Optimalisation of a Graphics Engine for 2D Platform Games on Mobile Devices

Bachelor thesis

July 2013

Student: B. Reigersberg

Primary supervisor: prof. dr. A.C. Telea

Secondary supervisor: prof. dr. ir. M. Aiello

Contents

1 Introduction **3**
- 1.1 Mobile Games . . . 3
- 1.2 Platform Games . . . 4
  - 1.2.1 Structure and History . . . 4
  - 1.2.2 Running games . . . 5
  - 1.2.3 Performance and Visual Quality . . . 5
- 1.3 MAGE . . . 6
  - 1.3.1 Architecture . . . 7
- 1.4 Structure of this thesis . . . 10

2 Problem **11**
- 2.1 Problem definition . . . 11
- 2.2 Problem modeling . . . 11
- 2.3 Rendering pipeline . . . 12
  - 2.3.1 Vertex Shader and Buffer Objects . . . 14
  - 2.3.2 Fragment Shader and Textures . . . 16
  - 2.3.3 Fragment Operations . . . 17

3 Optimalization Strategies **19**
- 3.1 Texture Resolution . . . 20
  - 3.1.1 Method . . . 21
  - 3.1.2 Results . . . 22
  - 3.1.3 Conclusion . . . 23
- 3.2 Texture Parameters . . . 24
  - 3.2.1 Method . . . 27
  - 3.2.2 Results . . . 27
  - 3.2.3 Conclusion . . . 28
- 3.3 Alpha Blending . . . 29
  - 3.3.1 Method . . . 30
  - 3.3.2 Results . . . 30
  - 3.3.3 Conclusion . . . 31
- 3.4 Texture Format . . . 31
  - 3.4.1 Method . . . 34
  - 3.4.2 Results . . . 35
  - 3.4.3 Conclusion . . . 36
- 3.5 Depth Testing . . . 36
  - 3.5.1 Conclusion . . . 36

4 Conclusion & Discussion **37**
4.0.2 Limitations 38
4.0.3 Future work 38

Chapter 1

Introduction

1.1 Mobile Games

Mobile games are games designed for mobile devices. Mobile devices include for example PDAs, mobile phones, tablets and other portable media devices. With the recent uprise of mobile devices such as the smartphones and tablets it may come as no surprise that the popularity of mobile gaming is also flourishing. In the US alone the number of mobile gamers jumps 35 percent every year with an estimated total of 21 million tablet gamers in 2012. This is lower in key EU areas, but still 15 percent. On iOS over 6.6 million games are sold every day in the US and EU alone [7].

Almost all smartphones and tablets can connect to the internet and grant the user easy access to so called digital application distribution platforms. These platforms allow the user to easily download new applications directly to his or her device, rate them and comment on them. These platforms are also providing developers and publishers with a platform to easily distribute their applications to a massive audience. Because of this, the number of applications on these platforms, including mobile games, is growing every day.

One of the mayor digital application distribution platforms, The Google Play Store for Android OS shows that mobile games are dominating their top rated, top grossing and most downloaded application lists [3]. One of the most popular mobile games is Angry Birds by the Finnish company Rovio Mobile and has been downloaded 1.7 billion times [5] .

Most smartphones and tablets lack possibilities for user interaction which are available on other devices used for gaming, such as a mouse and a keyboard for a PC or a controller for gaming consoles. Most smartphones and tablets only provide touch input and motion sensors but do have relatively high resolution screens. Due to this nature, the gameplay of mobile games is usually very simple and the visual appearance is very important.

1.2 Platform Games

1.2.1 Structure and History

Platform games form a genre of video games in which the game's character is required to move and jump from platform to platform to reach a certain destination or score. In doing this the character may also encounter various obstacles and enemies. It is the player's task to control the character. He must make sure the character avoids or destroys the obstacles and enemies, and time the character's jumps right to prevent the character's untimely demise.

Platform games are among the oldest of video games. The first Donkey Kong game, developed by Nintendo in 1981, is known to be the first true platform game [2]. This was a single screen platform game meaning that the screen contained one entire level. There was no scrolling screen or transitions to other levels/screens.

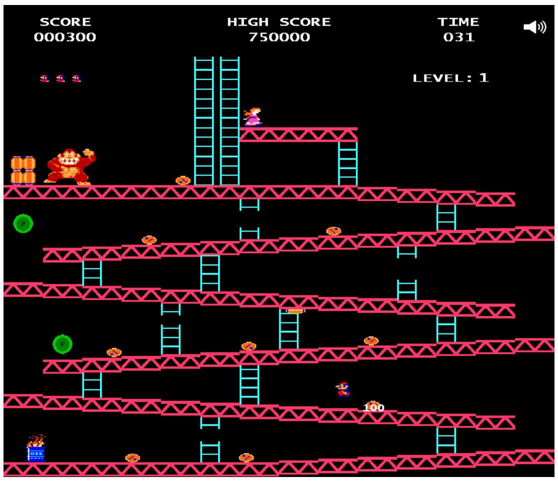

Figure 1.1: Donkey Kong (1981) by Nintendo was the first true platform game

Modern platform games are far more complex than that, usually including more complex AI for the enemies and a complete physics engine. Despite this, the basic gameplay concept remains largely intact - jumping over obstacles and avoiding enemies. Platform games have namely evolved graphically, moving from 8-bit pixel games to multiple layers of high resolution texture images and post-render effects.

These multiple layers are used to create a sense of depth by using parallax motion. Parallax is the effect that objects in the distance appear to move more slowly or even stand still, while objects that are close by appear to move faster. For example, when you're sitting in a car in motion the trees on the side of the road appear to rush past you while the sun, moon or a building in the distance appear to stand still. This works the same in platform games where the 'lowest' layer contains those objects which are supposed to appear far away from the camera and the 'highest' layers those which should appear close by. These platform games are a subclass of a family of computer graphics applications known as *2.5D graphics*.

Platform games were particularly popular in the 90's with a market share of 15 percent in 1998 and even more before [1]. After that, popularity dropped significantly to a market share of only 2 percent in 2002 [1]. This was likely due to the upcoming of overall more complex and 3D games. However, simple concepts are popular concept for mobile games. Therefore a whole new market exists for platform games. In the recent years so called running games have flooded this market.

1.2.2 Running games

Running games, also known as endless running games, are a subset of platform games. In these games the game's character runs automatically, typically in a left-to-right fashion,(in case of 2D examples) through a continuously generated world. The player does not control the horizontal movement and can only jump and/or shoot to avoid incoming obstacles and enemies. In most of these games this theoretically goes on forever, but due to increase in game speed or difficulty with time or distance, the player will eventually lose the game. The goal of such a game is not to reach a certain destination but to survive for as long as possible (Is term of either time and/or distance) or score as many points as possible. To illustrate the popularity of these games, Apple awarded Rayman Jungle Run, a running game seen in Figure 1.2, as the best iOS game of 2012. [4]

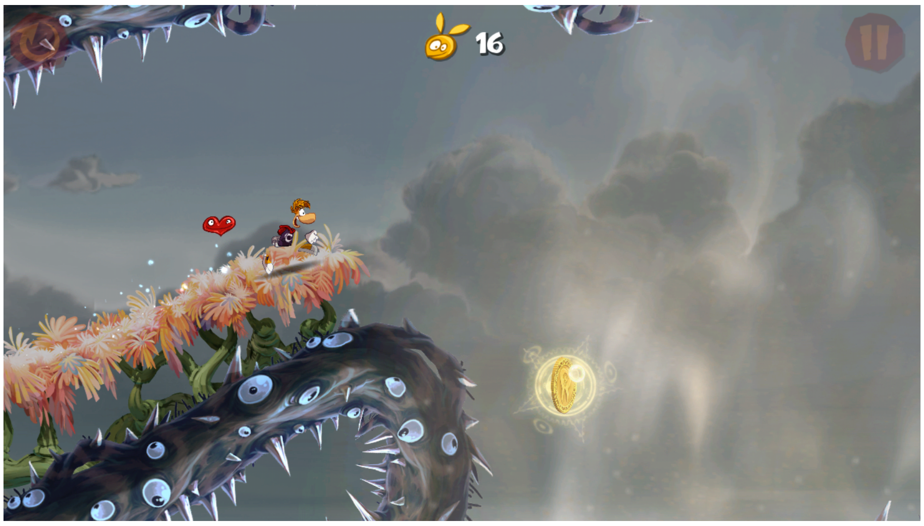

Figure 1.2: Rayman Jungle Run (Ubisoft), a popular running game

1.2.3 Performance and Visual Quality

The goal of running games is to challenge the player's reaction time and dexterity with the device's controls. Therefore, it is very important for these games to run smoothly. In order for these games to run smooth they must maintain a certain amount of frames per second.

Frames per second, often abbreviated to FPS, describes the rate of which the image on screen is refreshed. Updating a game's state and rendering the scene can take some time, which will influence the possible FPS. If the FPS is too low the illusion of motion is lost. If however the FPS is high but varies a lot in game the motion will seem uneven. Therefore it is important to keep the FPS constant as well as high. The minimum FPS is debatable and varies from game to game.

A drop in the FPS could easily cause the player to make a mistake, resulting in a game over screen - when it is actually the game which is at fault. Should this happen when the player is just about to beat his friend's score after say fifteen minutes of contiuous gameplay, the player would most likely get very upset, toss the game and the developers (Which would be us) can expect a negative review.

But not only FPS is important. As we mentioned before in section 1.1, the visual appearance is very important for mobile games. Now we want to have multiple layers of objects using detailed texture images and maintain a high and constant FPS. This all on mobile devices, which typically have limited hardware compared with a PC or a gaming console. This can be a problem. Therefore, we aim to study the challenges that a graphics engine for 2D platform games typically face in providing both a high and constant FPS as well as a high visual quality and propose ways to solve or ease these problems.

1.3 MAGE

For the purpose of studying performance challenges for a graphics engine for 2D platform games on mobile devices we will look at a possible game engine for such a game. MAGE, *My Android Game Engine*, is such an engine developed by myself and a small group of students of the Rijksuniversiteit Groningen for the Android operating system.

MAGE is written in Java (Android) and is a fairly small engine consisting of little over 15000 lines of code. The engine, at this point, provides only the basic functionality for a simple 2D platform game as seen in Figure 1.3. This makes it easy to pinpoint issues for performance.

Since we have access to MAGE's source code, and extensive knowledge of the structure and code of the graphics engine we can easily:

- Measure performance
- Study bottlenecks
- Propose, implement and test improvements

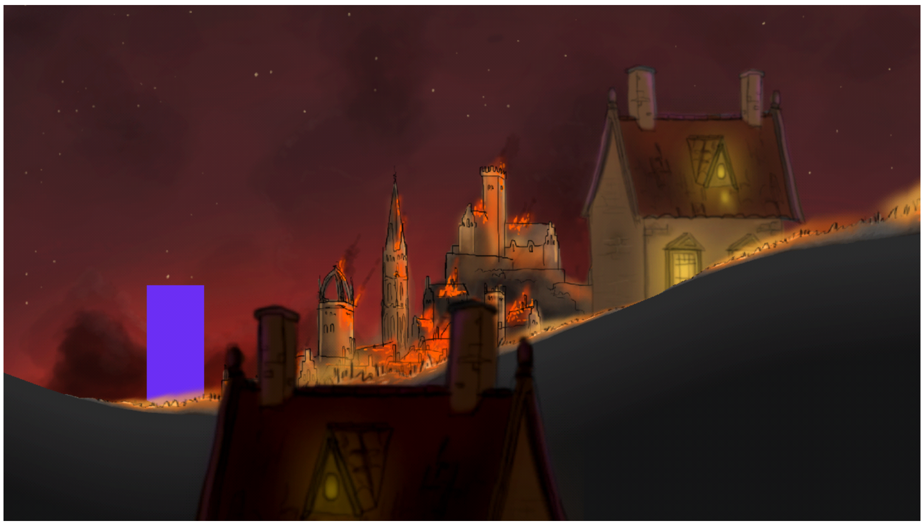

Figure 1.3: Example of a simple platform game implementation made in MAGE

We will use the implementation of the MAGE graphics engine excluding the rest of the engine and adjust it to a suitable testing environment. This way we can perform time measurements solely over the graphics engine without having to worry about the game and/or physics engine interfering with the timing results. We can also easily control the scene, eliminating independent variables and automate time measurements while changing the scene. We will also use a simple implementation of an endless running 2D platform game in MAGE to discuss wether possbile improvements are possible or desirable within the context and structure of an actual game.

1.3.1 Architecture

Figure 1.4 shows a very simplified overview of MAGE's architecture.

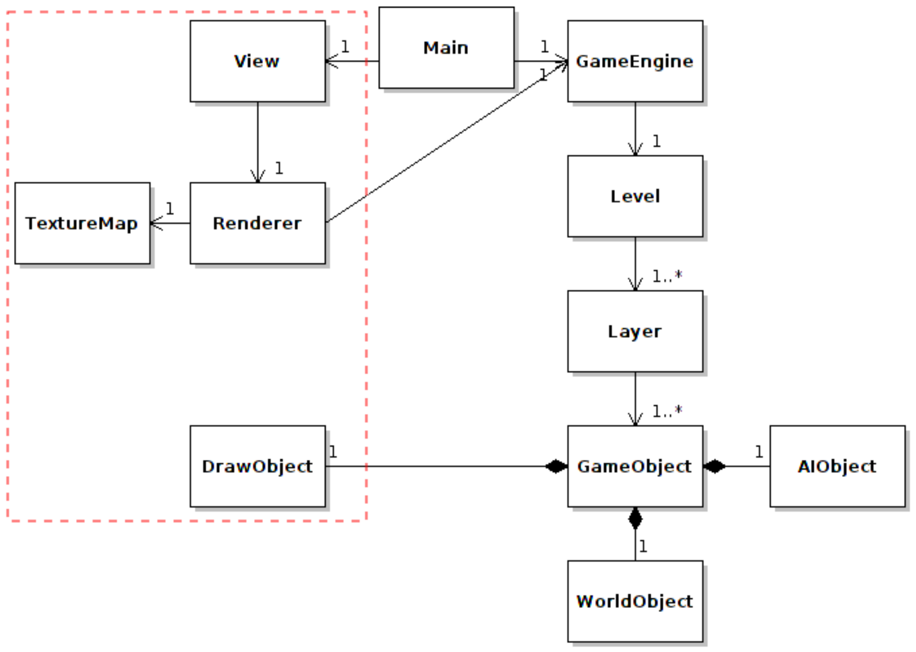

Figure 1.4: A simplified architecture. The red dotted line circles the classes which are considered part of the graphics engine

Note that this architecture is simplified and serves the purpose of giving an idea of the basic concept of the graphics engine. Classes for sound, music, physics, input etc. are not included in the architecture as are helper classes, implementations of interfaces and other things that are of no direct importance to the concept of the graphics engine.

Main

The Main class extends from the Activity class and contains the *onCreate*(...) method. This is Android specific and is comparable with the *main*(...) method in an ordinary Java project. This is the method which is called when the application is launched, and the Activity is started on Android. This class will perform the initialisation of the GameEngine and View objects and start a new thread which contains the game loop seen as pseudo-code in Listing 1.1.

Listing 1.1: Game loop

```
while(running) {
    start = currentTimeMillis();
    engine.update();
    view.requestRedraw();
    end = currentTimeMillis();
    sleep(1000.0f/FPS - (end - start));
}
```

The game loop updates the engine and then requests a redraw of the view every (1000.0/FPS) milliseconds. This effectivly draws FPS number of frames per second. This is done to keep the number of frames per second constant as stated as important in Section 1.2.3. Note that if engine.update() and view.requestRedraw() together take too lang - then $1000.0f/FPS - (end - start) < 0$. No sleeping will take place and the required number of frames per second can not be met.

GameObject

A GameObject is a representation of an in-game object. Every object in the game is an instance of the GameObject class. This ranges from obstacles such as a rock or a tree in the background to enemies, platform and the player's character. A GameObject is composed of a DrawObject, a WorldObject and an AIObject.

DrawObject contains information on how the GameObject looks and should be drawn. DrawObject is an interface which has a draw method. This method can be called by the graphics engine and draws this object to the screen. A typical implementation of a DrawObject has a set of vertices, texture coordinates (per vertex), colors (one per vertex) and a reference to a texture.

A WorldObject contains information about the GameObject's position and orientation in the model. This object is managed and updated by the physics engine. In order to draw a GameObject's DrawObject in the right position and the right orientation the graphics engine needs the WorldObject to construct a matrix of this position and orientation. This Matrix will function as the so called Model Matrix.

An AIObject contains information on how the GameObject behaves in the game and how it should respond to various events, circumstances and conditions. If, for instance, an enemy should move towards the player or how the player's character should react to user input is behaviour programmed in the object's AIObject. The AIObject in important to the graphics engine as it controls a GameObject's DrawObject. An AIObject would in case of an animation for instance determine which animation to play and when to go to the next frame. The animation feature however incomplete, so for now the graphics engine has no interaction with AIObject.

TextureMap

The TextureMap class is a hashmap which maps image names to Texture objects. The Texture class functions as a wrapper class for the texture id given by OpenGL and some additional information about the texture. The TextureMap is initialized at the start of every level and contains all loaded textures. Every textured DrawObject has an image name mapping to a Texture object in the TextureMap instead of each having their own texture id. This way we prevent loading the same texture twice when it is used among multiple DrawObjects.

Level

The Level class represents a single level in game. It is divided into multiple Layer objects.

Each Layer has camera factors for both x and y which determine how much faster or slower this layer moves relative to the camera movement, to achieve the layered 2.5D structure with parallax as mentioned in Section 1.2.For example, a layer with $x\_factor = 1.0$ and $y\_factor = 1.0$ moves just as fast as the camera in both horizontal and vertical directions. A layer with $x\_factor > 1.0$ and $y\_factor = 0.0$ would move faster than the camera in horizontal direction but not move vertically at all.

Each Layer also manages a list of all the GameObjects which are present in that Layer. All GameObjects which are in such a Layer of the active Level object form the set of all objects present in the game. These are the only objects being updated, used for collision detection and drawn.

Renderer

The Renderer is the class which contains the main drawing code and will perform the actual drawing on the screen. The Renderer can access the Layers containing all of the GameObjects via the GameEngine, which holds the active Level objects. In order to draw a GameObject to the screen, a Model View Projection matrix is necessary. This matrix is a composition of the Model matrix, View matrix and Projection matrix.The Model matrix is used to map an object's local vertices to model space, the View matrix is used to map these to camera space and finally the Projection matrix to map them to screen space. The Projection matrix only has to be calculated once (Assuming no window size changes on mobile devices) using the width and height of the screen. The View matrix has to be calculated for every drawn Layer, calculated using the camera position and the Layer's camera factors. The Model matrix has to be calculated for every GameObject, which as mentioned earlier, is kept in the WorldObject managed by the Physics Engine. How the basic drawing code looks can be seen in Listing 1.2

Listing 1.2: Main drawing code

```
for all Layers in the active Level {
  construct view matrix
  for all GameObject in this layer {
    retrieve model matrix from WorldObject
    calculate the model view projection matrix
    call draw method of DrawObject
  }
}
```

1.4 Structure of this thesis

We are going to study the challenges that a graphics engine for 2D platform games typically face in providing both a high and constant frames per seconds, as well as a high visual quality and propose ways to solve or alleviate these problems.

In order to do this we will analyse the graphics engine for such games. We will look at MAGE as we have understanding of and access to the source code. We will analyse the graphics engine for MAGE to find possible bottlenecks and performance issues. This way we want to find out how we can optimize such an engine to provide a high and constant FPS as well as a high visual quality.

In the next chapter we will further analyse the problem. Here we will model the problem and analyse this model. We will discuss which variables we can vary within this model and the graphics engine. This way we will construct a set of variables which we think could possibly affect the performance and visual quality of the game. In the chapter Optimalization Strategies we will look at the set of variables constructed in the Problem Analysis chapter and argue why they could affect the performance and visual quality and propose solutions or alleviations. Based on these solutions or alleviations we will perform tests and performance measurements to conclude wether it are good solutions or alleviations.

In the Conclusion & Discussion chapter we will summarize the various conclusions found in the Optimalization Strategies chapter and formulate a final conclusion. We will also discuss the limitations of our performance test.

Chapter 2

Problem

2.1 Problem definition

As explained in Section 1.2.3 we are studying the challenges that a graphics engine for 2D platform games face in providing both a high and constant FPS as well as a high visual quality. We are looking to propose solution to or alleviate these challenges.

A graphics engine can be described as a function with input and output. This input is typically a set of vertices and textures, called a scene. The output is the image presented on the screen after processing the vertices and textures. As it is important that the game looks good the visual quality of this image is very important. It is also important that the rate in which these images are created is high enough to result in an acceptable number of frames per second. Therefore we are interested in a quality of gameplay value $q_g \in Q_g$ consisting of the visual quality of the produced images Q_v and the interaction quality of the game Q_i. Given the set of all possible scenes S to draw as input, we now we can define a seperate cost function f to model the quality of a graphics engine.

$$f : S \to Q_g$$

2.2 Problem modeling

Before the scene is an image it goes through a set multiple rendering algorithms RA.

$$f : S \times RA \to Q_g$$

In order to optimize f to maximize Q_g we can either optimize over S and/or optimize over RA.

Optimization over S is possible. We can reduce the complexity of the scenes $s \in S$ by for example reducing the number of objects by removing objects which solely serve for the purpose of decoration. A simpler scene to draw will obviously result in a faster rendering of this scene, hence increasing Q_i. However, the decrease of the complexity of a scene will also likely result in the loss of visual quality, hence a decrease of Q_v. For example, removing decoration such as trees will negatively affect the richness of a forest scene and hence the visual quality. Also, this way the scene's designer would have the responsibility of the optimalisation of the scene and would likely have to create multiple scene-sets for different types of rendering engines. This is counterproductive, so optimizing over S is undesirable.

So we keep the specification of a scene $s \in S$ fixed and only alter our rendering algorithms $r \in RA$. This way the rendering engine will decide how a scene $s \in S$ is rendered. The rendering engine could take runtime performance factors and platform-specific information in to account with this decision. The rendered scene could also reduce the complexity of the input scene. The difference here is that now it is not the scene's designer's responsibility, but that of the rendering engine. So now the question is: How do we optimize f over RA?

2.3 Rendering pipeline

In order to optimize over RA we have to look at what RA is exactly. In section 1.3 it was mentioned that for our graphics engine we use OpenGL ES2.0. Thus in our case RA is the OpenGL ES2.0 rendering pipeline seen in Figure 2.1.

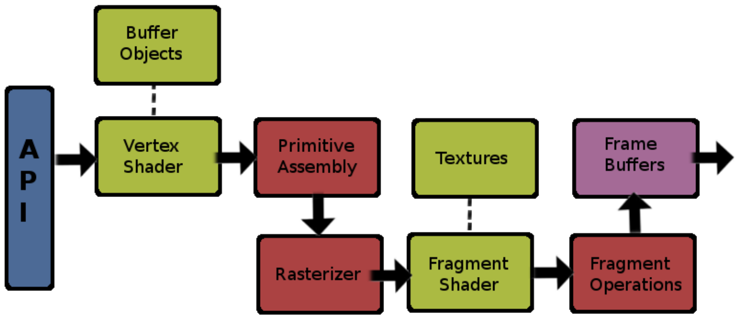

Figure 2.1: The OpenGL ES2.0 rendering pipeline

Vertex shader and buffer objects

The vertex shader will process the vertices which are submitted and in the buffer objects. Each incoming vertex is converted into a single outgoing vertex. This is done based on a user-defined vertex shader program. [6]. For example an incoming vertex could be converted to screen space.

The vertex shader can pass on other data to the fragment shader, which will be interpolated over the vertices. This can be used for texture coordinates for instance. If for example vertex A has a texture coordinate x value of 0.0 and vertex B has a texture coordinate x value of 1.0 then all fragments in between vertex A and vertex B will have an interpolated value between 0.0 and 1.0.These are called varying variables.

Finally, the outgoing vertices are passed on to the primitive assembly stage.

Primitive Assembly

In the primitive assembly stage the vertices received from the vertex shader are assembled into primitive shapes. There are three possible primitive shapes, being a point, a line segment and a triangle. The output of this stage is a set of primitives which are passed on to the rasterizer stage.

Rasterizer

In the rasterizer stage the primitives received from the primitive assembly stage will be rasterized. The output is a set of fragments. These fragments each have a position in screen-space and are used to compute the final color for a pixel. This happens in the fragment shader stage.

Fragment shader and textures

The fragment shader basicly loops over all fragments received from the rasterizer stage. For every fragment a color value and depth value are calculated based on a user-defined fragment shader program. The calculation may involve the use of the varying variables passed on from the vertex shader. Take the example of texture coordinates. These can (but don't necessarily have to) be used to perform a texture lookup in a given texture - which can only be done in this stage.

For each fragment the output is a color, depth and stencil value.

Fragment Operations

Before the fragment color from the fragment shader is written to the screen is passes through the fragment operations stage. In this stage the fragment values undergo various tests. A stencil and depth test, if enabled, are performed. If one of these fails the fragment is discarded. If it passed blending, if enabled, is performed on the fragment color and the existing color on the screen.

As seen in Figure 2.1 the stages are color-coded. The stages with a green color are the stages which we, as the programmer, have full control over. The stages with a red color we only have limited control over and the final framebuffer stage in purple is optional.

The limited control over the red stages means that we can, at the most, only control their parametrization. To illustrate the difference, in the vertex shader stage we have full control over what happens to the vertex before it passes to the next stage with the means of the vertex shader program, whilst at the fragment operation stage we can only turn the various test on and off with the means of parameters.

For our test game we will not use frame buffers as they are used for more complex post-render effects and such. These can really only be optimized for specific post-render effects and implementations - for example how much framebuffers are needed, the actual implementation of the effect in the vertex and fragment shader. This is too diverse from game to game to make a test case to represent this.

The green stages give us full control, so we can analyse these stages to see what we can do there to improve Q_g. So we will at least analyse the vertex shader and buffer objects stage and the fragment shader and textures stage. For the red stages we must determine wether or not they have some form of parameterization and wether they can be varied in the context of a 2D platform game.

Looking at the GL ES2.0 functions and their documentation there is no parameter for us to vary for the primitive assembly stage [?]. For the rasterizer stage only the width of rasterized lines can be varied by means of the $glLineWidth$ function. As we are not interested in drawing the lines and meshes for our platform games, but rather use textures instead, this too is of no use for us. The fragment shader offers multiple parameters to switch tests such as blending and depth on or off. As our platform game uses textures and colors and multiple layers we can expect to have some use for blending and depth. We can analyse these tests and the parameters.

So possible challenges can be found in the following stages in the rendering pipeline:

- The Vertex Shader and Buffer Objects
- The Fragment Shader and Textures
- Fragment operations.

2.3.1 Vertex Shader and Buffer Objects

The vertex shader could be optimized in the following respects:

- reduce number of vertices
- Optimize the user-defined vertex shader program.

The buffer objects could be optimized in the following respects:

- reduce buffered data
- tightly pack buffered data.

Both the vertex shader and buffer object can be optimized by a reduction of data. But how much data do we have in the first place? Imagine a typical 2D platform game scene as seen in Figure 2.2.

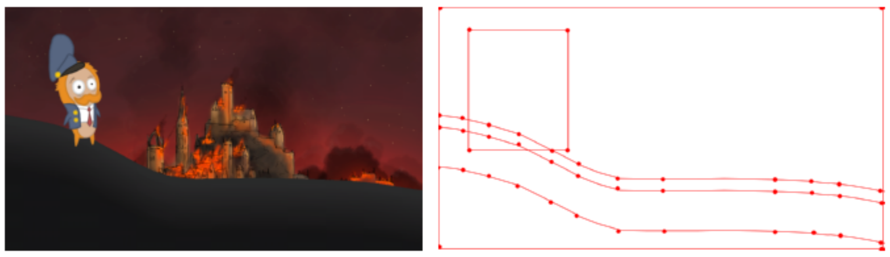

Figure 2.2: Analysis of number of vertices in a typical 2D platform game scene. On the left a typical game scene, on the right the same scene stripped from textures only showing estimation of vertices

Most objects are just simple rectangles consisting of just 4 vertices functioning solely as a frame to map a texture over. In the case of our platform game the ground is the most complex shape, but it doesn't even hold a candle to the complexity of even a simple 3D environment seen in Figure 2.3.

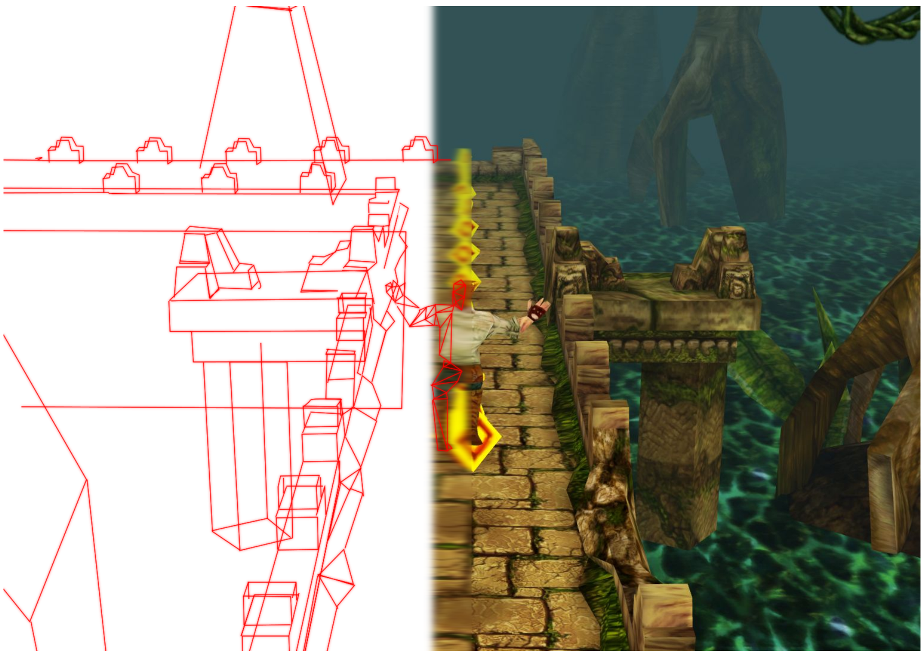

Figure 2.3: Sketch of lines and vertices in a simple 3D game (Temple Run)

So not only would it be impossible to make our shapes even less complex by reducing the number of vertices - it probably wouldn't improve a whole lot as 3D games such as Temple Run seen in Figure 2.3 show little to no performance issues dealing with far more complex geometry. Therefore we will not look to optimize this.

For this same reason we will not look how to optimize the storage in the buffers. There is just too little data to store to spend time on optimizing this at this point.

The vertex shader program also can not be optimized. In most 3D applications the program is used to perform calculations for lighting and other effects. In case of our simple 2D game the vertex shader only converts the vertices to screen-space and interpolates the texture coordinates over the fragments. Both of which we can not do without.

2.3.2 Fragment Shader and Textures

The fragment shader could be optimized in the following respects:

- Optimize the user-defined fragment shader program.

Note that following RA we can only reduce the amount of fragment by either reducing the amount of vertices or adjust them to change the resulting primitive shapes. So we can not optimize this.

The fragment shader program in our case only performs a single texture lookup per fragment in order to calculate the final color for the fragment. We can not do without this. Therefore the fragment shader program can not be optimized.

The textures could be optimized in the following respects:

- Resolution
- Format
- Parameters

All of these variations could possibly affect both Q_v and Q_i. Therefore these topics deserve more attention.

- Texture resolution is the amount of detail that a texture image has. The higher the resolution, the more detail can be seen in the texture image. Resolution of a texture image can be described in various ways. In this thesis, when I refer to the resolution of a texture image, I refer to the pixel resolution. The pixel resolution is the number of pixels in the image, often described as two positive integers with the first one being the number of pixel columns, or the width of the image, and the other the number of pixel rows, or the height of the image. For example an image with 512 pixel columns and 1024 pixel rows would have a resolution of 512 by 1024 pixels or simply 512 x 1024. It would be interesting to see how varying the resolution might affect the rendering speed.

- Texture format is the format the texture is stored in. One could picture an image as a large 2D array, each index representing a pixel in the image. The entry for this value usually represents a color value. The entry could for example be red, green and blue colors. Internally we can chose to store these images in a different format than their external format. As some formats are smaller than others it is interesting to see if and how this will affect the rendering speed.

- Texture paramaters are parameters which can be set in OpenGL per loaded texture. These parameters are used to control how the texture is treated in certain circumstances. We can vary these and study the effect on the rendering speed.

So for the fragment shader and textures stage important factors are:

- Texture resolution
- Texture format
- Texture parameters

2.3.3 Fragment Operations

In the fragment operations stage the following operations are performed.

- Depth testing
- Stencil testing
- Blending

We can influence these operations by means of changing parameters.

- Depth testing is used for depth management. It used an additional buffer called the depth buffer. Before a fragment is written, the depth value of this fragment is compared to the existing depth value in the depth buffer. If for example the value in the buffer is smaller, it means that another fragment which is 'closer' is already written in the color buffer and this our fragment is discarded. This way objects which are far away will appear to be behind near objects, in whichever order we draw them. Depth testing can be either turned on or off. As 2D games use an ortogonal projection there is no actual depth involved. However, there is a certain order in which we want things to appear. Should grass for example appear in front of the player or behind? In order to achieve this we would normally have to sort our list of objects to fit our needs. We could however abuse the depth testing to just give our objects a depth value (which doesn't matter as the projection is orthogonal) and we no longer have to worry about the drawing order. This would save us from having to implement an efficient sorting mechanism, so this topic deserves further attention.

- Stencil testing introduces another buffer called the stencil buffer. This buffer in contrast to the color and depth buffer holds values with application specifics meanings. Depending on a comparance between a certain reference value and a value in the stencil buffer (called the stencil test) a fragment is either passed on or discarded. In games this is primarily used for shadows in 3D application. In a 2D platform game there is no direct use for the stencil buffer and test. Hence, we will not test it.

- Blending is used to perform a blend between a new incoming fragment color in the color buffer and the color already present at that location. This may be, for example, the color of the background or the color of an object previously drawn. The blending is performed based on a blending equation, which has te be evaluated for every fragment drawn. Due to these extra calculations the rendering speed might suffer. Blending can be turned on or off and the equation can be changed. This topic deserves attention.

So for the Fragment Operations important factors are:

- Depth testing
- Blending

Chapter 3

Optimalization Strategies

In Chapter 2 we outlined that the most important factors which may influence Q_i and Q_v are as follows:

- Texture Resolution
- Texture Format
- Texture Parameters
- Blending
- Depth testing

In this chapter we will study the effect of these factors on the Q_i and Q_v. We consider our cost function $f : RA \rightarrow Q_g$ to be approximated by

$$f : (\mathit{Texture\ Resolution}, \mathit{Texture\ Format}, \mathit{Texture\ Params}, \mathit{Blending}, \mathit{Depth\ Testing}) \rightarrow Q_g$$

In order to study this function we will vary each of these factors seperately. By doing so we hope to understand how the overall quality is affected by these factors.

3.1 Texture Resolution

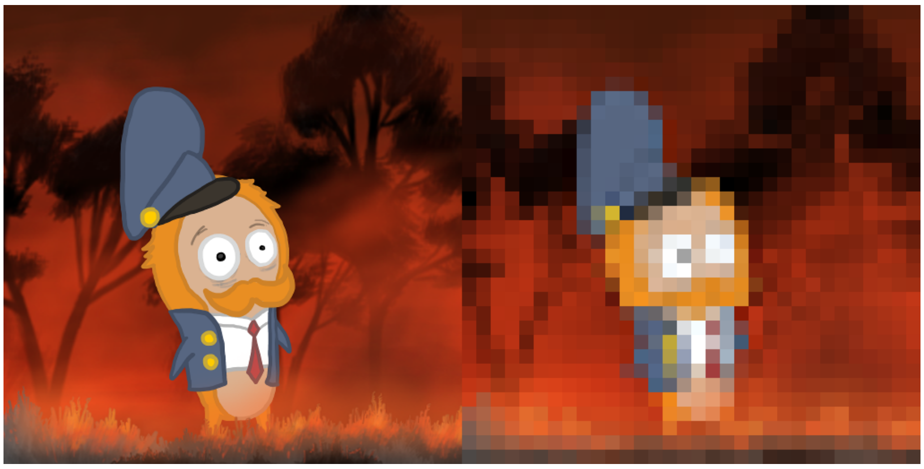

Figure 3.1: High resolution (left) versus low resolution (right)

Higher resolution texture images, so more detailled texture images, make for a better visual quality as seen in Figure 3.1. So to get the best visual quality possible, we need the highest resolution texture images as possible. Here we are limited by a few factors.

- The hardware limitation on texture sizes of the device's GPU will prevent us from loading in texture images which surpass this limitation.
- The Android heap size of the device will prevent us from allocating too much space for the texture image, which we will need for reading the image data from the image file.

But not only are we limited for the maximum resolution, we are also limited in which resolutions we can choose at all. In older hardware it was required for each texture image's dimensions to be a power of two. Meaning that

$$\exists n, m : width = 2^n \wedge height = 2^m$$

This requirement no longer holds true for more recent hardware however it does for some older smartphones. Also the OpenGL ES 2.0 documentation states:

> if the width or height of a texture image are not powers of two and either the GL_TEXTURE_MIN_FILTER is set to one of the functions that requires mipmaps or the GL_TEXTURE_WRAP_S or GL_TEXTURE_WRAP_T is not set to GL_CLAMP_TO_EDGE, then the texture image unit will return (R, G, B, A) = (0, 0, 0, 1).

So, if we were to chose textures that have non power of two dimensions, we are, if not directly restricted by hardware, restricted in our available options. For the sake of portability and leaving all our options open I will only use texture images which have power of two dimensions.

In order to determine the maximum texture size I tried loading different image sizes on three different devices. If a texture was loaded and displayed it means that the image could fit the heap and loaded to the device's GPU and is counted as succes. If an insufficient memory error was thrown or the image would not show the image was most likely too big and this is counted as a failure. The results can be seen in Table 3.1 and based on these results I chose for a maximum texture resolution of 2048 x 2048.

Device	512 x 512	1024 x 1024	2048 x 2048	4096 x 4096	8192 x 8192
HTC Legend	✓	✓	✓	×	×
Sinvigo R75	✓	✓	✓	×	×
Samsung GT-i9300	✓	✓	✓	✓	×

Table 3.1: Results of attempting to load several different resolution images on different devices. Succes means the texture was both loaded and displayed.

So our maximal achievable texture resolution is 2048 x 2048, and this will result in the highest Q_v in terms of texture resolution. However, when using such high resolution texture images might cause our rendering performance to decrease. Especially when looking at the concept of multiple layers, which might quickly increase the quantity of such textured objects on a single screen. Imagine a background image of the horizon, a mountain in the layer on top of that, trees on top of that and multiple other layers of trees to make the forest look deeper and finally some plants in the foreground - all high resolution textures. Now the first solution would be to see how much the rendering performance increases as we lower the texture resolution. From this results we could then find an optimal resolution - one which still provides sufficient visual detail but does not cost too much time to render. Note that this would ofcourse differ on other devices. A device with better hardware would probably be able to use higher resolution textures.

3.1.1 Method

In order to measure the effect of texture resolution on the render speed I used the following set-up. First I created a texture image with the chosen maximum of 2048 x 2048. I used photo editing software to create a set of this texture image scaled down to various resolutions. My final test set consisted of 11 the texture image in 11 different resolutions:

1. 64 x 64
2. 128 x 64
3. 128 x 128
4. 256 x 128
5. 256 x 256
6. 512 x 256
7. 512 x 512
8. 1024 x 512
9. 1024 x 1024
10. 2048 x 1024
11. 2048 x 2048

For the scene I used a rectange consisting of 4 vertices. This rectangle is fit exactly to the screen - meaning the top left vertex of the rectangle is the top left point of the screen and the bottom right vertex of the rectangle is the bottom right point of the screen. This way I know exactly how much pixels are being drawn on a single draw of this rectangle - $w_s \times h_s$, where w_s is the screen width and h_s is the screen height.

I draw this rectangle N times every frame to draw as many pixels per frame. This way the measurement of render speed in pixels per second is more accurate. The pseudo code for the drawing of the rectangle can be seen in Listing 3.1

Listing 3.1: Rectangle draw method

```
bind buffer objects;
bind texture;
start = currentTime(); //in milliseconds!
for(0 to N) {
    draw rectangle;
}
time = currentTime() - start;
```

So using this method the number of pixel drawn per frame is thus $w_s \times h_s \times N$ and the render speed in pixels per second as follows:

$$w_s \times h_s \times N \times (1000/(time))$$

To increase accuracy I measure the average time over 100 frames. This method is repeated for all of the chosen texture resolutions.

3.1.2 Results

The results of the time measurements with N = 100 on a Samsung GT-i9300 can be seen in Table 3.2 and the corresponding graph in Figure 3.2.

Resolution	Average time (ms)	Render speed (G pixels/s)
64 x 64	662.1	1.3919347531
128 x 64	667.8	1.3800539084
128 x 128	668.4	1.3788150808
256 x 128	669.3	1.376961004
256 x 256	673.9	1.3675619528
512 x 256	675.0	1.3653333333
512 x 512	676.2	1.3629103815
1024 x 512	681.4	1.3525095392
1024 x 1024	742.2	1.2417138238
2048 x 1024	1205.7	0.7643692461
2048 x 2048	2293.9	0.4017611927

Table 3.2

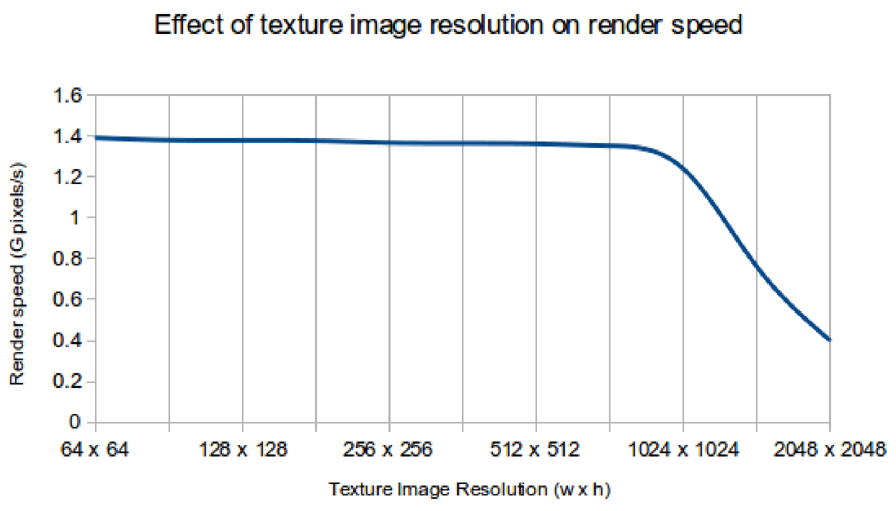

Figure 3.2: The effect of texture image resolution on the render speed

Surprisingly, the render speed seems fairly constant until we reach 1024 x 512. From this point, every increase in resolution results in a steep drop in render speed. Varying from 64 x 64 all the way up to 1024 x 512 results in a total decrease in render speed of only 2.832%, while taking only one more step up from 1024 x 512 to 1024 x 1024 the decrease in render speed over just that step is already 8.192%. This gets even worse with 43.485% from 1024 x 512 to 2048 x 1024 and 70.295% from 1024 x 512 to 2048 x 2048.

The step from 64 x 64 to 1024 x 512 results in an immense gain of visual quality, comparable with the difference portayed in Figure 3.1. The difference between 1024 x 512 to 1024 x 1024 is hardly noticable. The higher resolutions show no noticable differences at all.

3.1.3 Conclusion

As we can see in the results, texture resolution can negatively affect the render speed and thus Q_i. This is the case for textures higher than 1024 x 512. Ironically, from this same point the gain in visual quality Q_v is minimal to none. Now we could say that 1024 x 512 is the optimal texture resolution and should use it for all our object. There is however something we're overlooking.

The surface we draw our texture on remains constant size. As our surface covers the exact screen size of the mobile device we're testing on this constant size is the screen width x screen height of our testing device. So this 'constant' size is only constant for this particular device. So it is highly likely that our optimal 1024 x 512 resolution we so proudly found is not at all so optimal for other devices.

A quick run of our test on various other devices confirms this, as seen in Figure 3.3.

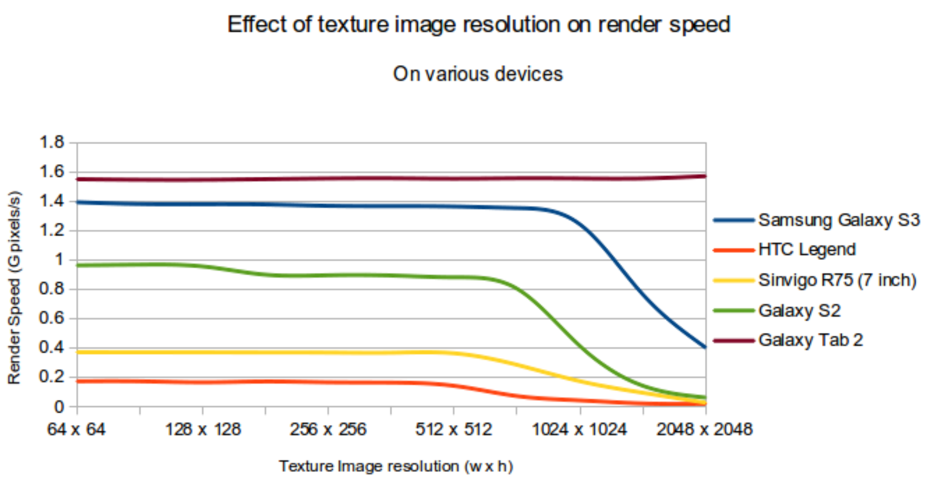

Figure 3.3: The effect of texture image resolution on the render speed on various devices

So we can conclude that a too high texture resolution can negatively affect Q_i and a too low texture resolution can negatively affect Q_v. There seems to be some optimal texture resolution for a certain surface size. A possible solution could be mipmaps, which we will discuss further in Section 3.2.

3.2 Texture Parameters

For each texture we can set various parameters with the glTexParameter function. The following four parameters can be changed:

- GL_TEXTURE_WRAP_S
- GL_TEXTURE_WRAP_T
- GL_TEXTURE_MIN_FILTER
- GL_TEXTURE_MAG_FILTER

WRAP_S & WRAP_T

The GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T parameters determine how a texture is wrapped over an object when the a texture coordinate falls outside of the [0,1] range. GL_TEXTURE_WRAP_S determines the behaviour for the texture coordinate s value (which corresponds to x in our 2D environment) and GL_TEXTURE_WRAP_T for the t value (which would then of course correspond to y). There are three different options for both of these parameters.

- GL_CLAMP_TO_EDGE
- GL_REPEAT
- GL_MIRRORED_REPEAT

The GL_CLAMP_TO_EDGE option will clamp any value which falls outside of the range. This basicly means any value < 0 will be set to 0 and any value > 1 will be set to 1. GL_REPEAT will repeat the values. This is done by simply ignoring the integer part of the value. Hence -1.67 will become 0.67 and 4.76 will become 0.76. GL_MIRRORED_REPEAT is similar to this but it will check wether the integer part is odd or even. If it's odd the value is set to 1 minus the fractional part of value if it is even is will be the same as GL_REPEAT. This way the repeating texture is also being mirrored.

For normal texture images we would not have any particular use for these texture images as our texture coordinates only range from 0 to 1. Even if we would use them, it would be for such specific purposes that there is no choice but to use it. Therefore I will not study this any further.

MIN_FILTER & MAG_FILTER

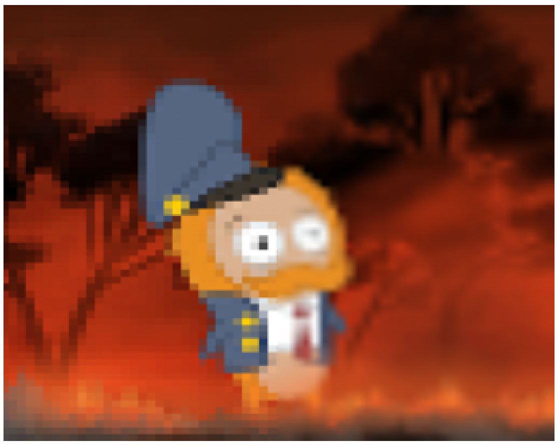

Figure 3.4: Difference between NEAREST (left half) and LINEAR (right half)

The GL_TEXTURE_MIN_FILTER and GL_TEXTURE_MAG_FILTER specify which minify and magnify filters to use respectively. These determine how samples are derived from the texture image when the surface on which it is drawn is smaller or larger than the texture respectively.

For the GL_TEXTURE_MAG_FILTER parameter there are the following options:

- GL_NEAREST
- GL_LINEAR

The GL_TEXTURE_MIN_FILTER parameter can be set to one of the following options:

- GL_NEAREST
- GL_LINEAR
- GL_NEAREST_MIPMAP_NEAREST
- GL_LINEAR_MIPMAP_NEAREST

- GL_NEAREST_MIPMAP_LINEAR
- GL_LINEAR_MIPMAP_LINEAR

GL_NEAREST will simply use the nearest texture element to the center of the fragment being rendered. GL_LINEAR will find the four nearest texture elements to the center of the fragment and use the weighted average. The visual difference between these two filters is displayed in Figure 3.4. GL_NEAREST is noticably pixelated so Q_v is higher for GL_LINEAR. In all previous test we used the GL_LINEAR filter for both minify and magnify.

The other options, which are only minify filters, are used for a technique called mipmapping.

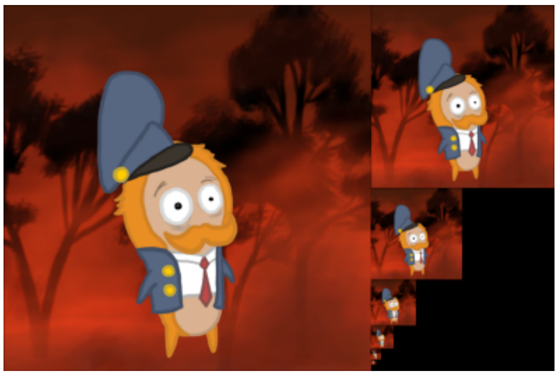

Figure 3.5: Example of a mipmap texture

A mipmap is a pre-calculated set of images. The set conists of the main image and versions of this image in lower resolution. For example a mipmap of a 256 x 256 resolution texture image would consist of this image and versions of 128 x 128, 64 x 64 ... 4 x 4, 2 x 2 and 1 x 1. An example can be seen in Figure 3.5. When we have generated such a mipmap, or rather have OpenGL generate such a mipmap, we can use the mipmap options for the GL_TEXTURE_MIN_FILTER parameter.

- GL_NEAREST_MIPMAP_NEAREST will first find the texture in the mipmap which most closely matches the drawing surface and then takes a texture element from this texture in a GL_NEAREST fashion.
- GL_LINEAR_MIPMAP_NEAREST will first find the two texture which most closely match the drawing surface, take a texture element from both in a GL_NEAREST fashion and use the weighted average of these two values.
- GL_NEAREST_MIPMAP_LINEAR will find the texture in the mipmap which most closely matches the drawing surface and takes a texture element from this texture in a GL_LINEAR fashion.
- And GL_LINEAR_MIPMAP_LINEAR will first find the two texture which most closely match the drawing surface, take a texture element from both in a GL_LINEAR fashion and use the weighted average of these two values.

3.2.1 Method

We will test the following texture parameters:

- GL_TEXTURE_MIN_FILTER
- GL_TEXTURE_MAG_FILTER

We will look at the following options for these filters:

- GL_NEAREST
- GL_LINEAR
- GL_NEAREST_MIPMAP_NEAREST (MIN only)
- GL_LINEAR_MIPMAP_NEAREST (MIN only)
- GL_NEAREST_MIPMAP_LINEAR (MIN only)
- GL_LINEAR_MIPMAP_LINEAR (MIN only)

In order to test the difference in render speed for GL_NEAREST and GL_LINEAR we will run the same test as described in Section 3.1.1 another time, but change both the GL_TEXTURE_MIN_FILTER and the GL_-TEXTURE_MAX_FILTER to GL_NEAREST. This way GL_NEAREST is used for both the high resolution textures, which have to be minified, as the low resolution textures, which have to be magnified. We can compare these results to the results we already found in Section 3.1.2, where we only used GL_LINEAR.

To test the mipmap option for GL_TEXTURE_MIN_FILTER there is no use in running the previous test, as the different resolutions are present in the mipmap. Instead we will let OpenGL generate such a mipmap from our maximum resolution texture (2048 x 2048). Then instead of drawing all different resolution textures, we only need to draw the mipmap with the different mipmap options for GL_TEXTURE_MIN_FILTER and measure the time for this.

3.2.2 Results

GL_NEAREST vs. GL_LINEAR

Resolution	Render speed GL_LINEAR	GL_NEAREST
64 x 64	1.3919347531	1.3819163293
128 x 64	1.3800539084	1.3781965007
128 x 128	1.3788150808	1.3804673457
256 x 128	1.376961004	1.3767553033
256 x 256	1.3675619528	1.3792277761
512 x 256	1.3653333333	1.3683741648
512 x 512	1.3629103815	1.3679679383
1024 x 512	1.3525095392	1.3586908448
1024 x 1024	1.2417138238	1.2785620915
2048 x 1024	0.7643692461	0.792842395
2048 x 2048	0.4017611927	0.434204947

Table 3.3

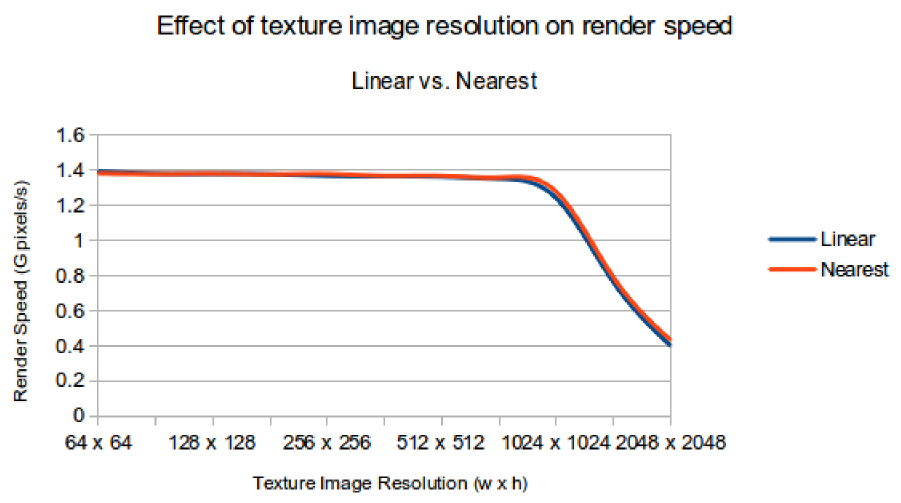

Figure 3.6: The effect of texture image resolution on the render speed, for both GL_LINEAR and GL_NEAREST as GL_TEXTURE_MIN_FILTER

Mipmapping

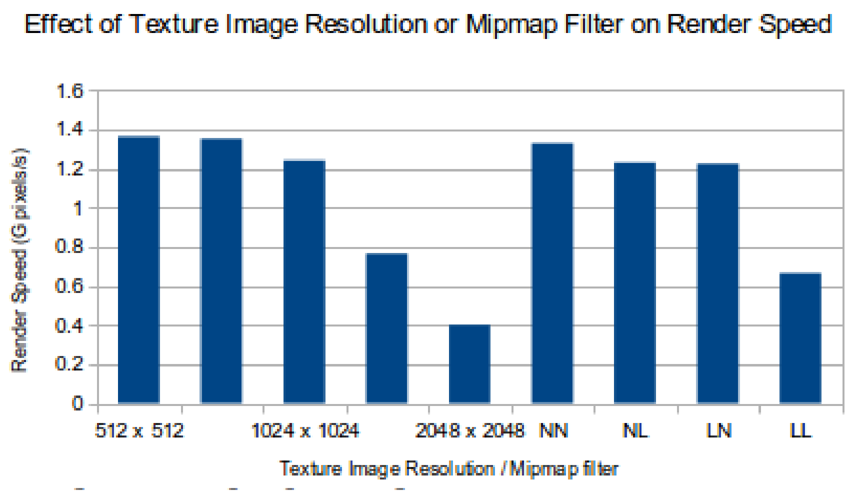

Figure 3.7: Effect of different mipmap filters on render speed

3.2.3 Conclusion

There is no significant difference in speed between using GL_LINEAR and GL_NEAREST. As stated earlier in this section, Q_v is higher for GL_LINEAR. Therefore, we will use GL_LINEAR for the GL_TEXTURE_MAG_FILTER.

The results for mipmapping show that this is a very good way to automatically find the optimal texture size mentioned in Section 3.1.2. GL_NEAREST_-MIPMAP_NEAREST though the fastest, results in noticable visual artefacts. This is because in essence, once it finds the nearest texture it is the same as GL_NEAREST. The other three options show no noticable visual differences.

Since GL_LINEAR_MIPMAP_LINEAR is significantly slower the best options for the mipmap filter are GL_LINEAR_MIPMAP_NEAREST and GL_-NEAREST_MIPMAP_LINEAR.

3.3 Alpha Blending

The blended color is calculated using a blending equation:

$$C_r = C_s \times S + C_d \times D$$

Where C_r is the resulting color, C_s is the new incoming color, the *source color* and C_d is the old existing color, the *destination color*. S and D are the source and destination factors respectively and can be set to various values. These factors determine how much each color weighs in the equation. Colors are treated as four-dimensional vectors with a dimension for each of the red, green, blue and alpha values. So we can also write the blending equation as follows, if:

$$C_s = (R_s, G_s, B_s, A_s)$$
$$C_d = (R_d, G_d, B_d, A_d)$$
$$S = (S_r, S_g, S_b, S_a)$$
$$D = (D_r, D_g, D_b, D_a)$$

Then:

$$C_r = (R_sS_r + R_dD_r, G_sS_g + G_dD_g, B_sS_b + B_dD_b, A_sS_a + A_dD_a)$$

Now imagine with blending enabled this blending equation will have to be evaluated for every single pixel drawn to the screen. Each pixel might take a little longer to be drawn and hence this could potentially lead to a decrease in the overall rendering speed. The simplest solution would be to just disable blending. This is possible, however this creates several undesired complications.

Transparancy and semi-transparancy depend on blending. A new incoming transparant or semi-transparant source color should obviously not just simply replace the destination color. Instead the resulting color should be a blend as such that it would appear that one can see the destination color through the source color. There are many ways to achieve something like this with blending by choosing different values for S and D. Most often S is chosen as GL_SRC_ALPHA, which would look like:

$$S = (A_s, A_s, A_s, A_s)$$

This way the higher the alpha value, or the visibility so to speak, of the source color, the higher the weight of this color in the equation, the more of the color you would see in the resulting color and vice versa. The rest of the total weight, GL_ONE_MINUS_SRC_ALPHA, would be given to the destination color:

$$D = (1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s)$$

A possible workaround which would allow us to disable blending is alpha testing. Alpha testing is testing each new incoming source pixel for it's alpha value. If this value does not meet a given set of requirements the pixel is simply discarded and the old existing color remains. This way we can for example eliminate pixel values with alpha < 1.0 and we have a transparency effect without having blending enabled. Another plus side is that if this were to work successfully we would no longer need to sort the objects before drawing but we can rely on OpenGL's depth (Z-) buffer. This because the color is simply discarded so the depth buffer would not be written with a value. However this technique would yield some poor results for semi-transparent pixels in the texture. Where these pixel colors would normally, and are expected to, blend with the existing color to make it seem you can see through it, these colors are now either discarded or shown fully opaque. This could result in some strange visual artifacts. This is only a solution for objects without any semi-transparent pixels. Considering these cons, I will not test this.

We could also eliminate all alpha pixels by using more vertices to completely define the shape, instead of letting the visual shape of an object depend on it's alpha pixels. Some algorithm that constructs the vertices for such a shape would have to be implemented. This would however potentially lead to very complex shapes. In case of animations every frame would now also need their own vertex buffer - as normally they are just two different image mapped upon the same rectangle. Also, this does not solve the problem for semi-transparent pixels in the texture image. Considering this, I will not test this.

So we can not completely turn of blending. We could however flag objects with completely opaque rectangular textures and only disable blending for such objects, as they don't need it. This way if it turns out to be faster, Q_g would improve as a subset of S would render faster.

3.3.1 Method

Using the same scene as described in Section 3.1.1, using a 2048 x 2048 texture I measured the render speed for both blending enabled and blending disabled.The device I used for testing was a Samsung GT-i9300.

3.3.2 Results

Blending	Average time (ms)	Render speed (G pixels/s)
On	23696	0.3889264011
Off	23410	0.3936779154

Table 3.4

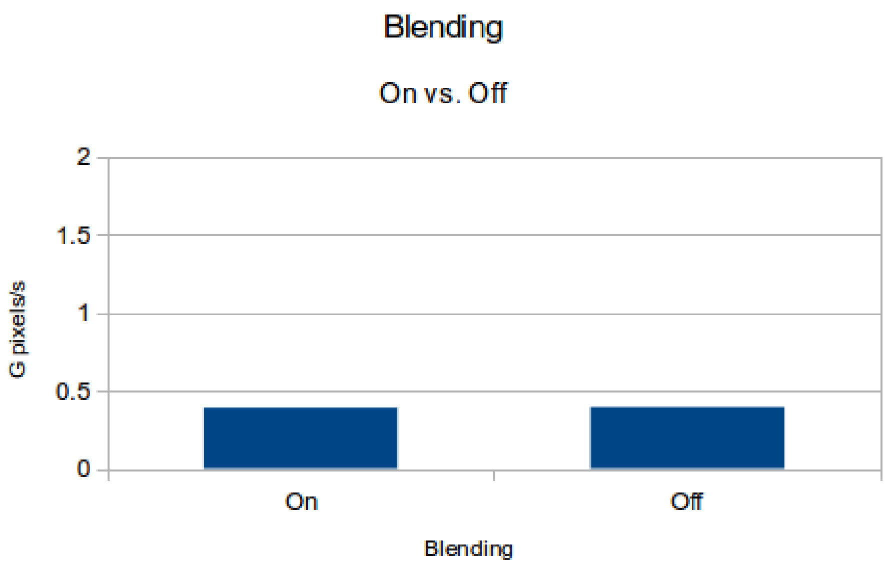

Figure 3.8: Render speed for both blending enabled and disabled

We see a 1.222 % increase in speed when turning off blending.

3.3.3 Conclusion

The results show only 1.222 % increase with blending turned off, even though the number of calculations has decreased drasticly. Most likely there is some built in optimalisation for blending - maybe even hardware.

So it is best to just turn blending on. The very small increase in speed is not worth of changing the entire structure of the graphics engine for. If we were to flag all rectangle opaque images and switch blending on and off the 1 percent increase would probably already be lost due to the time this would take.

3.4 Texture Format

The default way of loading a texture image on Android is using the BitmapFactory class which will return an instance of Bitmap object. The Bitmap object is basicly a wrapper class for a 2D bit signed integer table (2D array) representing the texture image. The dimensions of this table are $w \times h$ where w is the width of the image and h is the height of the image. So each integer entry within this table represents a single pixel of the image. Within this 32-bit integer the red, green, blue and alpha values of a single pixel are encoded so to speak. The integer is basicly divided in four, with each color value occupying 8 bits (or 1 byte) ordered red, green, blue, alpha. So the texture image is stored in a 4 bytes (32-bit) RGBA format. I will refer to this format as the texture image's external format.

A texture image can be stored internally in a different format than it's external format. This is called the internal format of the texture image. OpenGL ES 2.0 support five different internal formats which can be seen in Table 3.5.

Base internal format	RGBA	Internal Components
ALPHA	A	*A*
LUMINANCE	R	*L*
LUMINANCE_ALPHA	R, A	*L, A*
RGB	R, G, B	*R, G, B*
RGBA	R, G, B, A	*R, G, B, A*

Table 3.5: The different internal texture formats and their components which are supported by OpenGL ES2.0

Here we can see that RGBA stores the most components totalling 4 bytes, if we assume every internal component to be a single unsigned byte (Which is quite likely so). ALPHA and LUMINANCE only store a single component hence only a single byte. This reduction might also reduce the time it takes for the fragment shader performs a texture lookup - as less data has to be looked through and transfered. Therefore it might be interesting to analyse wether or not we can represent our textures in a more efficient format and wether or not this will result in a significant increase in render speed. First we will look at the different internal formats to discuss wether or not we can use this format.

ALPHA

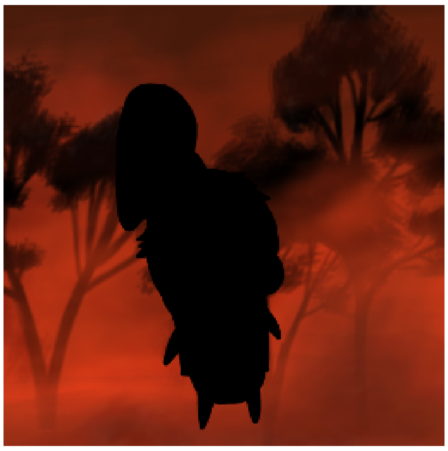
Figure 3.9: A texture in ALPHA format.

The ALPHA format stores only the alpha value of each texture element. This alpha value A_t represents the visibility of this texture element. When the fragment shader performs a lookup on a texture stored in this format the resulting color C_l is $(0,0,0,A_t)$. Thus the colors will range from $(0,0,0,0)$, which is fully transparent black, to $(0,0,0,1)$, which is fully opaque black.

This format stores only a single unsigned byte per texture element making it the smallest format available tying with LUMINANCE. However, using this format will cause us to lose all of our red, green and blue values. This is undesirable for most of our texture images. Textures with colors solely within the color range of this format might still be stored in this format. One could think of shadow textures as a possible example that could be stored in this format. However, for a simple platform game I can not think of possible applications of this format. It is potentially a lot faster than RGBA with 4 times a decrease in size, but so does LUMINANCE - which offers us more useful applications. For now I will discard the ALPHA format.

LUMINANCE and LUMINANCE_ALPHA

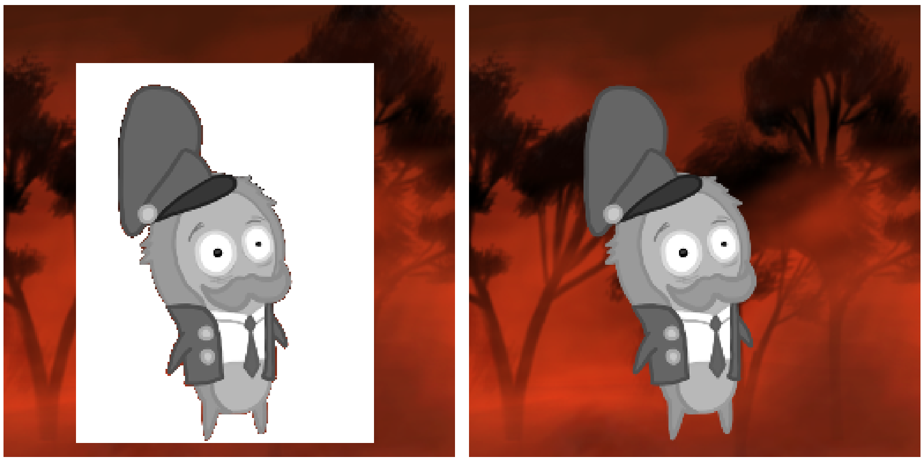

Figure 3.10: A texture in LUMINANCE format (left) and a texture in LUMINANCE_ALPHA format (right).

The LUMINANCE format stores only the luminance value of each texture element. This luminance value L_t with $L_t \in \mathbb{R}+$ between 0 and 1 representing the luminous intensity of this texture element. When the fragment shader performs a texture lookup on a texture stored in this format the resulting color C_l is $(L_t, L_t, L_t, 1)$. Thus the colors will range from $(0, 0, 0, 1)$, which is fully opaque black, to $(1, 1, 1, 1)$, which is fully opaque white. The colors in between are all greyscale (black and white) colors. Along with the ALPHA format this is the smallest format, but once again we lose too much color information to store most of our texture images in this format. Textures with solely black and white colors could be stored in this format however we will also lose the alpha value, so we will lose all transparency and semi-transparency in the image. Therefore we can only store a texture image in LUMINANCE format if this texture only contains black and white colors and for all of those colors alpha $= 1$.

This transparency issue can be adressed by using LUMINANCE_ALPHA format instead. This will store the luminance value of each texture element along with it's alpha value.

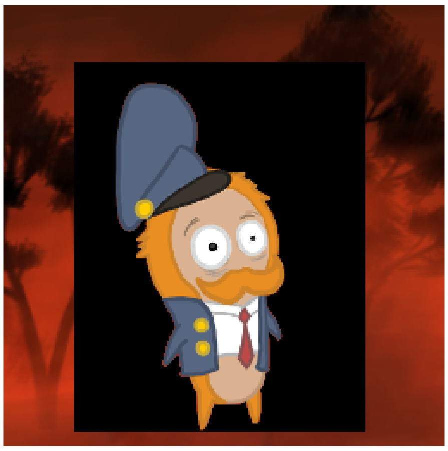

Figure 3.11: A texture in RGB format.

RGB

The RGB format stores the red, green and blue values of each texture element. These red, green and blue values will be equivalent to those in the original format R_t, G_t and B_t. Only the alpha channel is lost. The resulting color for a texture lookup in the fragment shader will be $(R_t, G_t, B_t, 1)$. Because of this transparent or semi-transparent texture elements will now be completely visible. This can clearly be seen in Figure 3.12, where the fully transparent pixels of the texture have become black - $(0, 0, 0, 0)$ to $(0, 0, 0, 1)$. This results in the same problem with transparency as the LUMINANCE format does. So this format can only be used for textures that only consist of fully opaque texture elements.

RGBA

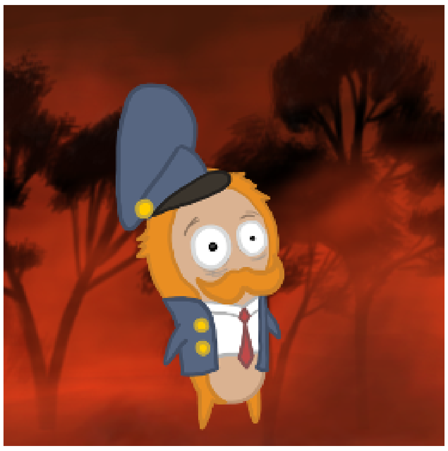
Figure 3.12: A texture in RGBA format.

RGBA is identical to the external format. This is the default value for loading in textures in OpenGL ES2.0 on Android. This format is the largest format among the possibilities, 4 bytes, but even though it may not decrease our rendering speed there is a pro to using this format. There is no conversion needed as the texture image is already in this format in the Bitmap class so this will decrease the loading time on startup. Also, there is no loss in texture data. As the previous discussed texture formats are limited to very specific textures.

LUMINANCE only black and white, no alpha

LUMINANCE_ALPHA only black and white

RGB no alpha

As discussed earlier most of our objects depend on their alpha value for their shape (ALPHA BLENDING REFERENCE HERE). So in most cases we can only use RGBA. However, if the other formats prove significantly faster then it might be worth it to parametrize this option for the scene designer. This way Q_g will still increase higher as $S_x \subset S$, the set of all scenes with textures which can use these smaller formats, will render faster.

3.4.1 Method

We will measure the performance of the following texture formats:

- LUMINANCE
- LUMINANCE_ALPHA
- RGB

- RGBA

To do this we use a method similar to the one described in Section 3.1.1, but instead of varying the texture resolution, we will vary the texture format. The texture format can not be changed during run-time, but has to be set when the texture is loaded, so with varying the texture format we mean switching between four instances of the same texture each loaded with a different format.

As the default format is RGBA and OpenGL ES2.0 does not provide a conversion mechanism, we have also added some functions to convert the RGBA texture to the needed internal formats. The time to convert these textures is not included in the results as we are only interested in the gain in speed during the actual game.

3.4.2 Results

Blending	Average time (ms)	Render speed (G pixels/s)
RGBA	6674	1.3808810309
RGB	6734	1.3685773686
LUMINANCE_ALPHA	6256	1.4731457801
LUMINANCE	6090	1.5133004926

Table 3.6

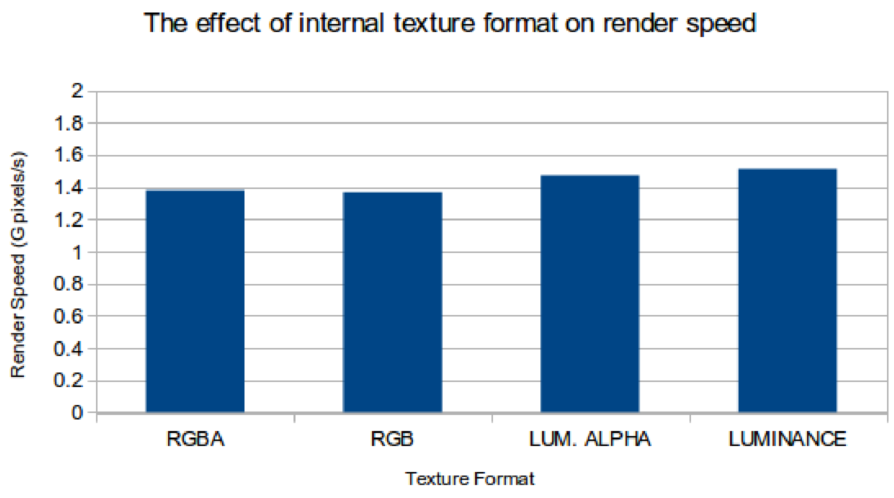

Figure 3.13: The effect of internal texture format on the render speed

No significant difference between RGBA, the original format, and RGB - though strangely a small decrease in render speed (0.891%). Both the LUMINANCE formats show better results. From RGBA to LUMINANCE_ALPHA shows a 6.682% increase in render speed and to LUMINANCE even 9.589%.

3.4.3 Conclusion

The results show that the LUMINANCE_ALPHA and LUMINANCE formats render only a little faster - but enough to be interesting. Because even though there is only a slight gain in Q_i there is no loss in Q_v whatsoever, so there is no reason not to use these formats.

The downside is that these formats can only be used on very specific textures and conversion is needed which could take long. The latter could be solved by already storing these textures externally in the LUMINANCE format.

3.5 Depth Testing

If we want things on the screen to appear in a certain order, for example grass is in front of the player and trees are behind him, then we have to draw the objects in this order - from back to front. Otherwise the tree would for example just be drawn over the player - overwriting his pixels in the color buffer. To maintain such a drawing order we need some sort of sorting mechanism. However, sorting all the objects in the game could be quite a hit on performance - especially when new objects spawn a few times every frame.

As mentioned in Section 2.3.3 we could try to abuse OpenGL's depth testing to prevent us from having to sort our objects ourself. We could easily enable depth testing use the depth value of every object as the z-coordinate. The depth test will now discard any incoming pixels on that position in the buffer with a higher depth.

However, there is one problem. Depth testing does not work well with blending. As a transparent pixel is drawn, it does write it's depth in the depth value. Therefore, any other pixel to be drawn behind the transparent pixel will be discarded by the depth test. However, this new pixel should be visible as the previous one was transparent.

If we were to turn of blending it should work. However, we concluded in Section 3.3.3 that we can not simply turn off blending in our context.

3.5.1 Conclusion

Since depth testing conflicts with blending, and we already concluded that we can not turn of blending there is no way we can use depth testing to do our sorting for us.

Chapter 4

Conclusion & Discussion

We have studied, analyzed and tested the following factors of a graphics engine:

- Texture Resolution
- Texture Format
- Texture Parameters
- Blending
- Depth testing

For each of these factors we have drawn a conclusion on wether or not they could be tweaked to gain performance while maintaining a high visual quality. And if yes, how one could do so.

- Texture resolution only start to significantly affect the render speed negatively when the resolution is higher than some value. This value depends on the size of the drawing surface. As texture resolutions higher than that value would ultimately be mapped on the smaller surface, there is minimal to no gain in Q_v to use any higher resolution textures than needed. To find the optimal resolution for a texture given a surface we propose mipmapping as a solution.
- Wether using GL_LINEAR or GL_NEAREST as the minify or magnify filter for a texture makes no difference in performance. However, the visual quality for GL_NEAREST is considerably lower than that of GL_LINEAR. Therefore we will use GL_LINEAR as our magnify function. For our minify filter we also looked at various mipmap filters. GL_NEAREST_MIPMAP_NEAREST is the fastest though shows noticable visual artefacts. The other three options show no noticable visual differences. Since GL_LINEAR_MIPMAP_LINEAR is significantly slower the best options for the mipmap filter are GL_LINEAR_-MIPMAP_NEAREST and GL_NEAREST_MIPMAP_LINEAR.
- Wether blending is turned on or off makes no significant difference for the performance. Turning it off causes a lot of undesired complications. Therefore we will leave blending turned on.

- A little speed can be gained by storing textures in smaller format such as LUMINANCE and LUMINANCE_ALPHA. However, this optimalisation can only be done for a very select number of textures, which have limited colors and thus fit in such format. Also, conversion is required.

- We can not use depth testing to do out depth sorting for us, as it conflicts with blending. As we concluded, blending can not be turned off in our context. Therefore, we should sort our objects ourself.

So overall we see that we can gain some speed, but not very much. High resolution textures and blending are required to maintain a high visual quality. We can improve some in the area of high resolution textures by using mipmaps and storing them in smaller formats if possible. In the context of such a simple 2D platform game there are no other ways to significantly improve the graphics engine for such a game. We are now limited by the hardware of the device in which scenes we can draw.

Even though 2D platform games are much simpler than 3D games, optimalisation of such a game proofs diffucult as there are so few options which we can change to gain speed without losing visual quality.

4.0.2 Limitations

The problems we analyzed only cover a simple 2D platform game concept. However, most 2D platform games use complex techniques on top of this basic concept to create richer scenes and a higher visual quality, for example animations and post-render effects. These techniques each come with new and unique challenges and issues, which are not covered in this thesis. These techniques are more complex and could provide more dimensions which we can optimize.

The graphics engine we analyzed and tested is using the OpenGL ES 2.0 API. Other graphics API's or other versions of the OpenGL ES API might offer different or more functionality, which might offer more dimensions for optimalization.

4.0.3 Future work

There is not much that can be optimized on the graphics engine for MAGE as concluded in this thesis. For further optimalization we can only look at the other aspects in the game engine, such as the game logic or the physics engine.

We will also expand MAGE to support animations and various post-render effects, including for example shadow and lighting effects. As mentioned in the Limitations section this will offer new and more complex challenges and more dimensions for optimalization.

Bibliography

[1] Daniel Boutros. A detailed cross-examination of yesterday and today's best-selling platform games. `http://www.gamasutra.com/view/feature/1851/a_detailed_crossexamination_of_.php`, August 2004.

[2] Gamesradar. Gaming's most important evolutions. `http://www.gamesradar.com/gamings-most-important-evolutions/`, October 2010.

[3] Google Play Store. `https://play.google.com/store/apps`.

[4] Kyle Hilliard. Apple Awards Rayman Jungle Run Game Of The Year. `http://www.gameinformer.com/b/news/archive/2012/12/13/apple-awards-rayman-jungle-run-game-of-the-year.aspx`, December 2012s.

[5] Ingrid Lunden. Angry birds maker rovio says 2012 sales up 101% to $ 195m with merchandising, ip 45% of that; net profit $ 71m. `http://techcrunch.com/2013/04/03/rovios-revenues-up-101-to-195m-non-games-45-of-that-net-profit-71m/`, April 2013.

[6] OpenGL Rendering Pipeline Overview. `http://www.opengl.org/wiki/Rendering_Pipeline_Overview`.

[7] Wybe Schutte. Mobile Games Trend Report. `http://www.newzoo.com/trend-reports/mobile-games-trend-report/`, March 2012.