kB
NVIDIA	nVidia GeForce GTX 260 (GT200)	1295MHz	24C / 192SP	896MB GDDR3 1GHz 448-bit	16k / 64kB / 16kB
NVIDIA	nVidia GeForce 555M (Fermi)	1180MHz	3C / 144SP	1.5GB DDR3 1.8GHz 192-bit	32k / 64kB / 48kB / 384kB
NVIDIA	nVidia GeForce 660 TI (Kepler)	980MHz / 1100MHz	7C / SP	2GB DDR5 6GHz 192-bit	64k / 64kB / 48kB / 384kB
AMD	AMD A6-3650 APU (Llano) / Radeon HD 6530D	444MHz	4C / 320SP	512MB DDR3 1.33GHz 128-bit (shared out of 8GB)	16k / 64kB / 32kB / 64kB
AMD	AMD Radeon HD 6850 (Barts)	775MHz	12C / 960SP	1GB GDDR5 4GHz 256-bit	16k / 64kB / 32kB / 256kB
intel	Intel i7-3xxxM APU (Ivy Bridge) / GT2 HD 4000	650MHz / 1050MHz	16C / 16SP*	512MB DDR3 1.33GHz 128-bit (shared out of 8GB)	16k / 64kB / 64kB / 2MB
intel	Intel i7-4xxxM APU (Haswell) / GT3 HD 5200	600MHz / 800MHz	40C / 40SP*	512MB DDR3 1.6GHz 128-bit (shared out of 8GB)	16k / 64kB / 64kB / 2MB + 128MB eDRAM

Figure 2.9 The memory size of different types of GPUs. *(Figure references from [6].)*

2.4.2 The 565 texture format

565 file format [7] is one of the texture format. It takes 5 bits to present red and blue, but 6 bits to present green. We have transformed an 888 file format which uses 8 bits to present red, green and blue to a 565 file format. It is hard to distinguish the differences between those two file formats. But, if we further try 454, then the differences become noticeable. Because the memory space is quite restricted in embedded system, we use this format to store the impostor.

2.4.3 Run-length encoding

Run-length encoding [8] is a simple and fast encoding technique. The compression ratio of this encoding method depends on the variety of the image. An image is composed of a lot of pixels and each pixel is composed of colors of red, green and blue. If we open an image file such as a bitmap file as a texture file, then there are values in it. In RGB565 texture format, the range of red and blue is from 0 to 31 and the range of green is from 0 to 63. Figure 2.10 shows how the run-length encoding works. The repeating characters (aaaa) is called a *run*. After encoding, the run becomes two bytes, run count and run value.

```
aaaaabbbbbbzzdww  =>  5a7b2z1d2w
```

Figure 2.10 How run-length encoding compresses the data. The input is a character string and the run-length encoding reduce the redundancy and create a number to present the repeat times.

According to section 2.4.1, we get really a limited size of cache to use so we need an image compression technique to compress the texture. There are reasons for us to pick this encoding method, firstly, it is quick to execute. This is important for us to encoding and decoding an image, because we need to pay the cost each time we use the impostor. If we take a long time to compress and de-compress the impostors, then we get a few improvements from our method. Secondly, since there is only one object in the impostor and a part of the pixels are transparent, these properties of the impostors let the run-length encoding method gives us better compression ratio.

3. Methodology

The impostor technique has been used to increase the speed of object rendering and the game developers acquire really huge improvement from it, however, the process of making it works is complicated and time-wasteful. Therefore, we propose an automatic impostor technique to get all the improvements from the software technique but hide the complex implementations and the painful debugging processes. To achieve the automation, we propose several hardware modifications. The major changes are shown in figure 3.1 and figure 3.2. These two figures need to be read together, since they cooperate with each other to manage the impostor technique for each object. In the following paragraphs, we will first describe the whole idea of these two figures and further explain the details individually.

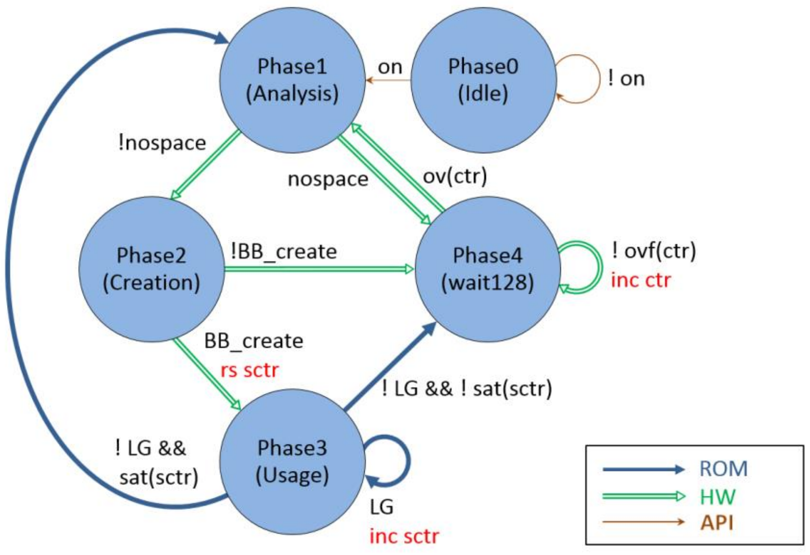


Figure 3.1 This figure shows the state machine to achieve creating and using impostors on the fly. It also shows how we can automatically manage the impostor technique. A view of four spheres, with a wide field of view. The upper left is a billboard texture of a sphere, using view plane alignment. The upper right billboard is viewpoint oriented. The lower row shows two real spheres.

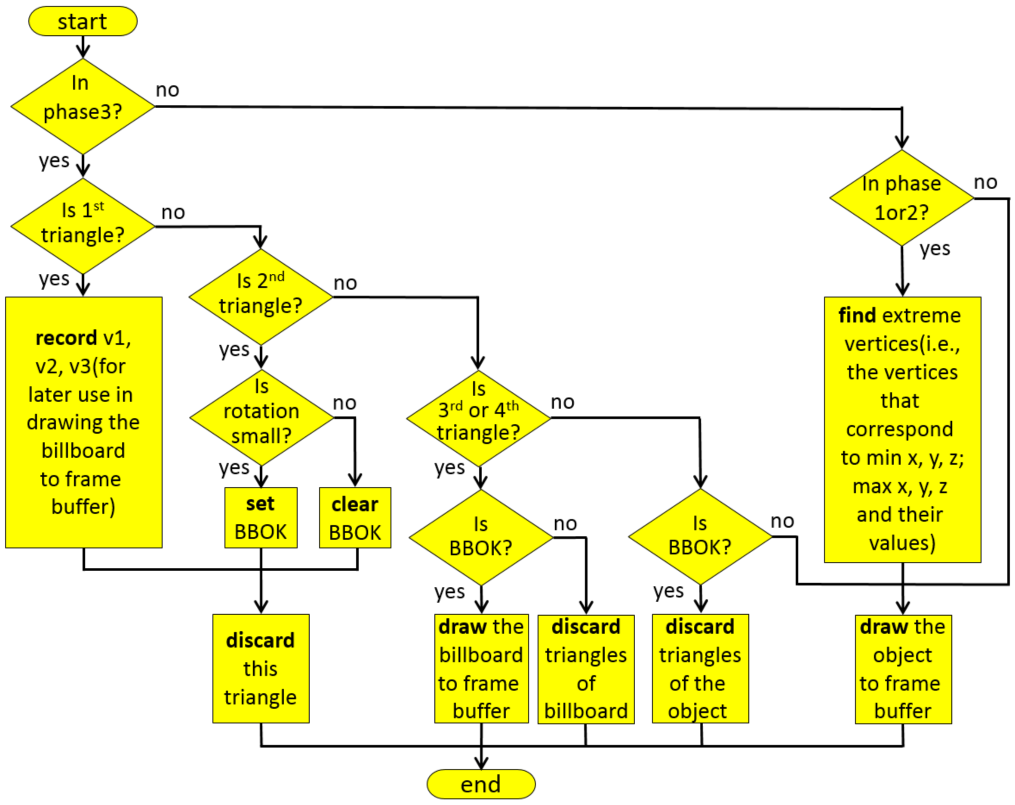

start
In phase3?
no
yes
In phase 1or2?
no
yes
Is 1st triangle?
no
yes
Is 2nd triangle?
no
yes
record v1, v2, v3(for later use in drawing the billboard to frame buffer)
find extreme vertices(i.e., the vertices that correspond to min x, y, z; max x, y, z and their values)
Is rotation small?
no
yes
Is 3rd or 4th triangle?
no
yes
set BBOK
clear BBOK
Is BBOK?
no
yes
Is BBOK?
no
yes
discard this triangle
draw the billboard to frame buffer
discard triangles of billboard
discard triangles of the object
draw the object to frame buffer
end

Figure 3.2 This figure shows the state machine to achieve creating and using impostors on the fly. It also shows how we can automatically manage the impostor technique.

3.1 The state machine and ROM code

There are four basic actions in the existing impostor technique: creating impostor, impostor error testing and using impostor. Once game developers try to apply the technique to an object, firstly, they need to call a series of API functions to create an impostor. Before using the impostor, impostor error testing should be performed to see whether the impostor can be used or not. If the testing passes, then game developers have to decide where to put the impostor and apply the impostor on a quad or two triangles. We basically follow these actions to design our state machine and the ROM code, but we also provide some changes to make this technique more efficient. And here comes the whole ideas:

3.1.1 Phase 1 – the analysis phase

This phase is not included in software technique, but there are reasons to involve this phase. First of all, the memory space is so limited in embedded system. So, in order to create a fit size of memory space for an object to cache its impostor, we create this phase to get the extreme values. Second, the existing method does an angle testing to detect the pixelation problem. It may not be an efficient way to detect, because it has a heavy computation to get the angle. For us, since we hold the extreme values, we can directly compare the current width of the object with the width of the impostor to see whether the pixelation problem happens or not.

The state machine begins with phase 1. There are three actions in phase 1: (1) identifying an object (2) gathering the extreme values and indices (3) store the results to the memory and transit to next phase.

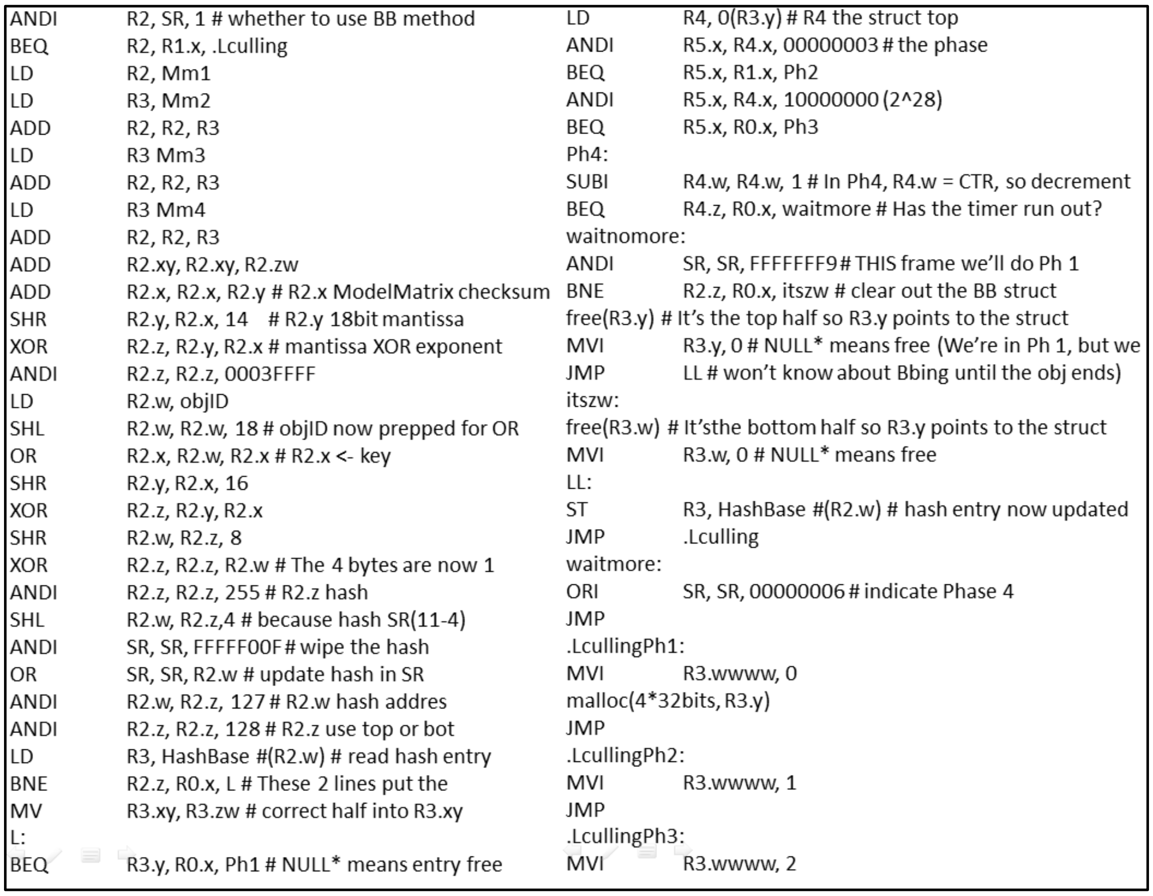
Figure 3.3 shows the pseudo code to identify an object at the beginning of an object. The purpose of the action 1 is to manage multi-objects. So, when the first triangle of an object comes to ROM code, we give it a hash key. This hash key is generated by merging object id and the sum of elements of model matrix. We put the sum of elements of model matrix into the key, because we observed that some objects are drawn many times such as the pillars in the benchmark. The sum of elements of model matrix is a sort of secondary key to help us to distinguish those multi-drawn objects. After giving the hash key to an object, we allocate a size of 24 32-bits memory space for an object. Some information is put in this memory space include current phase, saturated counter, the pointer that point to the head of the space to store the impostor, the pointer that point to the head of the bit vector, the extreme values and their indices and the width and height of the impostor. Then we finish the action 1.

In action 2, we are going to gather the extreme values and their indices. After the processes of vertex shader, the triangles of an object go into the ROM code to perform the culling, clipping and rasterizing triangle by triangle. Figure 3.4 represents the pseudo code that we add to find the extreme values and the indices of them. In the pseudo code, we follow the rules of the assembly code specification of the collaborative project. Since almost every GPUs has its own assembly code specification, it is impossible for us to provide the codes for each GPU. So we will just explain the concept of finding the extreme values. The idea is all of the triangles in an object go into ROM code triangle by triangle. Each time a triangle comes, we find the temporary minimum and maximum values and their indices of it and then compare the temporary one with the final one until all triangles walk through the ROM code. Beside, we also need to find the x and y values of the minimum and maximum z.

Actually, we could propose a dedicated piece of hardware to gather the extreme values and indices of them but it may not worth to create a dedicated hardware only for finding extreme values purpose and it is just a little change to the ROM code.

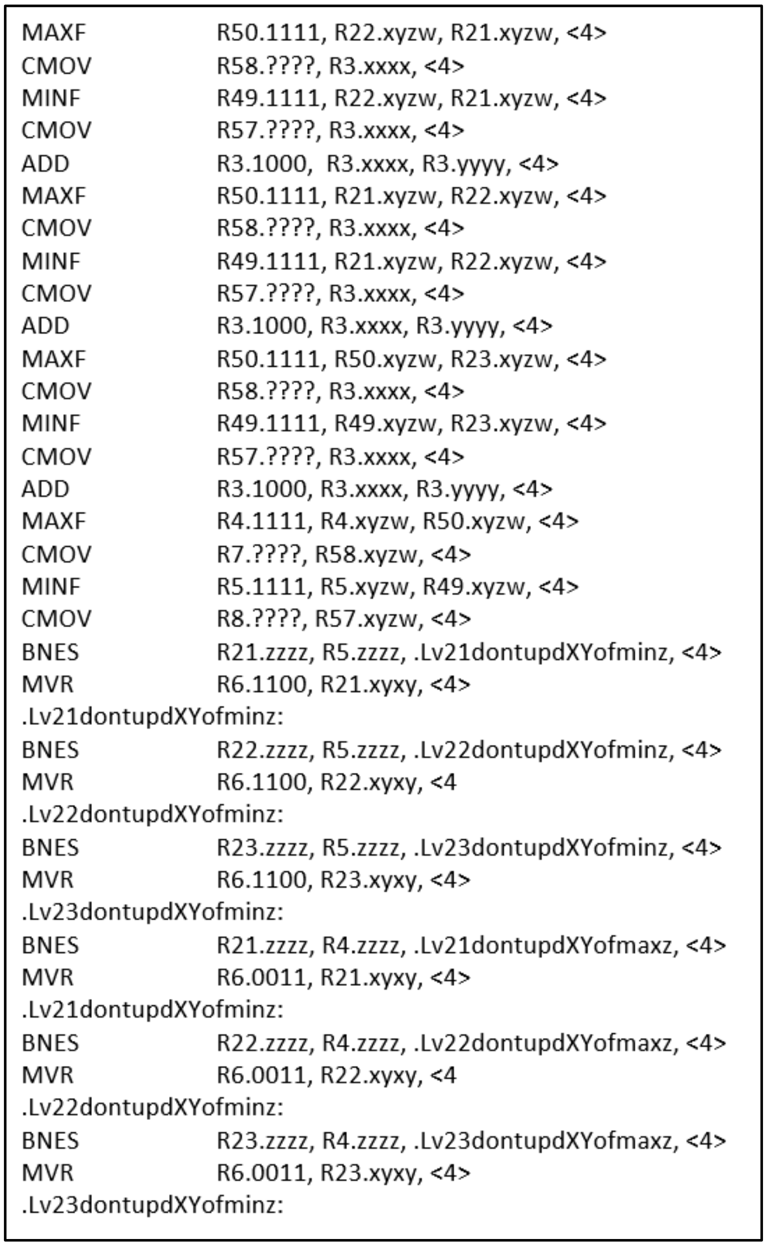
Figure 3.5 shows the pseudo code of the processes at the end of an object. At the end of the phase 1, we need to post process the data which be gathered from action 2. First, we need to re-identify the object, because we do not lock the registers which hold the hash key. Second, since we are in phase 1, we are going to transit to either phase 2 or phase 4.

(See figure 3.1.) The decision is made by whether there is no enough memory spaces are available. If the situation occurs, we go to phase 4, otherwise go to phase 2. If we go to phase 4, then we initialize the counter and done, otherwise, we should subtract minimum x from maximum x to get the width of the object, subtract minimum y from maximum y to get the height of the object and store the width, height, phase, the indices of extreme values to the memory.



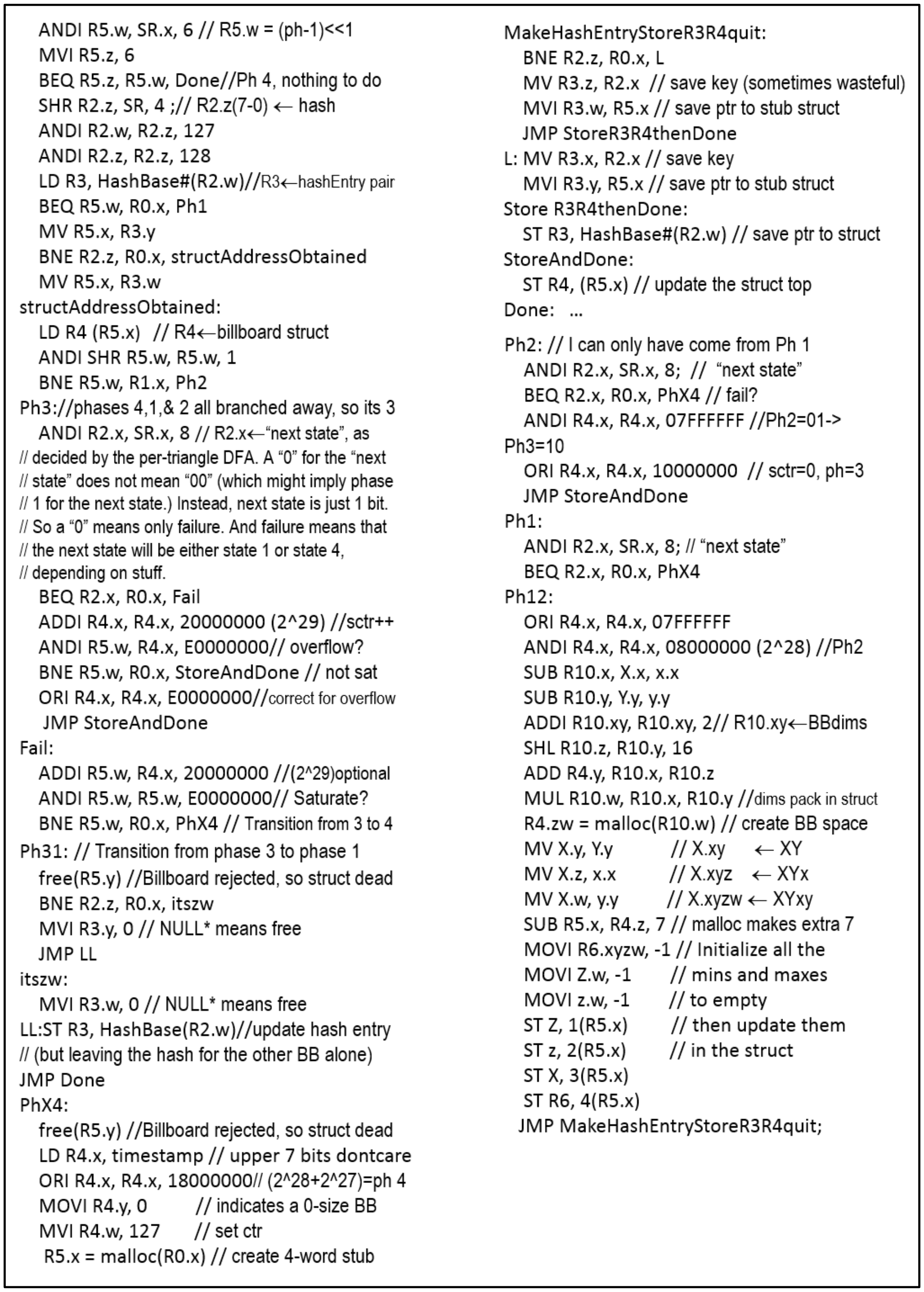
```
ANDI        R2, SR, 1 # whether to use BB method
BEQ         R2, R1.x, .Lculling
LD          R2, Mm1
LD          R3, Mm2
ADD         R2, R2, R3
LD          R3 Mm3
ADD         R2, R2, R3
LD          R3 Mm4
ADD         R2, R2, R3
ADD         R2.xy, R2.xy, R2.zw
ADD         R2.x, R2.x, R2.y # R2.x ModelMatrix checksum
SHR         R2.y, R2.x, 14    # R2.y 18bit mantissa
XOR         R2.z, R2.y, R2.x # mantissa XOR exponent
ANDI        R2.z, R2.z, 0003FFFF
LD          R2.w, objID
SHL         R2.w, R2.w, 18 # objID now prepped for OR
OR          R2.x, R2.w, R2.x # R2.x <- key
SHR         R2.y, R2.x, 16
XOR         R2.z, R2.y, R2.x
SHR         R2.w, R2.z, 8
XOR         R2.z, R2.z, R2.w # The 4 bytes are now 1
ANDI        R2.z, R2.z, 255 # R2.z hash
SHL         R2.w, R2.z,4 # because hash SR(11-4)
ANDI        SR, SR, FFFFF00F # wipe the hash
OR          SR, SR, R2.w # update hash in SR
ANDI        R2.w, R2.z, 127 # R2.w hash addres
ANDI        R2.z, R2.z, 128 # R2.z use top or bot
LD          R3, HashBase #(R2.w) # read hash entry
BNE         R2.z, R0.x, L # These 2 lines put the
MV          R3.xy, R3.zw # correct half into R3.xy
L:
BEQ         R3.y, R0.x, Ph1 # NULL* means entry free
LD          R4, 0(R3.y) # R4 the struct top
ANDI        R5.x, R4.x, 00000003 # the phase
BEQ         R5.x, R1.x, Ph2
ANDI        R5.x, R4.x, 10000000 (2^28)
BEQ         R5.x, R0.x, Ph3
Ph4:
SUBI        R4.w, R4.w, 1 # In Ph4, R4.w = CTR, so decrement
BEQ         R4.z, R0.x, waitmore # Has the timer run out?
waitnomore:
ANDI        SR, SR, FFFFFFF9 # THIS frame we'll do Ph 1
BNE         R2.z, R0.x, itszw # clear out the BB struct
free(R3.y) # It's the top half so R3.y points to the struct
MVI         R3.y, 0 # NULL* means free (We're in Ph 1, but we
JMP         LL # won't know about Bbing until the obj ends)
itszw:
free(R3.w)  # It'sthe bottom half so R3.y points to the struct
MVI         R3.w, 0 # NULL* means free
LL:
ST          R3, HashBase #(R2.w) # hash entry now updated
JMP         .Lculling
waitmore:
ORI         SR, SR, 00000006 # indicate Phase 4
JMP
.LcullingPh1:
MVI         R3.wwww, 0
malloc(4*32bits, R3.y)
JMP
.LcullingPh2:
MVI         R3.wwww, 1
JMP
.LcullingPh3:
MVI         R3.wwww, 2
```

Figure 3.3 The pseudo code to give each object an individual identification. This code is used the instruction set of the collaborated project.



```
MAXF        R50.1111, R22.xyzw, R21.xyzw, <4>
CMOV        R58.????, R3.xxxx, <4>
MINF        R49.1111, R22.xyzw, R21.xyzw, <4>
CMOV        R57.????, R3.xxxx, <4>
ADD         R3.1000,  R3.xxxx, R3.yyyy, <4>
MAXF        R50.1111, R21.xyzw, R22.xyzw, <4>
CMOV        R58.????, R3.xxxx, <4>
MINF        R49.1111, R21.xyzw, R22.xyzw, <4>
CMOV        R57.????, R3.xxxx, <4>
ADD         R3.1000, R3.xxxx, R3.yyyy, <4>
MAXF        R50.1111, R50.xyzw, R23.xyzw, <4>
CMOV        R58.????, R3.xxxx, <4>
MINF        R49.1111, R49.xyzw, R23.xyzw, <4>
CMOV        R57.????, R3.xxxx, <4>
ADD         R3.1000, R3.xxxx, R3.yyyy, <4>
MAXF        R4.1111, R4.xyzw, R50.xyzw, <4>
CMOV        R7.????, R58.xyzw, <4>
MINF        R5.1111, R5.xyzw, R49.xyzw, <4>
CMOV        R8.????, R57.xyzw, <4>
BNES        R21.zzzz, R5.zzzz, .Lv21dontupdXYofminz, <4>
MVR         R6.1100, R21.xyxy, <4>
.Lv21dontupdXYofminz:
BNES        R22.zzzz, R5.zzzz, .Lv22dontupdXYofminz, <4>
MVR         R6.1100, R22.xyxy, <4
.Lv22dontupdXYofminz:
BNES        R23.zzzz, R5.zzzz, .Lv23dontupdXYofminz, <4>
MVR         R6.1100, R23.xyxy, <4>
.Lv23dontupdXYofminz:
BNES        R21.zzzz, R4.zzzz, .Lv21dontupdXYofmaxz, <4>
MVR         R6.0011, R21.xyxy, <4>
.Lv21dontupdXYofminz:
BNES        R22.zzzz, R4.zzzz, .Lv22dontupdXYofmaxz, <4>
MVR         R6.0011, R22.xyxy, <4
.Lv22dontupdXYofminz:
BNES        R23.zzzz, R4.zzzz, .Lv23dontupdXYofmaxz, <4>
MVR         R6.0011, R23.xyxy, <4>
.Lv23dontupdXYofminz:
```

Figure 3.4 In this pseudo code, we have several assumptions: (1) the positions of three vertices of a triangle are put into register 21, 22 and 23. (2) The x component of register 3 holds the index counter and it is initialized only at the beginning of an object. (3) The register 49 and 50 hold the temporary minimum and maximum values while the register 5 and 4 hold the final minimum and maximum values. (4) The register 57 and 58 hold the temporary minimum and maximum indices while the register 8 and 7 hold the final minimum and maximum indices. (5) The x and y components of register 6 hold the x and y values of minimum z while the z and w components of register 6 hold the x and y values of maximum z.



```
   ANDI R5.w, SR.x, 6 // R5.w = (ph-1)<<1
   MVI R5.z, 6
   BEQ R5.z, R5.w, Done//Ph 4, nothing to do
   SHR R2.z, SR, 4 ;// R2.z(7-0) ← hash
   ANDI R2.w, R2.z, 127
   ANDI R2.z, R2.z, 128
   LD R3, HashBase#(R2.w)//R3←hashEntry pair
   BEQ R5.w, R0.x, Ph1
   MV R5.x, R3.y
   BNE R2.z, R0.x, structAddressObtained
   MV R5.x, R3.w
structAddressObtained:
   LD R4 (R5.x)  // R4←billboard struct
   ANDI SHR R5.w, R5.w, 1
   BNE R5.w, R1.x, Ph2
Ph3://phases 4,1,& 2 all branched away, so its 3
   ANDI R2.x, SR.x, 8 // R2.x←"next state", as
// decided by the per-triangle DFA. A "0" for the "next
// state" does not mean "00" (which might imply phase
// 1 for the next state.) Instead, next state is just 1 bit.
// So a "0" means only failure. And failure means that
// the next state will be either state 1 or state 4,
// depending on stuff.
   BEQ R2.x, R0.x, Fail
   ADDI R4.x, R4.x, 20000000 (2^29) //sctr++
   ANDI R5.w, R4.x, E0000000// overflow?
   BNE R5.w, R0.x, StoreAndDone // not sat
   ORI R4.x, R4.x, E0000000//correct for overflow
    JMP StoreAndDone
Fail:
   ADDI R5.w, R4.x, 20000000 //(2^29)optional
   ANDI R5.w, R5.w, E0000000// Saturate?
   BNE R5.w, R0.x, PhX4 // Transition from 3 to 4
Ph31: // Transition from phase 3 to phase 1
   free(R5.y) //Billboard rejected, so struct dead
   BNE R2.z, R0.x, itszw
   MVI R3.y, 0 // NULL* means free
   JMP LL
itszw:
   MVI R3.w, 0 // NULL* means free
LL:ST R3, HashBase(R2.w)//update hash entry
// (but leaving the hash for the other BB alone)
JMP Done
PhX4:
   free(R5.y) //Billboard rejected, so struct dead
   LD R4.x, timestamp // upper 7 bits dontcare
   ORI R4.x, R4.x, 18000000// (2^28+2^27)=ph 4
   MOVI R4.y, 0         // indicates a 0-size BB
   MVI R4.w, 127        // set ctr
    R5.x = malloc(R0.x) // create 4-word stub

MakeHashEntryStoreR3R4quit:
   BNE R2.z, R0.x, L
   MV R3.z, R2.x  // save key (sometimes wasteful)
   MVI R3.w, R5.x // save ptr to stub struct
   JMP StoreR3R4thenDone
L: MV R3.x, R2.x // save key
   MVI R3.y, R5.x // save ptr to stub struct
Store R3R4thenDone:
   ST R3, HashBase#(R2.w) // save ptr to struct
StoreAndDone:
   ST R4, (R5.x) // update the struct top
Done:  ...

Ph2: // I can only have come from Ph 1
   ANDI R2.x, SR.x, 8;  //  "next state"
   BEQ R2.x, R0.x, PhX4 // fail?
   ANDI R4.x, R4.x, 07FFFFFF //Ph2=01->
Ph3=10
   ORI R4.x, R4.x, 10000000  // sctr=0, ph=3
   JMP StoreAndDone
Ph1:
   ANDI R2.x, SR.x, 8; // "next state"
   BEQ R2.x, R0.x, PhX4
Ph12:
   ORI R4.x, R4.x, 07FFFFFF
   ANDI R4.x, R4.x, 08000000 (2^28) //Ph2
   SUB R10.x, X.x, x.x
   SUB R10.y, Y.y, y.y
   ADDI R10.xy, R10.xy, 2// R10.xy←BBdims
   SHL R10.z, R10.y, 16
   ADD R4.y, R10.x, R10.z
   MUL R10.w, R10.x, R10.y //dims pack in struct
   R4.zw = malloc(R10.w) // create BB space
   MV X.y, Y.y          // X.xy    ← XY
   MV X.z, x.x          // X.xyz   ← XYx
   MV X.w, y.y          // X.xyzw ← XYxy
   SUB R5.x, R4.z, 7 // malloc makes extra 7
   MOVI R6.xyzw, -1 // Initialize all the
   MOVI Z.w, -1       // mins and maxes
   MOVI z.w, -1       // to empty
   ST Z, 1(R5.x)      // then update them
   ST z, 2(R5.x)      // in the struct
   ST X, 3(R5.x)
   ST R6, 4(R5.x)
  JMP MakeHashEntryStoreR3R4quit;
```

Figure 3.5 The post process at the end of the rom code for an object.

3.1.2 Phase 2 – the creation phase

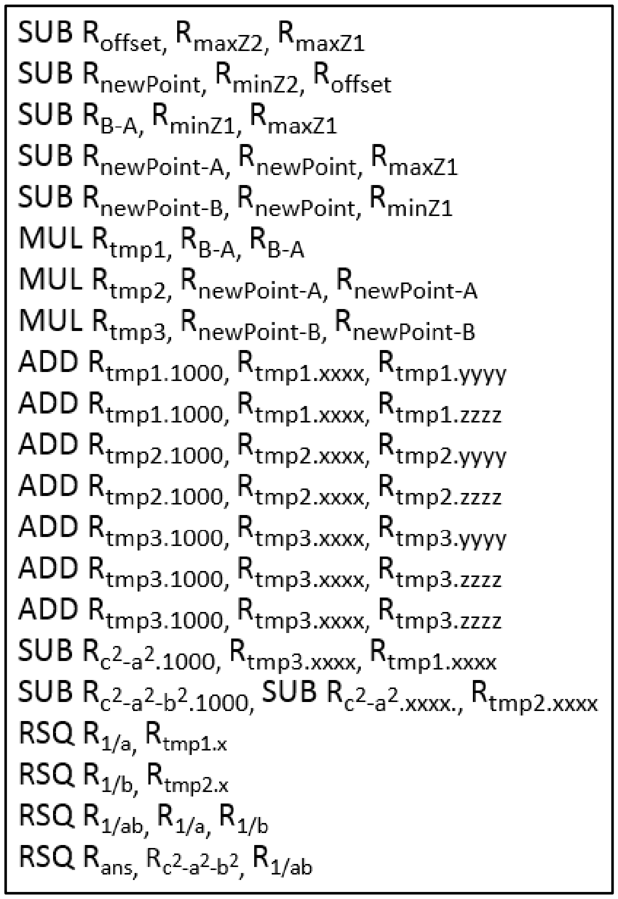
This is the second phase of the four called creation phase. The existing method has the similar function of this phase, because we must create the impostor for an object before we can use it. But, in our method, we do not create the impostor except the object passes the impostor error testing. The reason is because the memory space is so limited in embedded system, we must use it in an efficient way. That is why we test before to avoid creating the impostor for those objects have a strong probability to fail the test on next phase which is usage phase.

In this phase, we have to do 4 actions include (1) identifying the object (2) testing the impostor error (3) creating the impostor and bit-vector probably (4) updating the information of the object and transiting to the next phase. The action 1 of this phase and action 1 of phase 1 are the same.

In this method, we propose three kinds of impostor error tests include pixelation, rotation and delta. Before testing the errors, we make use of the indices of the extreme values to guesstimate the extreme values of this phase. The advantages of this are it allows us to create the impostor that never been used. And it reduces the whole actions that been used to gather the extreme values. The drawback is the guess could be wrong when the extreme points are different between two frames. But, it is acceptable, because the changes between two frames are slight enough to make this guess. To do this, we need the API to send us the points who hold the indices. In phase 1, we stored the indices in a specific location of memory so that the API can reference those indices and send us those points before starting to send the vertices of the object. After getting those points, we are going to describe all those tests.

First of all, the pixelation test, this error occurs when the player gets too closer to the object. We must avoid this, otherwise the player will complain the quality of the scene. To achieve this test, we wrote a pseudo code. The testing is simple, it just subtract the current width from the width from phase 1. If the difference is bigger or smaller than the threshold, then it fails the test. We suggest a conservative threshold 3. This number is decided after comparing lots of results of experiments. But, this is just our suggestions, it can be changed by someone who is willing to implement this method into hardware.

Second of all, the rotation test, imagining a ball with a side black and the other side white. When the ball rotates, the pixels on the ball are messed up by the rotation. Since we hold the x, y and z values of minimum and maximum z form last frame and we guesstimate those values by utilizing the indices for current phase, we can compute the angle between those two vectors. Figure 3.6 shows the idea. First, subtracting the maximum z from minimum z to get the two vectors. Second, translating one of the vector to let the two vectors have the same origin. Third, computing the length of each side of a triangle. Fourth, using cosine theorem to get the angle. In this method, we suggest a threshold 1. That means if the angle is bigger than the threshold, then it fails the test. Of course, this number is decided by our experiment, and can be changed by the developer.



```
SUB $R_{offset}$, $R_{maxZ2}$, $R_{maxZ1}$
SUB $R_{newPoint}$, $R_{minZ2}$, $R_{offset}$
SUB $R_{B-A}$, $R_{minZ1}$, $R_{maxZ1}$
SUB $R_{newPoint-A}$, $R_{newPoint}$, $R_{maxZ1}$
SUB $R_{newPoint-B}$, $R_{newPoint}$, $R_{minZ1}$
MUL $R_{tmp1}$, $R_{B-A}$, $R_{B-A}$
MUL $R_{tmp2}$, $R_{newPoint-A}$, $R_{newPoint-A}$
MUL $R_{tmp3}$, $R_{newPoint-B}$, $R_{newPoint-B}$
ADD $R_{tmp1.1000}$, $R_{tmp1.xxxx}$, $R_{tmp1.yyyy}$
ADD $R_{tmp1.1000}$, $R_{tmp1.xxxx}$, $R_{tmp1.zzzz}$
ADD $R_{tmp2.1000}$, $R_{tmp2.xxxx}$, $R_{tmp2.yyyy}$
ADD $R_{tmp2.1000}$, $R_{tmp2.xxxx}$, $R_{tmp2.zzzz}$
ADD $R_{tmp3.1000}$, $R_{tmp3.xxxx}$, $R_{tmp3.yyyy}$
ADD $R_{tmp3.1000}$, $R_{tmp3.xxxx}$, $R_{tmp3.zzzz}$
ADD $R_{tmp3.1000}$, $R_{tmp3.xxxx}$, $R_{tmp3.zzzz}$
SUB $R_{c^2-a^2.1000}$, $R_{tmp3.xxxx}$, $R_{tmp1.xxxx}$
SUB $R_{c^2-a^2-b^2.1000}$, SUB $R_{c^2-a^2.xxxx.}$, $R_{tmp2.xxxx}$
RSQ $R_{1/a}$, $R_{tmp1.x}$
RSQ $R_{1/b}$, $R_{tmp2.x}$
RSQ $R_{1/ab}$, $R_{1/a}$, $R_{1/b}$
RSQ $R_{ans}$, $R_{c^2-a^2-b^2}$, $R_{1/ab}$
```

Figure 3.6 The pseudo code to detect the rotation error.

Third of all, the delta test. Imagining that a person is looking at the nib of a pen. Once the pen is rotating a little, the person observe the huge change. But, if the person goes to the side of the pen and the pen rotates the same way, the person may not notice that. Rotation test can caught this error out with a very small threshold, but taking a too small threshold eliminates the lots of opportunities of using impostors. So, this test allows us to catch the first situation that be mentioned earlier. Figure 3.7 shows the idea.

```
SUB $R_{x5-x1}$, $R_{last\_minZ.xxxx}$, $R_{last\_minX.xxxx}$
SUB $R_{x2-x5}$, $R_{last\_maxX.xxxx}$, $R_{last\_minZ.xxxx}$
SBGTVF $R_{x5-x1}$, $R_{x2-x5}$, side1
MUL $R_{Dx \cdot c5}$, $R_{x2-x5.xxxx}$, $R_{cur\_minZ.zzzz}$
DIV $R_{Dx \cdot c5/z5}$, $R_{Dx \cdot c5.xxxx}$, $R_{last\_minZ.zzzz}$
side1:
SUB $R_{a5-a1}$, $R_{cur\_minZ.xxxx}$, $R_{cur\_minX.xxxx}$
SUB $R_{\Delta x-(a5-a1)}$, $R_{\Delta x.xxxx}$, $R_{a5-a1.xxxx}$
ABS $R_{\Delta x-(a5-a1)}$
```

Figure 3.7 The pseudo code to detect the delta error.

If the object passed all the three tests, we are going to create an impostor for the object. First, because we have the width and height of the impostor, we can create a fit size of memory space for the impostor. As we mentioned before, the memory space is so limited in embedded system. It is important to minimize the wasteful spaces. The software impostor technique cannot achieve this, since it does not have the size information. Second, whenever a pixel has been generated, it then goes to frame buffer. In our method, the pixel is not only sent to frame buffer but also sent to the memory space reserved for the impostor. The advantage to create impostor inside the hardware is there is only one rendering for an object and send the pixel to two target memory spaces. To create the impostor inside the software, the game developers have to render the same object twice. One of the rendering target is frame buffer and the other is frame buffer object. Though the impostors are re-created only when the object fails the tests, it is still an overhead to pay.

In order to save memory spaces, we create a bit-vector for the impostor. Each bit on the bit-vector corresponds to a pixel on the impostor. And the bits are initialized to 0. If a pixel was touched, then the corresponding bit is set to 1. Later, in the phase 3, if the bit says 0, then the transparent values of the pixel on the impostor is set to 0. That means pixels beyond the object are transparent.

After all above actions, the remainder works are updating the information and transiting to next phase. Since the extreme values could change, we need to update the values stored in the memory. We are now in phase 2, and we are going to go to either phase 3 or phase 4. If the object passed all impostor tests, then we go to phase 3, otherwise we go to phase 4. If we are go to phase 4, then we initialize the counter and done, otherwise, we should subtract minimum x from maximum x to get the width of the object, subtract minimum y from maximum y to get the height of the object and store the width, height and phase to the memory.

3.1.3 Phase 3 – the usage phase

This is the third phase of the four called usage phase. In the last phase, we created an impostor for an object. Now, we are trying to use it to reduce the complex computations of the original rendering. Here are actions to achieve this phase include (1) identifying the object, (2) test the impostor errors, (3) using impostor or drawing object, and (4) updating information of the object and transiting to next phase.

The first two actions are as same as the last phase. Recording the API sent two triangles for us to perform the impostor error tests and assuming the object passes the tests. Before going the action 3, we must stop the rendering process of the vertex shader, because we are going to draw the impostor instead of the object. There are ways to achieve this and our method is writing a special bit to 1 in the memory when we decide to use the impostor.

Before dealing with the next triangle, the vertex shader should check this bit. Now, we are in action 3 and we are going to use the impostor. Recording that we discard the first two triangles which are the two test purpose triangles, because the two triangles cannot be displayed on the screen. But, in this phase, once we figure out to use the impostor, we utilize these two triangles to be the quad or two triangles. To utilize these two triangles, first, we need to put them to the right place. The coordinates of the left-top, left-bottom, right top and right bottom of the quad are (minimum x, minimum y, minimum z), (minimum x, maximum y, minimum z), (maximum x, minimum y, minimum z), (maximum x, maximum y, minimum z). We interpolate the original coordinate of the two triangles to the above vales. Second, setting the texture coordinates up. Third, setting the impostor as a texture. Then, the hardware follows the normal processes to render the impostor instead of the object. After all above actions, we will see the illusion of the object to be displayed on the screen.

In the action 4, we can look at the state machine and find that there are three ways to go from phase 3. If the object failed the tests, then we check the saturated counter of the object. If the saturated counter is not saturated, then the object goes to phase 4, otherwise it goes to phase 1. The saturated counter holds the number of times that the impostor has been used. Once an object fails the tests and the saturated counter is not saturated, we think that this object is not a good candidate and we stop trying to use impostor to replace it. But, if the saturated counter does saturate, then we consider to give it chance to keep trying. If the object passes the test, then, of course, it is a good choice to apply this method and we keep trying to use impostor. If the object passed the test, then we add one to the saturated counter, stay in phase 3 and updating the extreme values. If the object failed the tests, then we reset the saturated counter to 0. And if the saturated counter is not saturated, then we need to initialize the counter and go to phase 4.

3.1.4 Phase 4 – the idle phase

This is the last phase of the four called idle phase. Because, objects who come to this phase are not good candidates to apply the impostor method, we stop trying to use the impostor on these objects. So, in this phase, we render the object in the original way and increase the counter. The object goes to phase 1 to re-try to use the impostor when the counter is bigger than a threshold. This threshold can be decided by the developer. Otherwise, the object keeps staying in phase 4.

4. Experiment Setup

4.1 The original number of instructions for GPUs to render the benchmark

In the model, we count the number of instructions should be executed by GPUs with and without applying our method. There are three major parts of processes in the original system: first of all, vertex shader, the job of the vertex shader is to place the object on the right screen position. To achieve this, all triangles of the objects need to pass though vertex shader triangle by triangle, and each coordinate of the three vertices of a triangle is transformed by vertex shader. Whereas vertex shader have to transform all the vertices, the total number of instructions that vertex shader have to deal with an object is:

$$\mathrm{instrOfVS(i)} = \#vs\_instructions(i) * \#vertices(i)$$

The above equation has two #s: number of instructions and number of vertices. We have all the objects in the GLBenchmark to count the number of objects and the number of vertices of each object. We also have all the shader language source code in the GLBenchmark and the compiler which can compile these code into assembly code. And, we compile all those shader language code and count the number of instructions that vertex shader has to deal with.

Second of all, the rom code, rom code is in charge of many things such as back face culling, clipping, rasterizing, early z testing, early alpha testing and varying processing. For back face culling, the number of instructions is:

$$\mathrm{instrOfBFC(i)} = 37 * \#triangles(i)$$

The rom code of the collaborated project takes 37 instructions to perform the back face culling. Each triangle of an object have to pass through these 37 instructions. And the number of triangles is the number of the vertices divide by 3.

For clipping, the number of instructions is:

$$\mathrm{instrOfClip(i,f)} = 62 * \#unCullTriangles(i, f)$$

The rom code of the collaborated project takes 62 instructions to perform the clipping. Because the culling has gotten rid of parts of the triangles, only the triangles that pass the culling test on the given frame would pass through these 62 instructions.

For rasterizing, the number of instructions is:

$$\mathrm{instrOfRR(i,f)} = 84 * \#pixels(i, f)$$

The rom code of the collaborated project takes 84 instructions to perform the rasterizing. Rasterizer is responsible for generating the pixels so the number of instructions to perform the rasterizing have to multiple by the number of pixels of an object on a given frame. To get the number of pixels, we only draw the object we want to count triangle by triangle and count the number of pixels of each triangle.

For early z testing, the number of instructions is:

$$\mathrm{instrOfZT(i,f)} = 2 * \#pixels(i, f)$$

Because the rom code of the collaborated project does not perform the early z testing, we guesstimate the number of instructions to perform it by ourselves. We guess 2 instructions to achieve this, because it only loads the last z value and compares it with the new z value. Pixels that have been generated by the rasterizer have to pass through this test.

For early alpha testing, the number of instructions is:

$$\mathrm{instrOfAT(i, f)} = 2 * \#unOccludedPixels(i, f)$$

The rom code of the collaborated project does not perform the alpha testing either, again, we guesstimate the number of instructions to perform it by ourselves. We guess 2 instructions to achieve this, because it only loads the last alpha value and compares it with the new alpha value. Pixels that have been generated by the rasterizer but *not* been occluded have to pass through this test. To acquire the number of un-occluded pixels, we also draw the object triangle by triangle. But, this time, we also draw the other objects to allow them to occlude the object we want to count.

For varying processing, the number of instructions is:

$$TIofVP(i) = 16 * \#verying(i)$$

The rom code of the collaborated project takes 16 instructions to process the varying, and this number need to be multiplied by the number of varying.

Third of all, fragment shader, the job of fragment shader is to decide the color of a pixel. To achieve this, all pixels that pass the early z test and the early alpha test of an objects (we call them shaded pixels, because they all need to pass through fragment shader.) need to pass though fragment shader pixel by pixel. So, the total instructions to perform is:

$$\mathrm{instrOfFS(i, f)} = \#fs\_instructions(i) * \#shadedPixels(i, f)$$

We can add all the numbers up to get the number of instructions that GPUs take to render an object on a given frame. That is:

$$\mathrm{instrOfObj(i, f)} = \mathrm{instrOfVS(i)} + \mathrm{instrOfBFC(i)} + \mathrm{instrOfClip(i, f)} +$$

$$\mathrm{instrOfRR(i,f)} + \mathrm{instrOfZT(i,f)} + \mathrm{instrOfAT(i,f)} + TIofVP(i) + \mathrm{instrOfFS(i,f)}$$

To calculate the total number of instructions that GPUs need to execute, we need to apply the above equation to all the objects and all the frames. That is:

$$\sum_{i=1}^{i=\#obj} \sum_{f=1}^{f=\#frame} \mathrm{instrOfObj(i,f)}$$

4.2 The number of instructions for GPUs to render the benchmark after applying our method

In our method, the state machine (See figure 3.1) has several variations such as no idle stage, idle 1 cycle, idle 2 cycle, and so on. Moreover, the impostor error are different, too. Though they are different forms, they are very similar. We are not going to create models for each case, it would be redundant. Instead, we will describe the state machine which has idle state and calculate the number of instructions that each state takes. For other cases, we can simply get rid of the idle state or take away some of the tests. Here comes the number of instructions that each state takes:

In the first stage, the analysis phase, the vertex shader do the same things so no changes on the number of instructions. Then the object goes into rom code, we first need to take 33 instructions to identify an object and we only pay this cost at the beginning of an object. In addition, we take instructions to find the extreme values and their indices. That is:

$$\mathrm{instrOfFEV(i,f)} = 15 * \#unBackFaceAndunClippedTriangles(i,f)$$

We add 34 instructions into the rom code of the collaborated project to find the extreme values and their indices. The 15 needs to be multiplied by number of triangles that are not back face and not been clipped, in other words, these triangles are on the screen and face to viewers. At the end of an object in the rom code, if the object fails one of the tests, then we do not need to store the information of the object since we are not going to use these values. But we do need to transit to the next phase. In this case, we add 2 instructions to transit the state. On the other hand, we take 8 instructions to store the information of the object into memory and transit the phase. For fragment shader, there is no changes on the number of instructions.

We can add all the cost up to get the total cost of the analysis phase. That is:

$$\begin{aligned}\mathrm{instrOfPh1(i,f)} = {} & \mathrm{instrOfVS(i)} + 33 + \mathrm{instrOfBFC(i)} + \mathrm{instrOfClip(i,f)} + \\ & \mathrm{instrOfFEV(i,f)} + \mathrm{instrOfRR(i,f)} + \mathrm{instrOfZT(i,f)} + \\ & TIofVP(i) + \mathrm{instrOfAT(i,f)} + 2or8 + \mathrm{instrOfFS(i,f)}\end{aligned}$$

In the second stage, the creation phase, the vertex shader has to transform 6 more vertices which hold the extreme indices. It takes:

$$\mathrm{instrOfVSExtra(i)} = 6 * \#vs\_instructions(i)$$

Then in the rom code, it takes 36 instructions to identify the object. And we spend 56 instructions to do the pixelation, rotation and delta test. If the object passes all the tests, then we are going to create an impostor for this object. Because the cost of writing a generated pixel to a texture is hidden by the writing a generated pixel to the frame buffer, we do not need to count the cost of this. At last of the rom code, it takes 8 instructions to store the information and transit the phase. The fragment shader has nothing different to do, so the number of instructions is not changed. The number of instructions of this way is:

$$\begin{aligned}\mathrm{instrOfPh2TP(i,f)} = {} & \mathrm{instrOfVSExtra(i)} + \mathrm{instrOfVS(i)} + \ 36 + \mathrm{instrOfBFC(i)} + \\ & \mathrm{instrOfClip(i,f)} + \ \mathrm{instrOfRR(i,f)} + \\ & \mathrm{instrOfZT(i,f)} + \mathrm{instrOfAT(i,f)} + 8 + TIofVP(i) + \\ & \mathrm{instrOfFS(i,f)}\end{aligned}$$

On the other hand, if one of the test was failed then we draw the object originally. The cost is same as above equation except it does not need to store the information. So, the cost is:

$$\mathrm{instrOfPh2TF(i,f)} = \mathrm{instrOfPh2TP(i,f)} - \ 6$$

In the third state, the usage phase, the vertex also has to transform 6 extra vertices. And we also take 56 instructions to detect the impostor errors. If the object passes all the tests, then we are going to use the impostor. In this case, we stop the process of the vertex shader, and we have to count the number of instructions that has been executed by the vertex shader. We assume that vertex shader and rom code have the same speed and there are more than 50 instructions in average in vertex shader code. So, we just assume during the process of the impostor error tests, there are two vertices has been transformed by vertex shader. Therefore, the number of instructions that vertex shader needs to deal with is:

$$\mathrm{instrOfVSWithImp(i)} = \#vs\_instructions(i) * 2$$

For rom code, it takes 36 instructions to identify the object. Since we are going to apply the impostor on a quad or two triangles, there is no need to do the back face culling. But we have to take 6 instructions to interpolate the position values of the first two triangles from extreme values to the position of the impostor. Moreover, instead of the original number of varying, we only need to process the position and texture varying. So, the number of instructions to process the varying is:

$$TIofVPWithImp(i) = 16 * 2$$

At last of the rom code, we take 8 instructions to store the information and transit the phase. For fragment shader, because we use the impostor to replace the object, we take 10 instructions for each pixels to apply the impostor on a quad. In addition, instead of the number of the shaded pixels of an object, we deal with the number of shaded instructions of the impostor. So, the number of instructions of fragment shader is:

$$\mathrm{instrOfFSWithImp(i,f)} = 10 * \#shadedPixelsOfImp(i,f)$$

So the total number of instructions that have to deal with in this case is:

$$\begin{aligned}\mathrm{instrOfPh3TP(i,f)} = {} & \mathrm{instrOfVSExtra(i)} + \mathrm{instrOfVSWithImp(i)} + 36 \\ & \mathrm{instrOfClip(i,f)} + \mathrm{instrOfRR(i,f)} + \\ & \mathrm{instrOfZT(i,f)} + \mathrm{instrOfAT(i,f)} + 8 + 32 + \\ & \mathrm{instrOfFSWithImp(i,f)}\end{aligned}$$

On the other hand, if one of the test was failed then we draw the object originally. The cost is:

$$\begin{aligned}\mathrm{instrOfPh3TF(i,f)} = {} & \mathrm{instrOfVSExtra(i)} + \mathrm{instrOfVS(i)} + 36 + \mathrm{instrOfBFC(i)} + \\ & \mathrm{instrOfClip(i,f)} + \mathrm{instrOfRR(i,f)} + \\ & \mathrm{instrOfZT(i,f)} + \mathrm{instrOfAT(i,f)} + 2 + TIofVP(i) + \\ & \mathrm{instrOfFS(i,f)}\end{aligned}$$

In the fourth state, the idle phase, we almost do the same thing with the original rendering except we have to take 36 instructions to identify the object, take 1 instruction to increase the counter, take 1 instruction to store the counter back to the memory and take 2 instructions to transit the phase when the counter exceed a threshold. So, the number of instructions of phase 4 is:

$$\begin{aligned}\mathrm{instrOfPh4(i,f)} = {} & \mathrm{instrOfVS(i)} + 36 + \mathrm{instrOfBFC(i)} + \\ & \mathrm{instrOfClip(i,f)} + \mathrm{instrOfRR(i,f)} + \\ & \mathrm{instrOfZT(i,f)} + \mathrm{instrOfAT(i,f)} + TIofVP(i) + \\ & 1 + 1 + \mathrm{2or0} + \mathrm{instrOfFS(i,f)}\end{aligned}$$

5. Results

5.1 Performance comparison

Throughout this paper, we want to define a thing called No Noticeable Error threshold (NNEth), and by inspection, we determine the NNEth for most objects is 6 and 3, but for taller objects, it is actually 4 and 0.1. The 6 and 4 are for pixelation test. That means if the width of an object changes too fast, then it fails the test. And the 3 and 0.1 are for rotation test, it means if an object rotates too much, then it fails the test.

In figure 5.1, there are three lines, the red line is no compression. That means we can avoid the overhead of compression, but that also means fewer objects get billboarding. The purple line and blue lines need to be read together. The purple line presents that we do perform the compression, but we ignore the overhead, whereas the blue line adds the compression overhead into the measurement. So, the blue line is always on the above. And the more objects get billboarding, the more compression overhead we need to pay.

Figure 5.1 shows that when the memory space is very restricted, the one that do not perform compression takes more instructions than the one do perform compression. The reason is that because the memory space is so restricted, only few objects get billboarding. Even though we do not need to pay the compression overhead, we still need to pay more instructions to render the objects. But, when more memory spaces are allowed, the lines get cross each other. Because now though we do not perform compression, there are enough memory to create the IBTs and since we do not perform the compression, we do not need to pay the compression overhead.

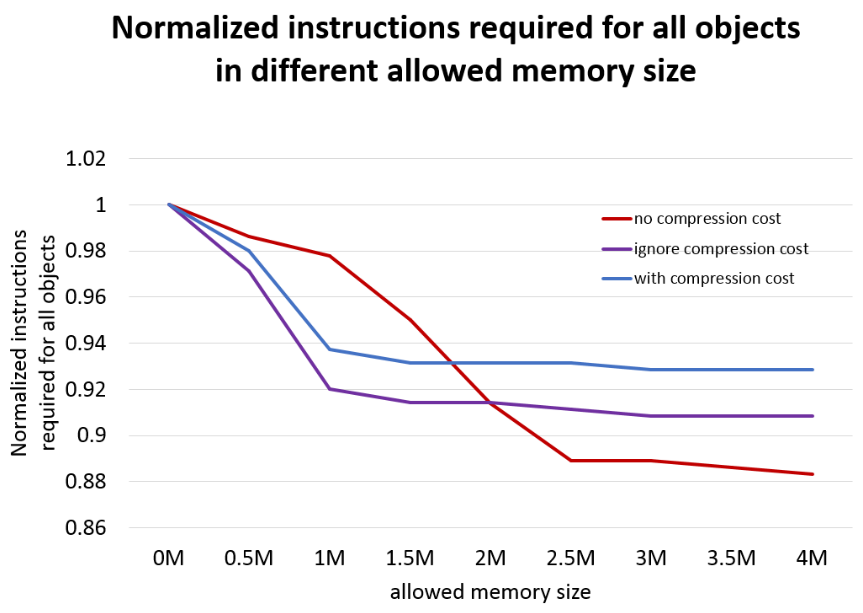


Figure 5.1 This figure shows that in the case of given different allowed memory size, how many instructions are reduced.

Figure 5.2 shows the improvements of our method. For all objects, by given more than 1.5M memory space, we get more than 10% improvement totally. That means we can reduce 10% number of instructions from the whole rendering process. Moreover, if we only measurement the improvement of those chosen objects, we get more than 25% improvement. From the result, it shows the more objects get involved, the more improvement we can get. Some reasons prevents objects getting billboarding. For example, an object is moving from the dark room to bright room, then the light effect of two frames on the object are changed. That prevents the object getting billboarding, even though the object did not fail the impostor errors.

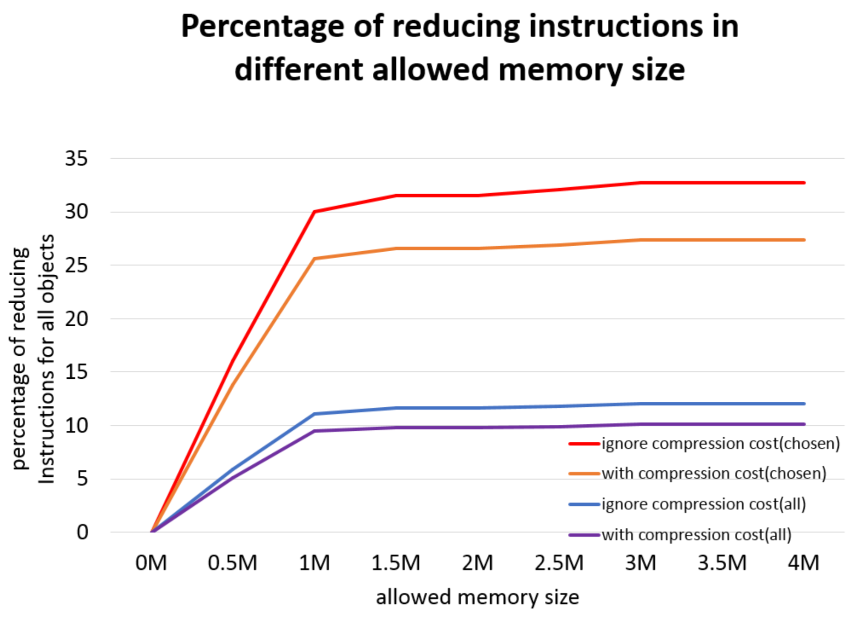


Figure 5.2

Figure 5.3 shows how does the idle cycles effect the results. When an objects often fails the test, it may not be a good candidate to keep trying to get billboarding. So, we stop trying for a period of time. We think that once an object failed the impostor error tests, it might fail the tests for a period of time, so the idle cycles might improve the performance. The result does proof this surmise, when the memory space is really constraint. But, the surmise goes wrong when more memory space are given. The reason is the idle prevent trying to create IBTs for those objects cannot use IBTs. It allows other objects to get IBTs. But, when the memory space gets bigger, the idle stage reduces the times of using IBTs. It leads the more idles, the less improvements. In this thesis, the users can modify this value themselves to get the maximum improvement.

The values of the impostor error thresholds are proposed by this paper could be modified. We try many values for these three thresholds, and we find that these values give us an unnoticeable scene. If we enlarge the values, we get better performance but poorer quality of the scene. On the other hand, if we shrink the values, we get almost the original quality of the original scene but very few improvements.

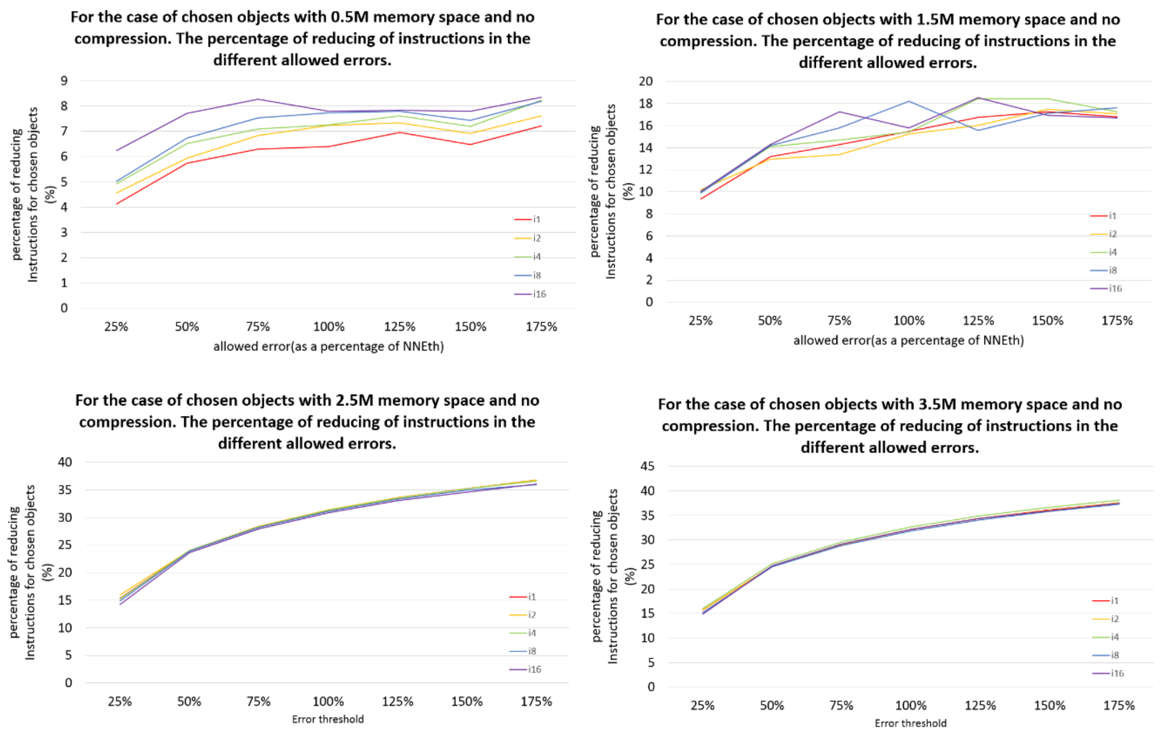


Figure 5.3

Figure 5.4 shows how many memory spaces we need at each frame. The memory space is really restricted in embedded GPUs. So, we create a compression method to reduce the size of IBTs. Although it takes instructions to perform the compression, it is still worth to compress. With the compression, we can have more IBTs in the memory. It increase the opportunities for objects to get billboarding.

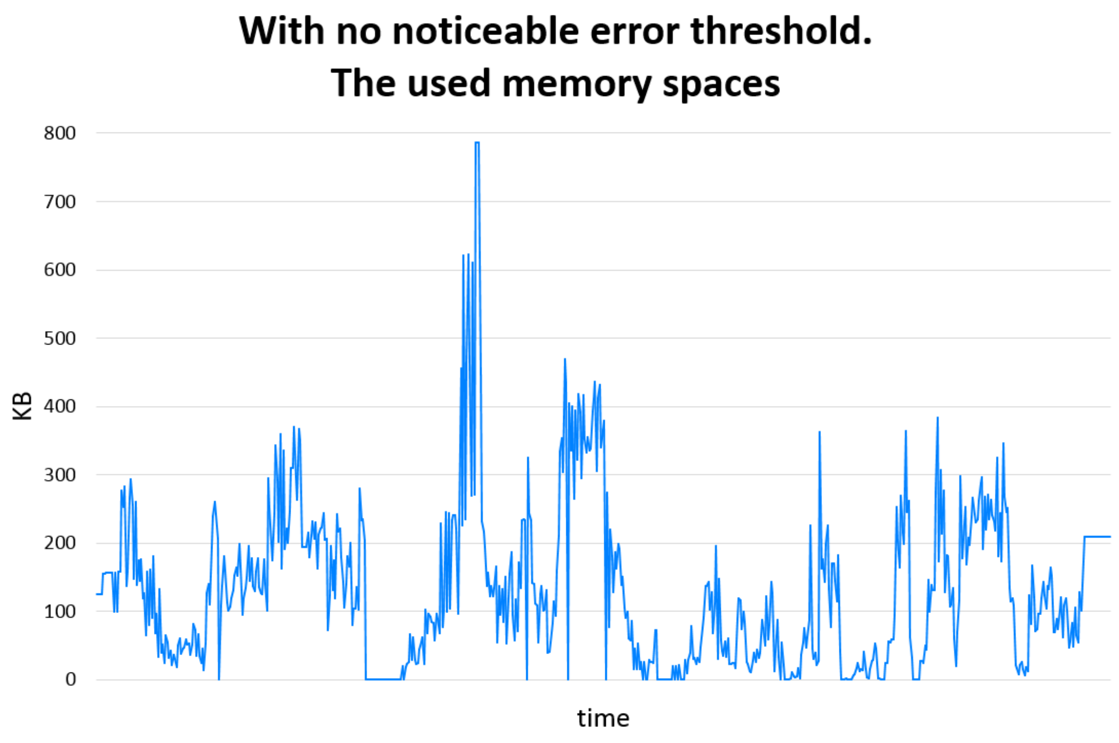


Figure 5.4

Figure 5.5 shows that the more objects are in phase3 the more improvement we can get. Because when an object is in phase3, that means it keeps using the IBTs. And the cost of rendering IBTs is cheaper than the cost of rendering objects. So, if the more objects stay in phase3, the more cost of rendering gets reduced.

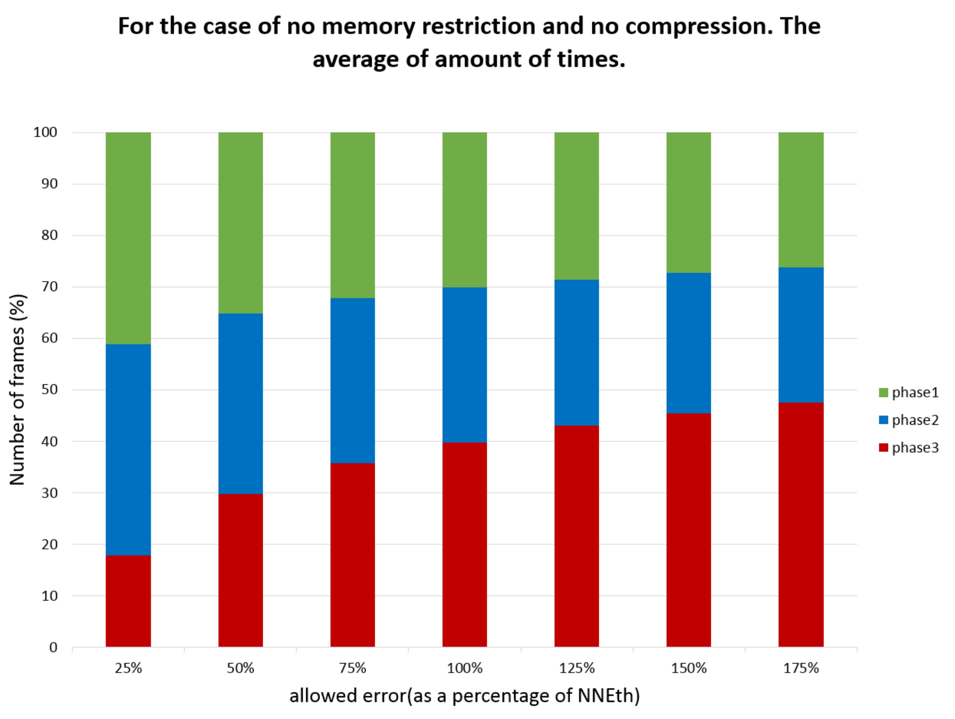

For the case of no memory restriction and no compression. The average of amount of times.
Number of frames (%)
100
90
80
70
60
50
40
30
20
10
0
25%
50%
75%
100%
125%
150%
175%
allowed error(as a percentage of NNEth)
phase1
phase2
phase3

Figure 5.5

5.2 GLBenchmark

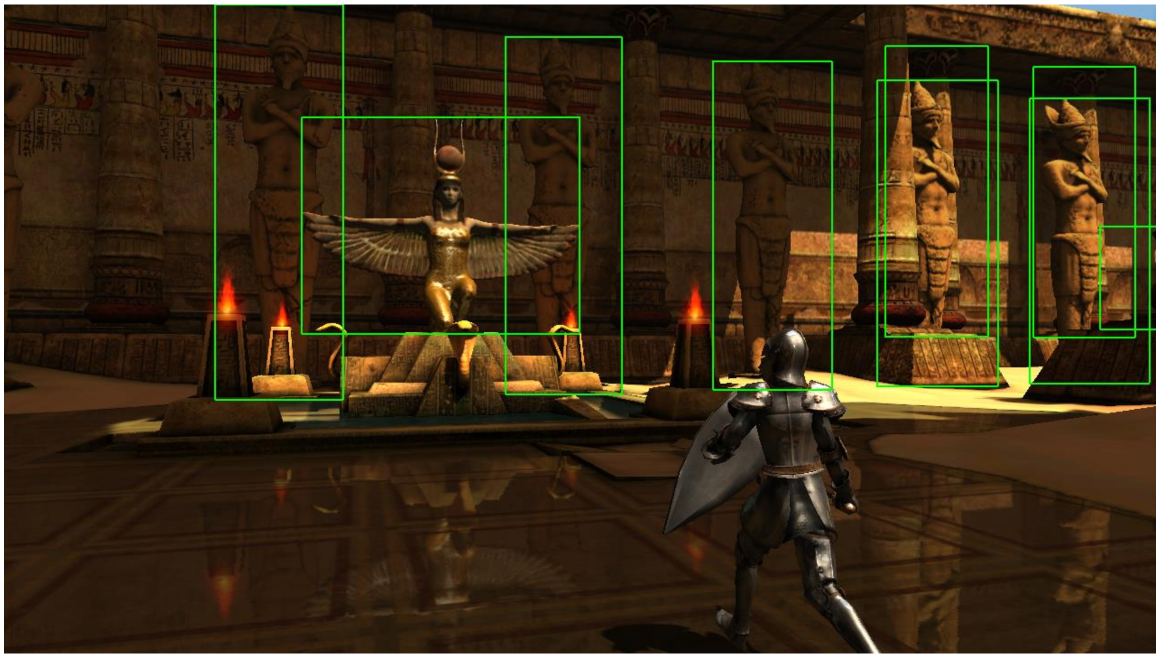


Figure 5.6

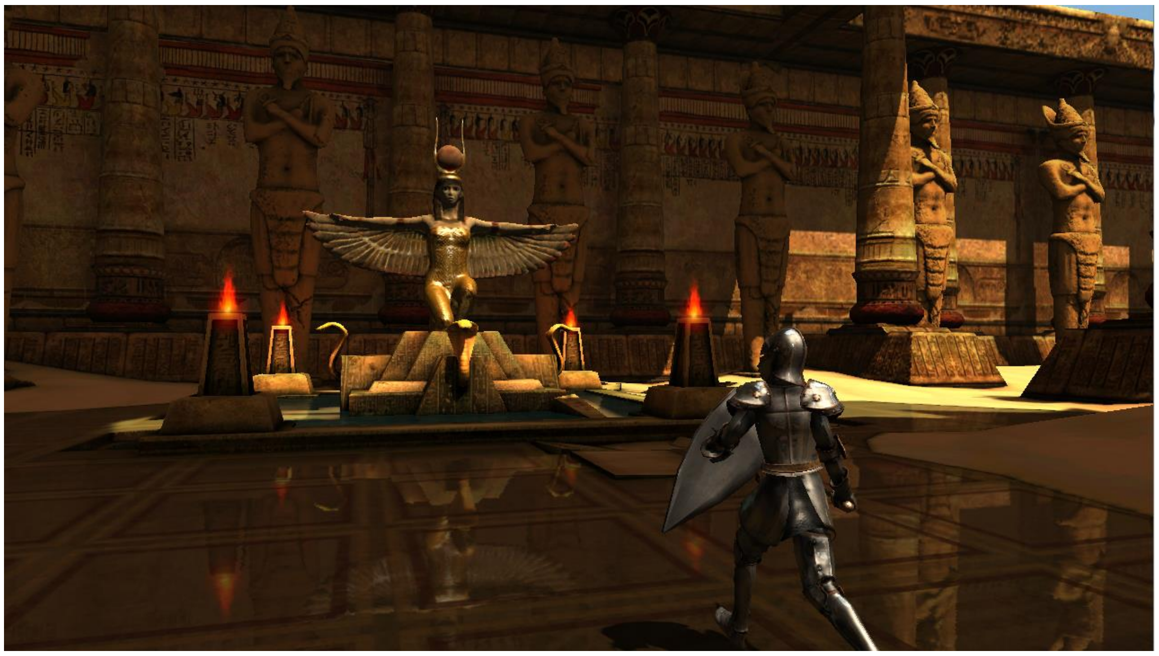


Figure 5.7

Figure 5.6 shows the result of using the impostor method to replace the original object. The green boxes in Figure 5.6 are the impostors of objects. And figure 5.7 shows the original scene. It is hard to distinguish the differences between these two pictures. That is why game developers use the impostor method to reduce the complex 3D rendering.

If we over re-use the impostor, we would see the poor quality scenes like figure 5.8. Comparing figure 5.8 with the original picture figure 5.9. It is so easy to observe the differences. Because of the too large threshold, the impostors is over re-used.

The camera is moving in and rotating around the room. It causes two impostor errors, pixelation and rotation errors. There are two snakes in the figure 5.9. We can see the pixelation error on the left one and the rotation error on the right one. Since we do not want to let the player notice the differences, we propose three impostor errors and the methods to detect the errors.

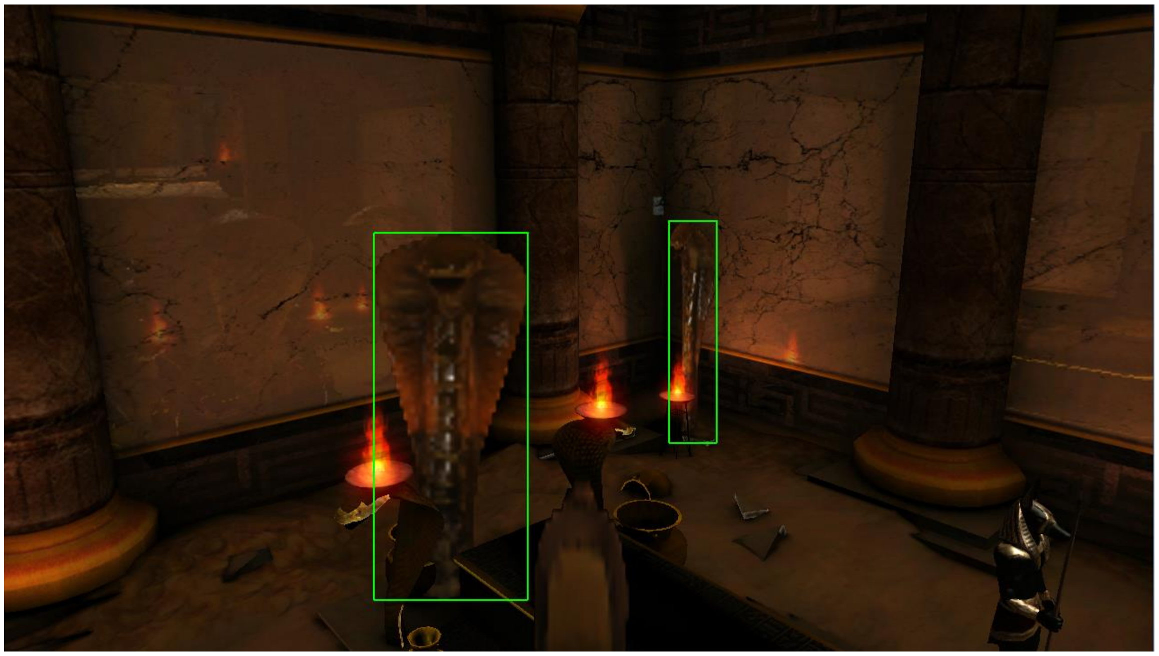


Figure 5.8

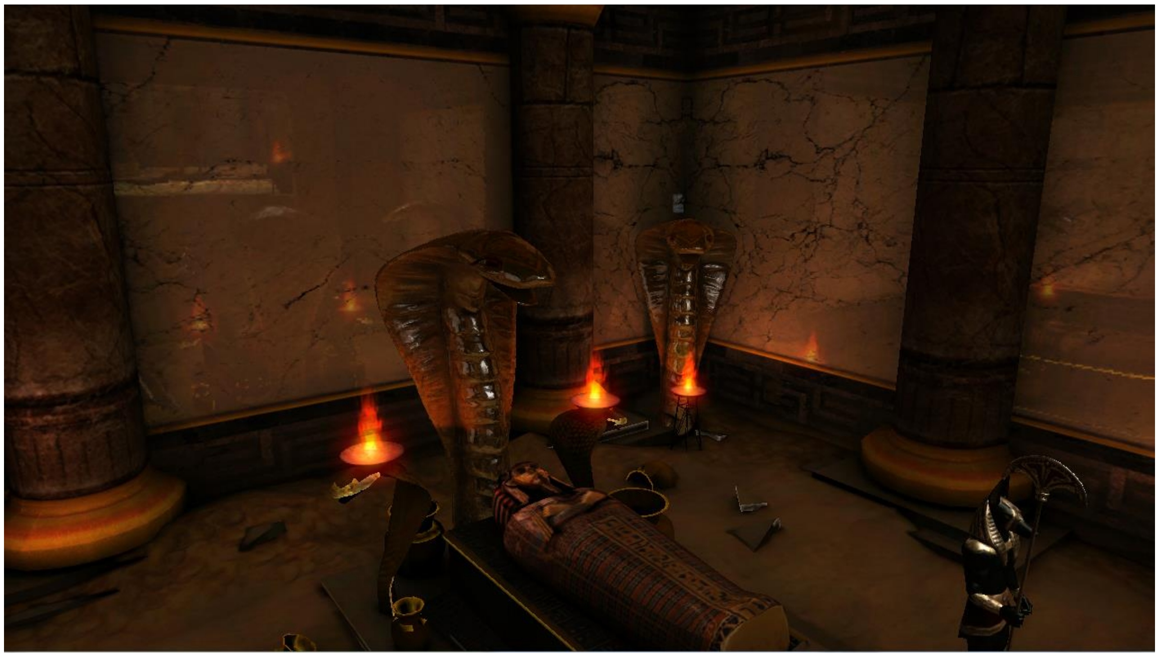


Figure 5.9

6. Conclusion

In this paper, we try to let game developers acquire all the benefits of the impostor mechanism and prevent the complicated implementation details. After the whole paper, we think it is possible to be done and it is worth to do, because the game developers only need to turn on the technique for objects and get the benefits. Though we do not really implement this method on the hardware, we do simulate the method in the software and we reduce more than 10% number of instructions.

In the future, we think that it is possible to implement this method on the hardware to get the whole benefits from software based impostor mechanism but do not need to worry about the complex details.

7. Reference

[1] Tomas Akenine-M¨oller, Eric Haines, Naty Hoffman. Real-Time Rendering*(Third Edition)*

[2] Frame buffers https://www.open.gl/framebuffers

[3] Billboarding http://www.lighthouse3d.com/opengl/billboarding/

[4] Schaufler, Gernot, "Dynamically Generated Impostors," GI Workshop on "Modeling—Virtual Worlds—Distributed Graphics, D.W. Fellner, ed., Infix Verlag,

[5] Kenneth Rohde Christiansen∗, "The use of Imposters in Interactive 3D Graphics Systems" Department of Mathematics and Computing Science Rijksuniversiteit Groningen Blauwborgje 3 NL-9747 AC Groningen

[6] Sheng-Chang Chang, "A GPU hardware-based method for automatic occlusion detection and optimization for objects and subobjects" Department of Computer Science and Engineering National Sun Yat-sen University Master Thesis

[7] GPUs memory latency http://www.sisoftware.net/?d=qa&f=gpu_mem_latency

[8] RGB565 format http://www.theimagingsource.com/en_US/support/documentation/icimagingcontrol-class/PixelformatRGB565.htm

[9] Run-length encoding http://www.fileformat.info/mirror/egff/ch09_03.htm

[10] GLBenchmark http://gfxbench.com/result.jsp