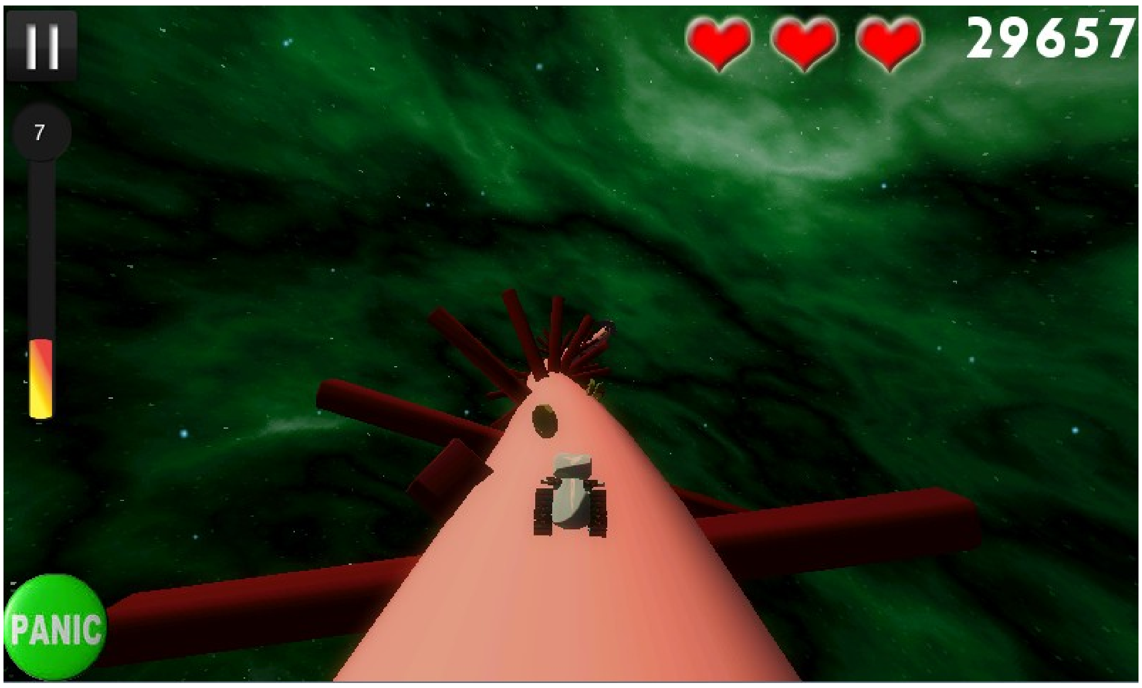
CHALMERS



Tisado

– A graphics intense smartphone game

Bachelor's Thesis
Computer Science and Engineering Programme

GUSTAV BODARE
ANN MICHELSEN
EDVARD SANDBERG
ROBERT SILVERFLOD
DAVID TERMANDER

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2012
Bachelor's Thesis DATX12-37 - Graphics intense smartphone game

Abstract

This bachelor's thesis describes the design- and implementation process of a 3D smartphone game, where the developers are restricted by both time and hardware limitations. It explores the various aspects of creating a modern game, including effective real-time rendering and the creation of a 3D world, as well as related fields, such as collision detection, optimization techniques and visual effects.

The thesis investigates the most commonly used techniques for developing a fun and visually appealing smartphone game, and presents our implementations and solutions to the obstacles facing the modern game developer.

Acknowledgements

First of all, we would like to thank Mario Zechner for creating libgdx, the most awesome framework. Ever. We would also like to thank a few other people who helped making this project such an enjoyable journey for us. In particular, we would like to thank Kalle Hämäläinen for his incredible patience and much needed help with everything that comes with game development. We would also like to thank Stefan Bachmann for all the time and explanations, as well as the other frequent users of the #badlogic irc channel.

Contents

1 Introduction

The smartphone applications industry is a relatively new one, but it is already a big market. There are companies that focus solely on developing and releasing smartphone applications, and even individual developers can make a living by creating such applications. There are two main markets for mobile applications today; AppStore for products developed for devices running iOS, e.g. iPhone and iPad, and Google Play for Android products. Android holds 58.8% of the market shares and iOS holds 32.2%. The rest of the market shares are split between lesser companies [1].

The development of mobile hardware is rapidly moving forward. This means that there are often new devices with stronger hardware than those previously released. Thus, making room for applications with more intense graphics and logic. Since there are always better looking and more advanced games being developed, most games have a short lifespan.

The possibility to help develop an interesting market and participate in the development of a growing community of game developers has been a major source of inspiration for us.

1.1 Purpose

The purpose of this project is to, within the limit of four months, design and implement a visually appealing, entertaining and graphically intense smartphone game. To permit the focus of the project to target the graphics, the game logic has to be relatively simple, whereas the visuals can be successively enhanced. Additionally, the intention is that our resulting product will be adequate for commercial release.

Furthermore, the various aspects of creating a modern smartphone game will be explored, such as modeling, animating and rendering a 3D world, as well as implementing sound and visual effects. Due to our target platform, a smartphone, there will unavoidably be a heavy focus on optimization techniques to keep a high graphical standard while avoiding performance issues.

The purpose of the report itself is to present solutions to problems such as:

- How do we make a fun and visually appealing game, for a platform we have no experience with, in less than four months?
- How can we fit as many graphical effects as possible in the limited hardware of a smartphone?

We hope that the answers to these questions can provide a good foundation for people with interest in both game development in general and, more specifically, developing smartphones games.

1.2 Limititations

The project was limited both by time, it had to be done within four months, and by personnel, five students working fifty percent part-time. To be successful in producing a full fledged smartphone game within that time frame, certain delimitations had to be made. As our game was focused on graphical prowess, the game logic had to be limited for the sake of project completion. The decision was made early on that no physics engine was to be created. Instead, an illusion of a physics engine would be implemented through simpler means. Also, due to the technical nature of the project, and the high time consumption of making animations, interesting graphical effects were prioritized over numerous animations.

Since smartphones generally lack GPU processing power in comparison with desktop computers, limitations to the rendering structure had to be made. As an example, the amount of objects simultaneously displayed on the screen had to be revised at several points during the project. Other delimitations, such as the exclusion of various otherwise desirable visual effects, had to be devised for the same reason.

1.3 Method

Early in the development of the project, we had to decide upon a suitable game idea. Each project participant came up with at least one idea for a game, and these ideas were discussed and rated. Out of these choices, one was selected and refined.

This idea was originally a game where the player controlled a man running on a pipe with increasing speed, avoiding obstacles and collecting coins. During the project, this changed from a man to a robot to reduce the complexity of the animations required.

When the game idea had been decided, we needed to choose a platform to develop for, a framework to use and how to make sure that the development kept moving forward.

1.3.1 Development Process

We chose to use an iterative developement method for our project, working with short deadlines, and weekly meetings to set new deadlines. The reason for using such a method was to better distribute the time needed for different parts of development. Features that turned out to be difficult to implement were brainstormed during these weekly meetings. Since none of us had any prior experience of working with a project of this magnitude, we tried to distribute every task to more than one person. We did this so that every group member always had someone to discuss their problems with.

To avoid problems with merging code, we knew that a version handling system would be required. We were provided with a Subversion server by Chalmers, making our choice of what system to use significantly easier. We used a plug in for Eclipse called Subclipse, allowing us to manage the repository from our server directly in Eclipse.

1.3.2 Choice of platform

Since there are several mobile platforms available to choose between, when developing a smartphone game, we needed to research these platforms to come up with the most suiting choice. This research involved reading hardware benchmark tests and investigating the programming languages available to develop the game with. In addition, we also considered, and highly valued, the preferences of the group members.

We agreed that the choice to be made stood solely between Android and iOS, due to the popularity of the devices. To achieve the advanced 3D graphics we desired, the use of OpenGL ES 2.0 was required. The benchmark tests we studied, by AndroidPolice[2] and ExtremeTech[3], declared that iPhone 4S is the strongest device available for OpenGL ES 2.0. However, these benchmarks also showed that the earlier devices using iOS were weaker than many of the most popular Android devices.

With this in mind, we discussed the programming languages and platform preferences of the project participants. Because Android can be developed in Java, a language all of us were familiar with, the development was likely to advance faster than it would with a new language such as Objective-C, the languade used for iPhone developement. Thus, if we chose to develop for Android platforms, more time would be available for actual game development and tuning.

After discussing the choices, the advantages and disadvantages of each platform, we decided that developing for Android was the most suited choice for us.

1.3.3 Choice of framework

A game development library can be used to quickly develop a game. Because of the limited time available for our project, we felt that it was a natural choice to make use of such a library. We looked at other developers' experience with frameworks for Android[4], and found a library called libgdx. This library allowed for development, as well as debugging, to be done on a desktop computer; something that would otherwise not be possible since the emulator that comes with the Android SDK does not support OpenGL ES 2.0 [5].

Libgdx is built upon several different back ends, allowing us to use the same Java code for deploying the application on different platforms, such as Windows, Linux, Mac OS, Android and HTML5(See Appendix A.1). It is also written with some JNI (Java Native Interface) for performance-critical sections [6]. None of the participants of the group were familiar with writing JNI and, again due to the time limit, it is not likely that we would have been able to write a faster framework ourselves.

Libgdx also comes with a large community. There is a forum, an official wiki, a well documented API and an active irc channel available for support and help with any problems that may arise in development. The forum contains a sub forum, focusing on collecting unofficial extensions written by the libgdx community. This forum provides extensions, such as bloom, which turned out to be useful for our project.

To finally make our decision, we looked at other games developed with libgdx to see how well they had succeeded [7]. Considering all these things, libgdx felt like a suiting choice of framework for us to develop our game with.

1.4 Outline of study

This report is structured into seven chapters; each discussing the different aspects of the project. Chapter 2 gives a brief introduction to the game Tisado, where the features and the ideas behind the game are presented. The following four chapters explores different aspects of the game. Each of these chapters gives a short introduction to the subject that is to be discussed. Thereafter, previous works in that subject is presented. Towards the end of each chapter, the results are presented. The last chapters are dedicated to results, discussion, and conclusion.

2 The Game Tisado

We have named our game Tisado, which is an acronym for "Travel in space and dodge objects". In the game, the player controls a robot, viewed from behind, traveling along a pipe (see Figure 1). The speed of the robot will increase during the game, and the player has to move to avoid obstacles, while collecting as many power ups and coins as possible. The player controls the robot by tilting the smartphone. The aim of the game is simple; travel as far as possible without colliding with any obstacles, and gain the highest possible score.

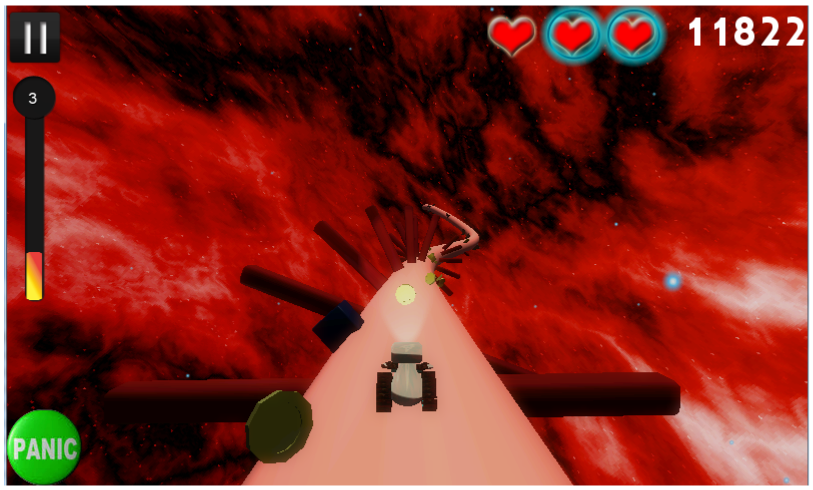


Figure 1: A beautiful screen shot from Tisado.

2.1 Features and ideas

In this section, we will present the features that were implemented in the game. We chose only to implement features that would enhance the experience, while not requiring too complex logic. In our implementation of the game, we built our classes around a main class. The main class controls all game states, network, screen updates and native functions (see Appendix A.2).

2.1.1 Jumping

By jumping, the player will have the opportunity to avoid obstacles that may otherwise be impossible to pass. However, some obstacles are too high to pass by jumping. This is something that requires the player's attention, to avoid collisions. Furthermore, the player has the possibility to collect coins and power ups when jumping. To initiate a jump, the player can simply press anywhere on the screen.

2.1.2 Power Ups

The game contains many different kinds of pick ups, such as coins, extra life, immortality and boost. Coins are the ones that will show up most frequently. By collecting a coin, the score increases. There is a coin bar at the left side of the screen, and each collected coin will add to the size of the bar. When the bar is full, the value of all subsequently collected coins will increase by ten percent. A sound effect is played every time the character collects a coin, and the coin quickly moves from the player's location to the coin bar (see Figure 2).

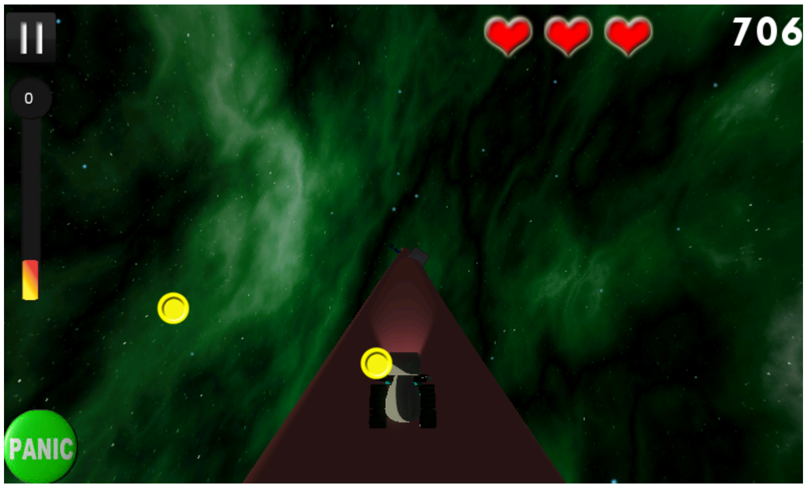


Figure 2: Coins flying towards the coin bar.

An immortality power up will grant the robot invincibility for five seconds. During this time, the robot will flicker and the color of the pipe will change into

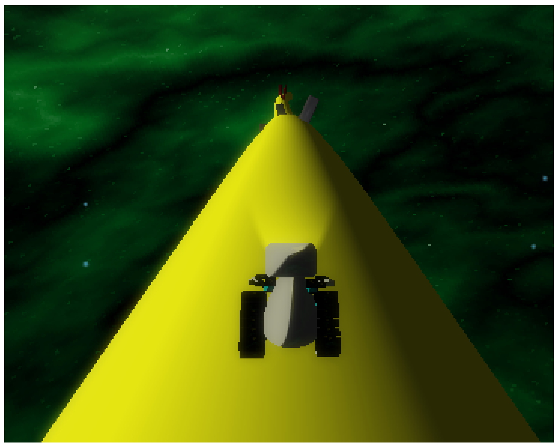


Figure 3: The immortal power up, pipe is now yellow.

yellow (see Figure 3). When the immortality expires, the pipe will change back to its normal color.

The life power up gives the character one extra shield (see Figure 4). The robot can only have as many shields as hearts. Thus, loosing a life will reduce the maximum amount of shields.

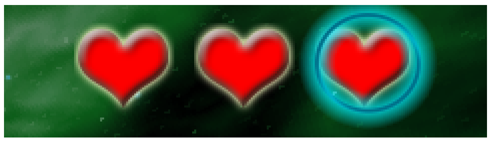


Figure 4: The shields, displayed in the user interface.

The boost power up grants both incredible speed and immortality for three seconds (see Figure 5), as well as turn the pipe blue.

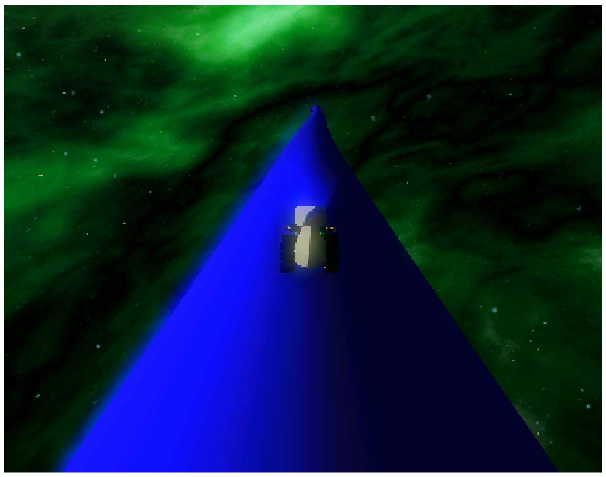


Figure 5: The boost power up, pipe is now blue and the bloom effect is increased.

2.1.3 Panic Button

We implemented what we call a panic button. When pressing the panic button, the game will enter slow motion (see Figure 6). It can be used when there are segments of obstacles that are particularly difficult or when a special power up is to be collected. The slow motion effect will last for three seconds, and has a long cool down to force the player to use the button wisely. The button is located at the bottom-left corner and when used, the green color will be drained quickly. The button will then slowly be filled as the cool down passes.

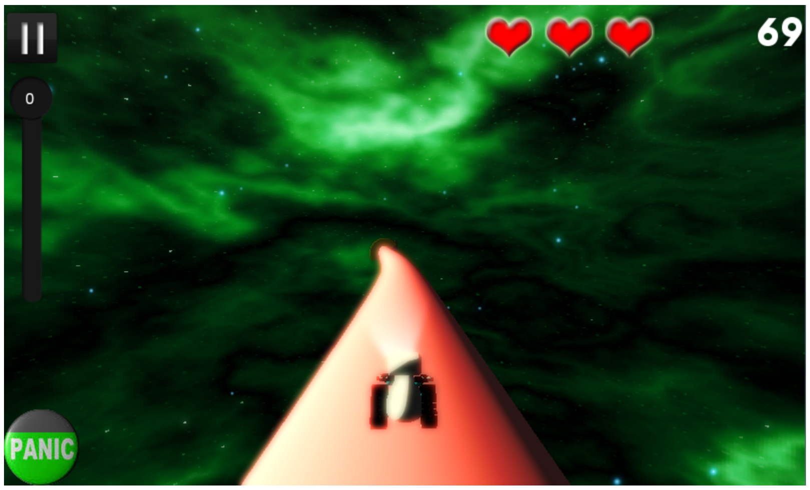


Figure 6: Slow motion effect with increased bloom.

2.1.4 Environment Objects

To make the world feel more alive, we added environment objects. There are two different kinds of environment objects in the game; stars(see Figure 7) and new level circles(see Figure 8).

The stars were added to the game to give a feel of speed. Stars fly towards the camera, with a speed relative to the speed of the character. Thus, when the character speed increases, for example by taking a boost power up, the stars will fly past even faster. This is done to make the increase of speed feel higher than it actually is. These stars were implemented with the use of particles. See section 3.3 Particle System, for more information.

The circles were added to show the player when a new level is about to start, and they have no other impact on the game other than adding to the visual experience. These circles were made by multiple rotating cubes positioned in the shape of a circle around the pipe.

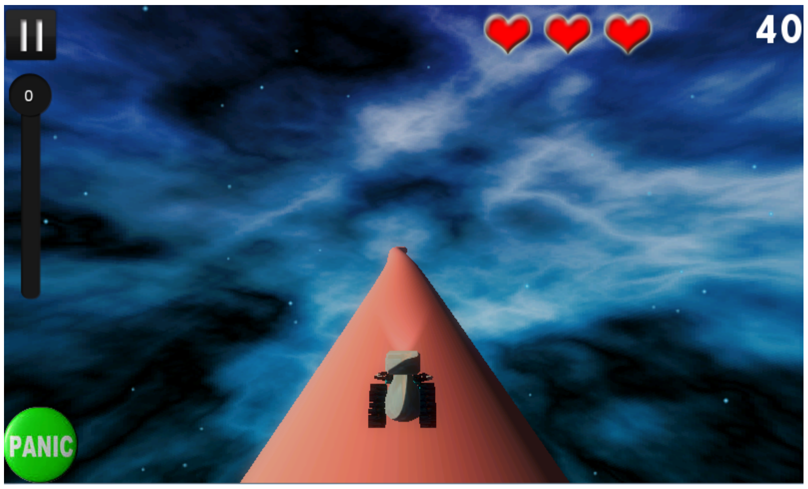


Figure 7: Stars, flying towards the camera.

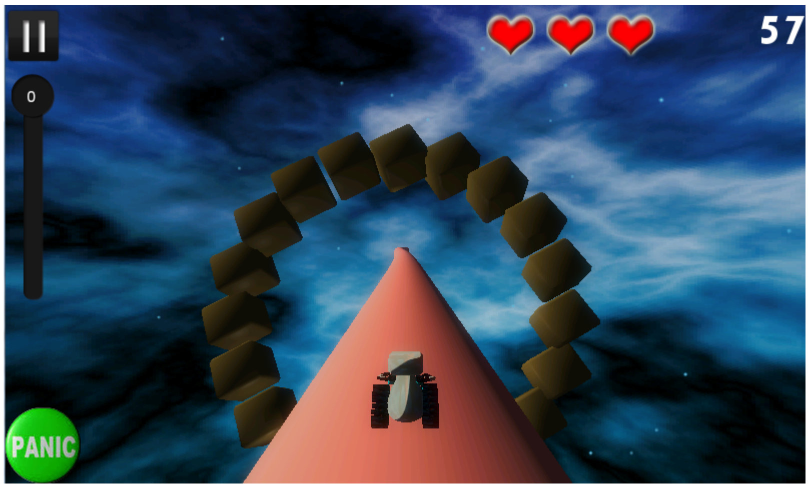


Figure 8: Rings made up of cubes, visualizing new levels.

2.1.5 Sky sphere

To create the feeling that the game takes place in space, we used a sky sphere (see Figure 9). The sky sphere will always be centered around the player to invoke the feeling that the environment is infinitely far away. The sphere is created and mapped with a texture in 3Ds Max. When a new level is reached, the texture will change with the use of bloom. See section 3.4.2 Bloom, for more information.

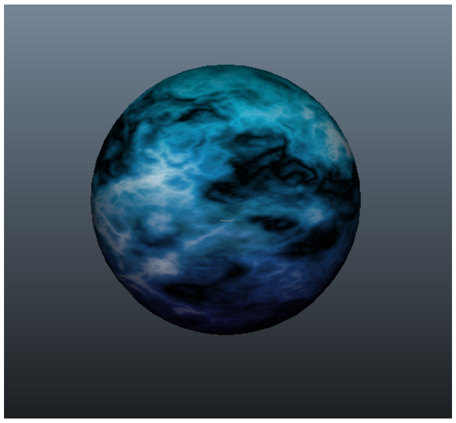
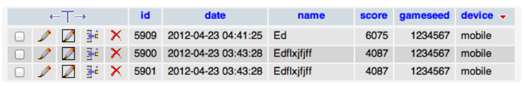


Figure 9: A sky sphere made with Maya.

2.1.6 Online highscore

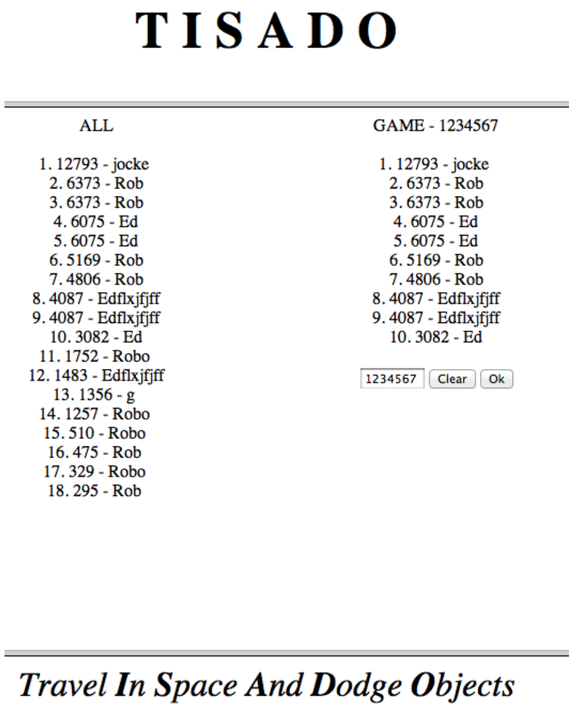
There is a local highscore, which will be updated after each game if the user qualifies, as well as an optional online highscore. If the user activates the online highscore in the settings, the game will connect to the online database server instead of the local one. We use a simple MySQL database to hold all the highscores(see Figure 10).



←T→	id	date	name	score	gameseed	device
	5909	2012-04-23 04:41:25	Ed	6075	1234567	mobile
	5900	2012-04-23 03:43:28	Edflxjfjff	4087	1234567	mobile
	5901	2012-04-23 03:43:28	Edflxjfjff	4087	1234567	mobile

Figure 10: Highscores in Tisado is saved in a MySQL database

The MySQL database is accessed from a PHP script that is running on a webserver which in turn is controlled by the android device via a HTML protocol (see Appendix B.3). The highscore is presented on the device by pressing the button 'highscore'. It is also presented live on the webpage 'http://www.tisado.se' (see Figure 11). On the android device, you can choose between showing the total highscore from all different kinds of games, or just from a specific game.



TISADO

ALL	GAME - 1234567
1. 12793 - jocke	1. 12793 - jocke
2. 6373 - Rob	2. 6373 - Rob
3. 6373 - Rob	3. 6373 - Rob
4. 6075 - Ed	4. 6075 - Ed
5. 6075 - Ed	5. 6075 - Ed
6. 5169 - Rob	6. 5169 - Rob
7. 4806 - Rob	7. 4806 - Rob
8. 4087 - Edflxjfjff	8. 4087 - Edflxjfjff
9. 4087 - Edflxjfjff	9. 4087 - Edflxjfjff
10. 3082 - Ed	10. 3082 - Ed
11. 1752 - Robo	
12. 1483 - Edflxjfjff	1234567 Clear Ok
13. 1356 - g	
14. 1257 - Robo	
15. 510 - Robo	
16. 475 - Rob	
17. 329 - Robo	
18. 295 - Rob	

Travel In Space And Dodge Objects

Figure 11: Screenshot from www.tisado.se - presenting total and specific gameseed highscore

3 Computer Graphics

The following chapter explores the various graphical ascpects of Tisado. It includes techniques for modeling, animating, and enlightening a 3D scene, as well as applying reflections, post-processing, and particle effects. Generally, each section supplies background information, investigates several possible implementation methods, followed by a presentation of our results.

3.1 Lighting

In this section, we will first discuss various methods for creating virtual lighting for a 3D scene. We will then describe our implementation of the shading in our game.

To simulate real world lighting in games, a technique called shading is often used. Basically, shading tries to resemble lighting by varying the level of darkness, depending on the angle and distance between the surface and one or more light sources. Usually, four different types of lighting are combined to create the final shading. These are ambient lighting, diffuse lighting, specular lighting, and emissive lighting [8]. There are several methods to perform shading, most of which results in different levels of realism, but also requiring different amounts of processing power. Three common shading techniques are flat shading, Gouraud shading, and Phong shading [9].

3.1.1 Shading Models

Flat shading simply uses the normal for each triangle of an object to calculate the same shade for all pixels within that plane. This results in very unrealistic looking objects, unless the polygon count is huge, though the complexity of calculating the lighting is fairly low [10]. Gouraud shading calculates the shading by using the normals for each vertex, and then interpolate the color over each pixel in between vertices. This gives more realistic lighting compared to the flat shading, although at a slightly higher computational price [11]. The most realistic lighting model, that we felt was viable for use in a game, is the previously mentioned Phong shading. It works similarly to Gouraud shading. However, instead of interpolating the colors, it interpolates the vertex normals and calculates the shading for each pixel. This technique is also sometimes referred to as per-pixel shading [12](see Figure 12).

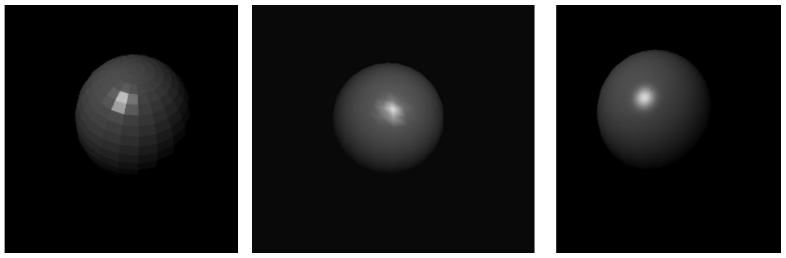


Figure 12: The shading qualities of flat shading, Gouraud shading, and Phong shading.

3.1.2 Light sources

There are usually several types of light sources used when rendering a scene. Three basic, and widely used types of light sources, are directional lighting, spotlight lighting, and point lighting (see Figure 13). They can all be implemented with the different shading models described in the previous section. Directional light illuminates objects from a certain direction rather than a position. In the real world, there is no light that behaves in this manner. However, it can simulate lighting that is very far away and could in fact be considered as infinitely far away. An example of this is the sun, which illuminates the ground from seemingly the same direction due to its vast distance from the earth. Point lights emmits light equally in all directions from an infinitely small point. Unlike the directional lighting, point lighting usually fades over the distance from the source. Spotlights are similar to point light. However, they only emit light in one direction in a cone like fashion[13].

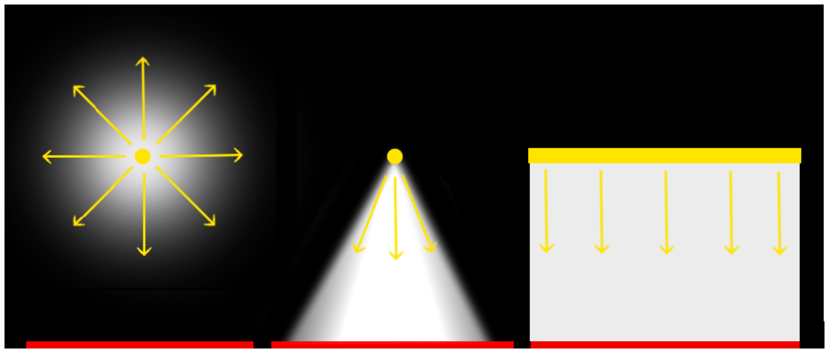


Figure 13: The various properties of point light, spot light, and directional light.

3.1.3 Shaders

Traditionally, computer graphics have been rendered by sending data through a fixed hardware pipeline, which then visualizes the data on the screen. The programmer was limited to a predefined amount of visual effects and shading techniques. During recent years, the hardware has started supporting the use of shader programs, more commonly known as shaders.

A shader is a program defined by the programmer and then run by the graphics hardware. This allows the user to implement graphics and visual effects with great precision and freedom, while also providing a good opportunity for optimization. The shader consists of two parts – a vertex shader and a fragment shader [14]. The purpose of the vertex shader is to calculate the position of the vertices for each object in the scene. The fragment shader is then invoked for all pixels within the triangle, to calculate the resulting color, usually by computing the lighting in accordance with one of the models described in section 3.1.1 Shading Modles. This is merely the basics of shader programs, as they can be used for many different purposes, such as post processing. See section 3.4 Post Processing, for more information.

3.1.4 Our lighting model and shaders

Since one of the goals for the project was to create high quality graphics, we decided to use the Phong shading model for all objects in the scene. Smartphone GPUs generally have lower processing power than desktop computers and thus, we had to limit the number of light sources and optimize the code to keep the rendering speed at an acceptable level. We implemented support for up to three light sources, where one of the lights used directional lighting, simulating a sun, one spotlight was used as light beams from the player, and one spotlight moved randomly in the scene to create a dynamic experience rather than actually representing a real light source. In Figure 14, our three light sources can each be seen one at a time, together with nothing but ambient lighting. To handle slightly different shading on different kinds of objects, four shader programs were created. One of these was a very simple shader, which only drew the objects at the correct position and applied a texture wihout any light shading. This one was used for the distant environment that was basically a large sphere, with a texture, surrounding the character. The other three shaders were all variations of the Phong shading model, where one shader could only draw monochrome objects with the correct light shading, one that did the same while also being able to apply textures to the objects, and one that applied both textures and reflections. See section 3.2 Reflections, for more information.

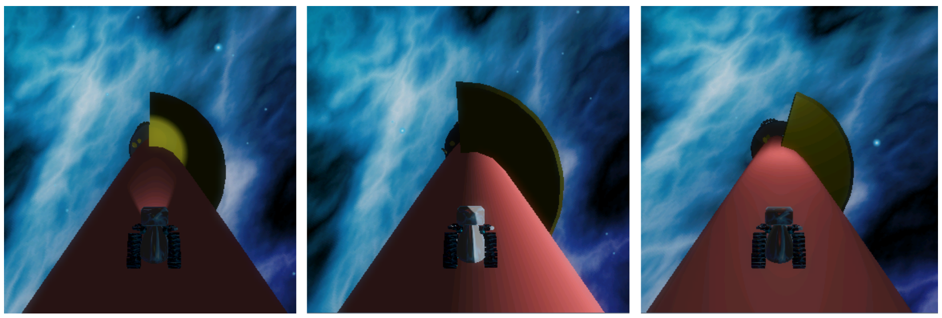


Figure 14: Spot light located at the front of the ship, casting light only within a cone shaped region. Directional light - here providing a homogeneous shading on the right side of the pipe. Point light located above the pipe, giving a smooth circular brightness on the upper part.

3.2 Reflections

The first part of this section focuses on general methods of simulating reflections in computer graphics. We will then describe how we created the appearence of reflective material for our player model.

A shiny surface, such as the hood of a car, looks unrealistic if it has no visible reflections of the environment. Rendering realistic reflections can be done in very sophisticated ways using ray tracing [15]. However, for real time applications, ray tracing is in most cases not a viable solution due to the complex calculations performed. To create the illusion of reflections, a technique called environment mapping can be used, with which a static representation of the environment is used to simulate reflections. There are several different implementations for such a technique, including spherical environment mapping, dual paraboloid mapping, and cube environment mapping.

3.2.1 Environment Mapping

To calculate reflections with environment mapping, a reflection ray is computed per fragment of the reflecting object. The environment texture(see Figure 15) is then used to, for each ray, fetch the color of the reflection. Spherical environment mapping is performed by representing the environment as a single texture with a spherical representation of the environment. The drawback of this method is that parts of the environment located near the edges of the sphere is represented with lower resolution. Dual paraboloid environment mapping is similar to sphere mapping but uses two textures representing the environment as paraboloids, one for the front and one for the back. Of the three methods, cube environment mapping represents the reflected scene with the smallest difference in quality for disparate/varying reflection directions. Six textures are used - one for each side of a cube. Due to the necessity of loading six textures each time a scene is rendered, this technique requires the most processing power [16].

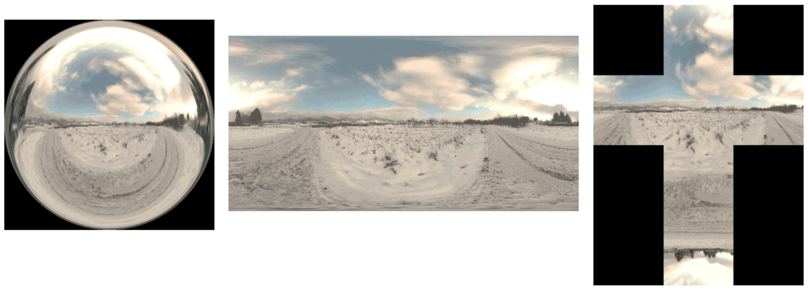


Figure 15: The environment representations for spherical-, dual paraboloid-, and cube -environment mapping. (images, curtesy of OakCorp, http://www.oakcorp.net/chaos/hdri.shtml)

3.2.2 Our reflection model

We chose to restrict usage of reflective material to only the player character model, mostly due to practical reasons, such as saving processing power. The reflections were implemented with the least demanding model, sphere mapping, since the limited screen size heavily reduces the visual gain of using a more complex model. The texture used as an environment map was simply a spherical representation of the texture used on the sky sphere. Although it might seem to be a simple solution, the visual difference between reflective and non-reflective material can clearly be seen in Figure 16.

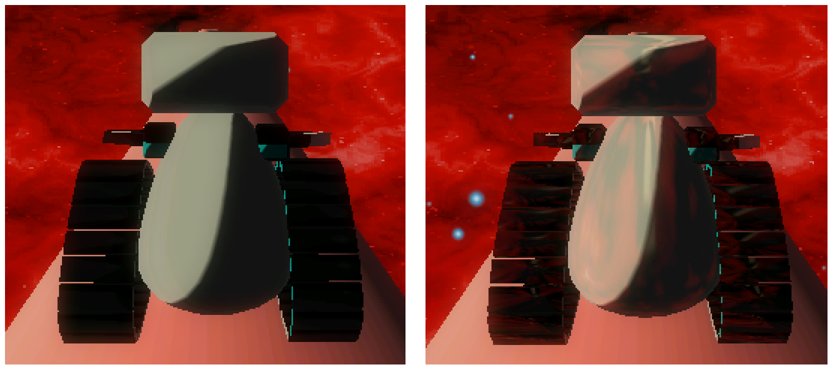


Figure 16: The visual difference between rendering without reflections (left), and with reflections (right).

3.3 Particle System

Particle effects are a way to represent effects that are hard to capture with other techniques. Effects such as fire, smoke, dust, fog and moving water are a few examples of what particle systems are designed to simulate. As the name suggests, the particle system imitates these phenomenons by creating a lot of small particles. These particles may be rendered as billboards, which is a textured quad that is always facing the camera. They may also be rendered as small points or pixels.

Every particle is given a set of properties. Lifetime, velocity, color and opacity are examples of properties a particle may have. In the simulation, the particles are spawned at a location, and moved at every frame depending on the velocity of the particle. When it reaches the end of its lifetime, it is removed or recreated as a new particle. By just using a small set of texture with varying size and opacity you can create complex and great looking effects.

3.3.1 Our Particle System

We use a particle system to create stars(see Figure 17) flying towards the camera. We chose to use billboards to do this. This choice was made because it is possible to understand and develop a billboard particle system in a short amount of time, and it suits the needs of our effect well.

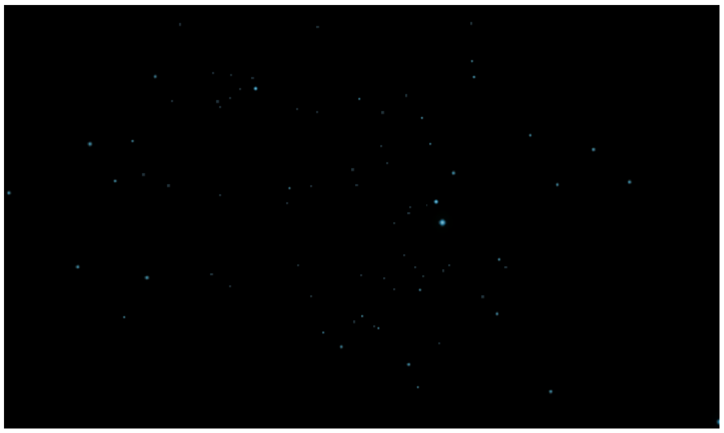


Figure 17: Particles used to create stars.

The particle system is built as one mesh, where each billboard is a quad, randomly placed in a predefined area. This mesh is rendered in view space and rotated with a model matrix to make sure that the stars does not rotate with the camera, but instead they remain at their position. The movement of each individual particle is handeled by the GPU with a formula that includes start position, velocity and time alive. Using the GPU for these calculations is faster than using the CPU and it allow us to compute the position of each individual star without having to change the mesh. The depth value of the particles will loop from zero to the value of the farplane, of the camera's view frustum, plus an offset to make sure that it does not feel like the same star pattern repeating itself. To further make sure that it does not feel like the star pattern is repetitive, we let each star have individual speed. The lifetime variable is updated via a uniform, and this value is updated with a formula depending on its previous value, the difference in time from previous update and current update, as well as the speed of the character.

3.4 Post-Processing

In this section we will discuss post-processing. Focus will be kept on the basics, and the bloom effect which is relevant to our project. Usually, when a game scene is rendered, all objects are drawn independent of each other. With the use of post processing, one can create effects that takes the whole scene into account. A variety of visual enhancements can be achieved through post-processing, such as motion blur, bloom, fog, and depth of field [17].

3.4.1 The process of post-processing

To be able to modify the scene before it is drawn to the screen, an image of the scene is rendered to an off-screen buffer, saving the relevant data, such as the color and depth of each fragment. The image is then used as a texture, applied to a full screen quad. The fragment shader will be run for each pixel, and since the scene is now treated as a texture, the neighboring pixels can be accessed. If we, as an example, wanted to create blur for our scene, we could now access nearby pixels and blend them together with the current pixel [18].

3.4.2 Bloom

Bloom is a post-processing effect that is used to reproduce how bright parts of an image sometimes seem to bleed into darker parts. Although this phenomenon is actually an artifact of real-world cameras, the effect can feel very natural and visually pleasing. Bloom can be achieved through the following process [19] .

- A high-pass filter is applied to a copy of the post-processing image.
- Blur the filtered image using Gaussian blur [20].
- Blend the filtered image and the original post-processing image together.

The result will look like the original image. However, the light parts now have blurry edges as though they were glowing (see Figure 18).

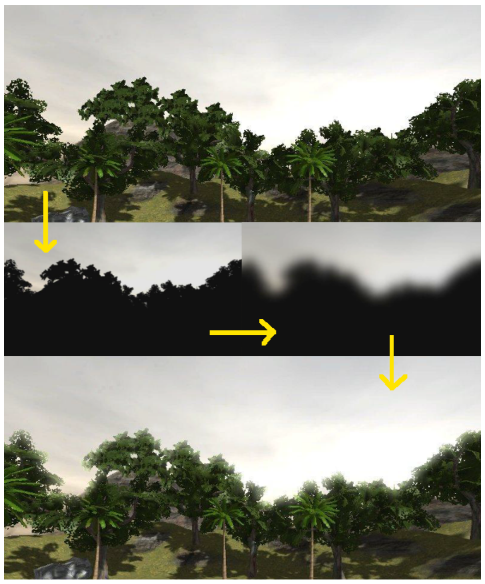


Figure 18: The figure shows the process of bloom, starting with the original image, followed by a copy of the image, passed through the high pass filter. The copy is blurred and blended with the original image for the final effect. (images, curtesy of Leadwerks Software, http://www.leadwerks.com/files/Tutorials/CPP/Post-Processing_Effects.pdf)

3.4.3 Our use of bloom

To implement bloom in our game, we chose to use a library created with the sole purpose of delivering visually appealing and computationally effective bloom [21]. It was constructed for use in collaboration with the game development framework libgdx, the framework used to implement our graphics. This library calculates bloom through the same process as described in section 3.4.2 Bloom. Our game took advantage of the bloom effect for several different purposes. Mainly, we used it the way it is traditionally used; to induce a stronger feeling of the bright parts of the screen actually being bright (see Figure 19).

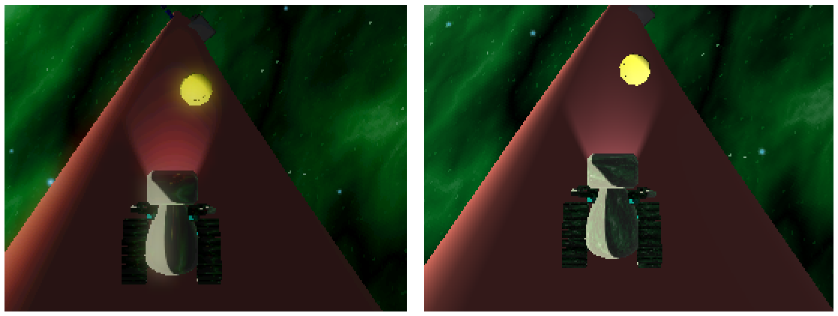


Figure 19: The bloom can clearly be seen on the left, as a soft glow around the edges of the coin, whereas the coin on the right has sharp and well defined edges.

We also found it interesting to increase the intensity, and lower the threshold for the high pass filter, of the bloom, when the player uses a boost power up. There is no logical reason as to why the bloom would intensify in accordance with the speed, though it turned out to provide a stronger sense of high velocity (see Figure 20).

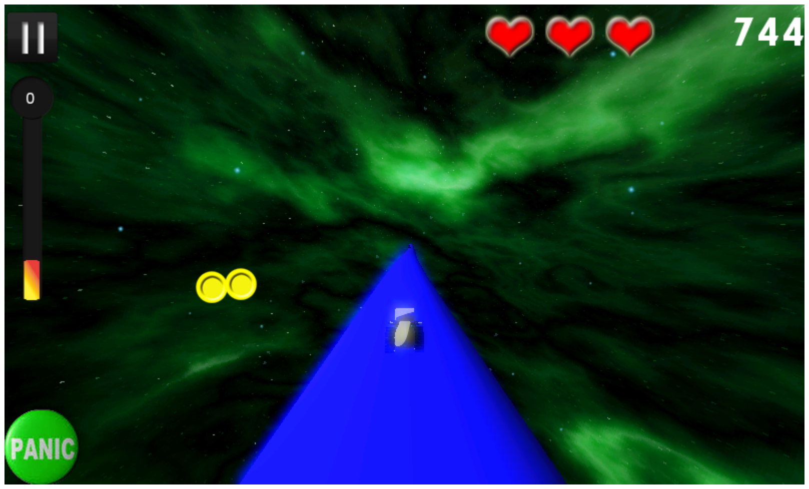


Figure 20: The intensity of the bloom is greatly increased while the player is traveling with boost speed.

In the game, as a new difficulty is reached, the environment texture changes to increase the sense of advancement. To hide the stale switch of textures, we also made us of the bloom. When a texture swap is about to occur, the bloom intensity starts to increase rapidly, while the high pass filter's threshold is simultaneously lowered, to the point where the screen is almost completely covered in bright blur. During this moment, when the game is barely visible due to the brightness, we change the environment texture, followed by a quick decline in the bloom intensity until it has reached its normal values. This process results in a short and bright flash on the screen, with a new environment after the end of the effect (see Figure 21).

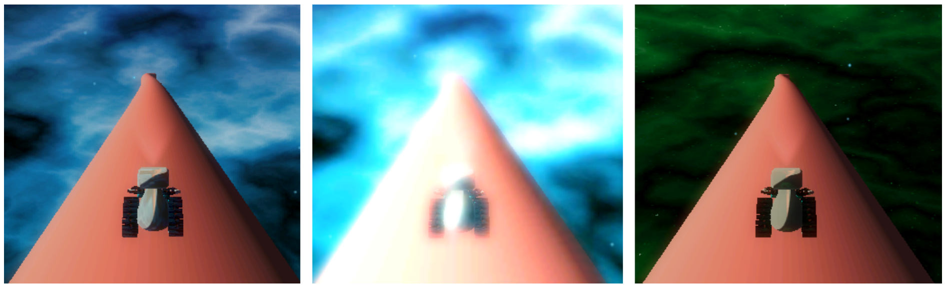


Figure 21: When a new level is reached, the screen goes from normal, to intensely bloomed, and then back to normal. After the effect, the environment texture has changed.

3.5 Modeling

The modeling process is an essential part when creating 3D games. In this section we will discuss the creation of models.

3.5.1 Modeling process

The three most common ways to build a model is with polygonal modeling, curve modeling, and digital sculpturing. Polygon modeling is, as the name suggests, a technique that uses polygons, also called meshes, to represent a model. Curve modeling uses mathematical curves, also called NURBS. Digital sculpting, which is the most recent method of these three, allows the modeler to modify the model as if it was made of real-life substance [22].

Models created for games are usually made with polygons, and not NURBS. This is because polygons are computationally less demanding. It is also important to use as few polygons as possible when creating models for games. This, due to that the 3D engines have limited processing power for rendering [23]. This is even more important when developing for smartphones. The two boxes in Figure 22 will be displayed the same for the viewer, but the one to the left will be less demanding to render than the one to the right since it consists of a lower amount of polygons.

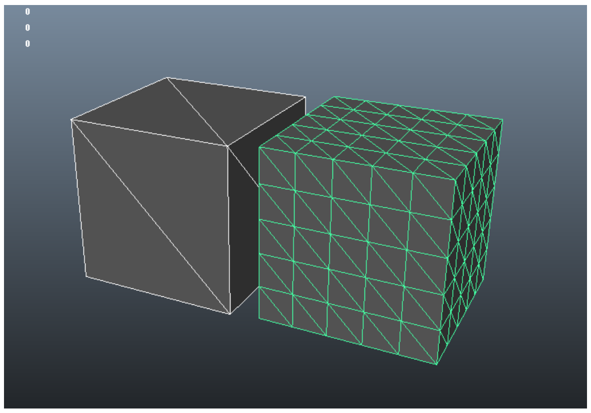
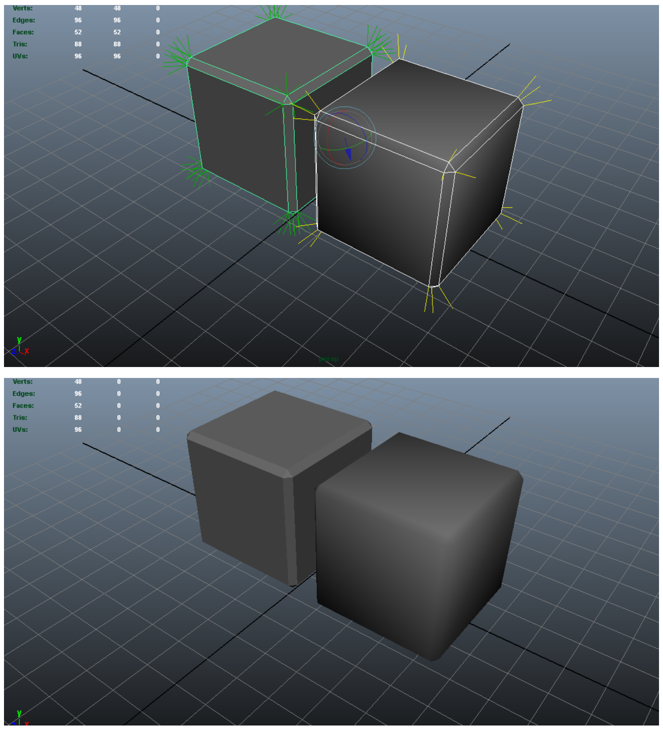


Figure 22: Two visually identical boxes with different amount of vertices.

3.5.2 Our modeling

Today, there are many tools for creating 3D models on the market. Three common tool are Maya, 3Ds max and Blender. We used the programs Maya and 3Ds Max when creating our models.

When creating models from scratch, we used primitives such as cubes, cylinders, and spheres. These primitvies were attached to each other, to create the various objects that were needed. The difficulties when modeling were how to create good looking models with as few vertices as possible. We could not create models with the same amount of vertices as the models used for desktop computer games, due to that the capacity of a smartphone is lower. With this in mind, we chose to create the obstacles by using simple cubes and cylinders. To create smooth edges, we made them beveled. This increased the amount of triangles, but our opinion was that it added a better look to the obstacles and that it was worth it. To increase the smoothness of the edges, the normals were modified. Instead of three normals per vertex, we used one averaged normal in each vertex (see Figure 23). This created a much better look to the game.



```
Verts:   48   48   0
Edges:   96   96   0
Faces:   52   52   0
Tris:    88   88   0
UVs:     96   96   0
```

```
Verts:   48   0   0
Edges:   96   0   0
Faces:   52   0   0
Tris:    88   0   0
UVs:     96   0   0
```

Figure 23: Two boxes with three normals per vertex (left) compared to one normal per vertex (right).

The level of detail was another part that had to be evaluated. The first version of the coin model consisted of around 230 vertices, which is a huge amount compared to our obstacle models. The second model that was made, the one that is now used in the game, consist of 90 vertices (see Figure 24). In the game, the difference between these two models is almost impossible to notice, since everything is moving so fast. By doing this change, we saved some of the smartphone's processing power.

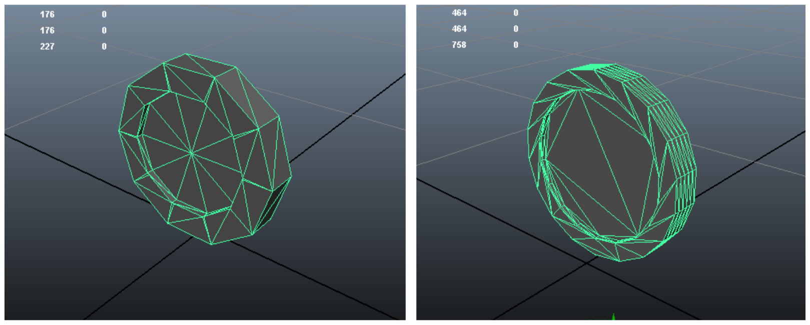


Figure 24: First (right, 230 vertices) and second coin model(left, 90 vertices) used in Tisado.

3.6 Animations

This section describes animations. First, we will start with a section on the background of animations. Finally, we will cover our own animations.

3.6.1 Background

The earliest research in computer graphic and animations can be tracked back to 1963, when Ivan Sutherland invented the Sketchpad, which was the first computer program that could interactively create line drawings on a computer screen [24]. But it was not until in the early 1970s that computer animations started to spread over the world . This was due to that the University of Utah had received funding from the government. As a result of this, they made some groundbreaking animations between 1972 and 1974. Ed Catmull made an animated hand and face, Barry Wesser created a walking and talking human, and Fred Parke created a talking face. Today, this might not look that much but at that time, this was ground breaking [25].

3.6.2 Our Animations

In our game, we have created animations for the player character. The character is in the shape of a robot, with tank treads instead of legs. These tank treads are animated to visualize movement. The animations were done using a script in 3Ds max, that made the treads move forward along a path (see Figure 25).

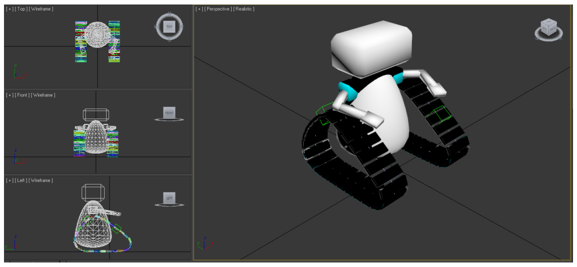


Figure 25: The player model of Tisado, displayed in 3Ds Max.

The animation was exported into a format called MD2. This format supports animated 3D models, and was originally used by Id Software's Quake II engine. Libgdx supports four different formats for animated models; these are MD2, MD5, G3D and G3DT. It is not possible to export from 3Ds Max to G3D and G3DT. The choice of MD2 over MD5 was made because we were told, by Stefan Bachmann from The Grey Studios, that MD2 is faster on smartphones. The MD2 file format was created in 1997 [26] and, since it is an old file format, there is no built in exporter for MD2 in 3Ds max. Instead, a script had to be used that could export to MD2 from 3ds Max.

4 Game Logic

The logic used in Tisado describes a simulation of a robot traveling on a pipe. This includes simulating gravity from the centre of the pipe. It also includes controlling the character by jumping and turning, checking collisions and the generation of an infinite random world. This chapter will explain how these features were implemented.

4.1 Gravity

Gravity is a complicated feature to add to a game, and it becomes even more complicated in a 3D world with solid objects. To avoid the problems that comes with general gravity implementation, we chose to solve this in a simpler way.

The character's position and movement is calculated as if it was moving on a line rather than a pipe. This is done by using an algorithm that takes the rotation and position of the character, the radius of the pipe and the direction of the line and calculate the position of the character on the surface of the pipe. The algorithm tracks the point at the top of a theoretical circle with the given radius, perpendicular to the z-axis. This point is rotated along the circle according to the rotation value, and finally the entire circle is rotated to face the direction given (see Appendix B.1).

To simulate a jump, an offset is added to the radius used for calculating the position in the gravity algorithm. This offset is calculated as a physics simulation of a jump made in 2D, where only base gravity and initial up force is needed to compute the current height and return value.

4.2 Movement

Most modern smartphones are equipped with an accelerometer that measures the acceleration exerted by the gravity of our planet [27]. There are three axes that measures the orientation: x, y, and z (see Figure 26).

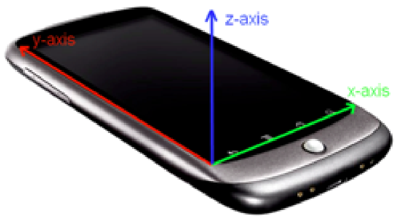


Figure 26: The accelerometer axis on an Andoid phone. The z-axis points out of the phone (picture taken from Zechner M. (2011) Beginning Android Games)

The gravity of our earth is 9.82 m/s^2 [28]. Therefore, if one axis of the phone is pointing down, it will result in the maximum value of 9.82, and in the opposite way will result in -9.82. The character is controlled by tilting the phone to the left or right, and since we have limited our game to be displayed in landscape mode (see Figure 27), the only concern is the value from the y-axis when calculating the rotation speed of our character.

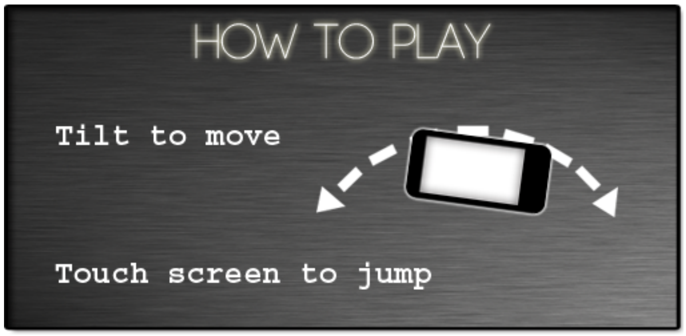


Figure 27: The player character is controlled by tilting left and right.

To be able to turn quickly to avoid obstacles in high speed, as well as being able to turn quite slow in lower speed, there were some need for a combination of algorithms. We also needed to limit our angle of the y-axis, when the maximum rotation speed was reached. We used the algorithm, $v = ka^2$ where v is the speed of the characters rotation around the pipe, a is the acceleration of the y-axis, and k is a constant amplifier. After that, the algorithm transform to a more moderate linear algorithm, until maximum angle of the y-axis has been reached, see Figure 28 and code implementation in (see Appendix B.2).

$$\begin{cases} v = ka^2 \; ; \; a < accelerationStopper2 \\ v = la \; ; \; a < accelerattionStopper1 \\ v = m \; ; \; a \geq accelerationStopper1 \end{cases}$$

Figure 28: Simplified algorithm for the acceleration around the pipe.

4.3 World Generation

In order to meet the concept of Tisado, we needed to develop a system to generate a random map. The user is allowed to either set a specific seed to be used, or to let the application randomly generate the seed. With this seed, we generate five segments of pipe which are put together into a long pipe. Each of these segments are generated separately, and filled with obstacles and coins depending on current difficulty of the game.

Each segment is generated as nodes, following a spline with start points placed to match the end points of the previous segment. Each of these nodes are expanded to multiple points forming a circle. These points are rotated so that the circle faces the next point of the segment. The points that are forming the circle are used to render the pipe. Each segment contains 784 vertices that are connected by 1632 indices (see Figure 29).

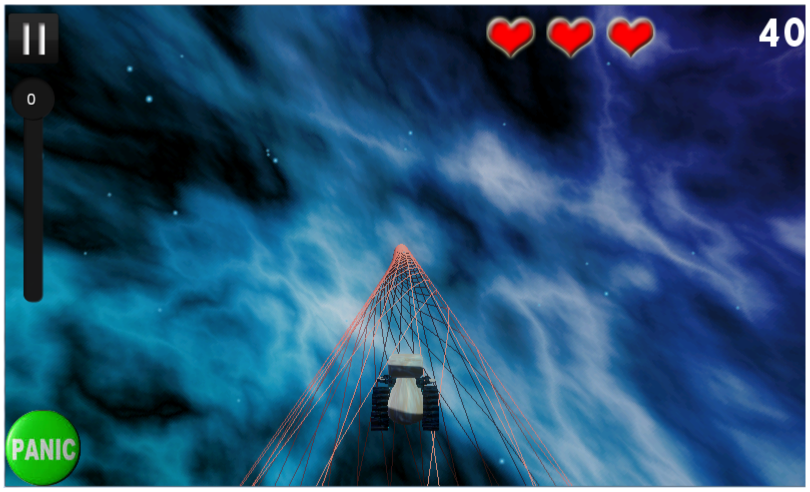


Figure 29: The vertices of the pipe, made visible by rendering with lines rather than triangles.

The character is always on one of the segments, and when the character passes on to the next segment, the previous segment is regenerated and moved to the end of the last segment. This approach makes it easy for us to make the game infinitely long. It also comes with many other advantages, such as allowing us to use frustum culling (see 5.6.3 Frustum Culling) to avoid sending the entire pipe to the GPU each frame.

The level of difficulty in the game increases depending on how many segments the user has passed. Because the pipe is generated in five segments, and each segment is regenerated with the current difficulty when the character has passed it, this means that there will be a delay of four segments before the user notices the increase in difficulty. To avoid this, we made sure that the application is aware of how many segments are left before the user will be notified of the increased difficulty, and that the actual difficulty increase happens four segments before the user is notified.

4.4 Collision Detection

Key features in Tisado are to be able to dodge objects, collect coins and power-ups. What we needed was a way to detect collisions between these objects and the player. This is done with the help of something called collision detection. In this chapter we will give a brief introduction to what collision detection is, and explain some commonly used techniques. We will finish by describing how we solved this task. What we looked for was a collision system that was fast enough to run on smartphones, while still being accurate enough to detect collisions between the player and the obstacles.

The process of collision detection is to find out when two object intersect. There are several ways to achieve this. The problem can be divided into two groups - collision detection and collision handling. Collision detection classically seeks a binary answer: intersection or no intersection. Collision handling is the response to the collision.

Collision detection is often divided into two parts, a broad phase and a narrow phase [29].

4.4.1 The broad phase

The purpose of the broad phase, is to reduce the number of potential collision checks needed by checking whether or not two object are in the vicinity of each other. A common practice is to use some sort of bounding volume that covers the object. This bounding volume can be any simple geometric shape, such as a box, a cone or a sphere. Since they are of simple geometric shapes, they take a short time to calculate and give a good indicator if the objects are in the vicinity of each other. By having this easy to calculate bounding volume, the overhead is reduced by simply checking whether or not the two objects are close to each other. When they are close to each other, more detailed checks can be made to see if they actually collide. These detailed checks are also known as the narrow phase.

Another method to reduce the number of collisions to be checked is known as Sweep 'n Prune, also known as Sort and Sweep. This algorithm is a broad phase algorithm, which sorts all the objects that may collide in a list; often a linked list. This list is sorted in such a way that objects close to each other in the game world are close to each other in the list. Since objects are sorted in such a list, the narrow phase algorithm needs only to check neighboring objects in the list for a collision, thus reducing the number of total checks needed. The Sweep 'n Prune algorithm exploits the fact that objects do not move very far between frames, therefore the list will remain almost sorted in the next frame. Sorting an almost sorted list takes linear time with a simple insertion sort and therefore the list is easily maintained [30].

4.4.2 The narrow phase

The narrow phase is to check if two objects, that passed the broad phase, have actually collided. Since this check is more precise than the broad phase, it takes a longer time to calculate. This test is often done to collect information about the collision so that it may be handed over to the collision handler. An example of additional information could be the angle of impact, the speed of the colliding objects and the mass of the two objects. Since the broad phase helps to reduce the number of narrow phase checks done, the narrow phase checks may be more precise.

One way to check if two objects collide is to check every triangle of the first object against all the triangles of the other object. This has a very poor performance and is rarely used. Another way to do this is to represent the object with one or more bounding volumes. Since most objects in games are meant to represent real world objects, they may be hard to represent with a single bounding volume. Therefore, objects are often represented with numerous bounding volumes to better emulate their shape.

4.4.3 Continuous vs discrete time

There are two ways to implement a collision detection system; with discrete time or continuous time. When working with discrete time, the collision detection algorithm is called at regular time intervals, and collisions may only be detected at these time intervals. The continuous time, however, operates by predicting when and where a collision will happen. The benefits of a discrete time system is that you only have to feed the algorithm with a list of bodies that can collide, while a continuous time system needs to know about all the bodies at all times to be able to predict when a collision will occur.

A problem with discrete systems is that they may miss collisions if the bodies pass through each other before a call to the algorithm can detect the collision [30].

4.4.4 Our Implementation of Collision Detection

In our game, we chose to implement a simple but effective collision detection. The playerd model can collide with obstacles and pick ups. Since the obstacles and pick ups are of geometric shapes, we chose to use axis-aligned bounding boxes for all the obstacles (see Figure 30). To represent the player model, we used a list of six vectors that covered key points of the model (see Figure 31).

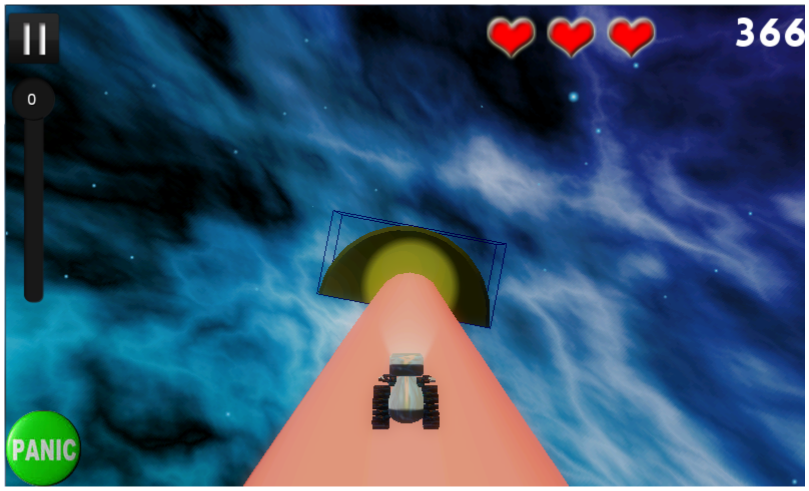


Figure 30: The bounding box of a large half circle is rendered.

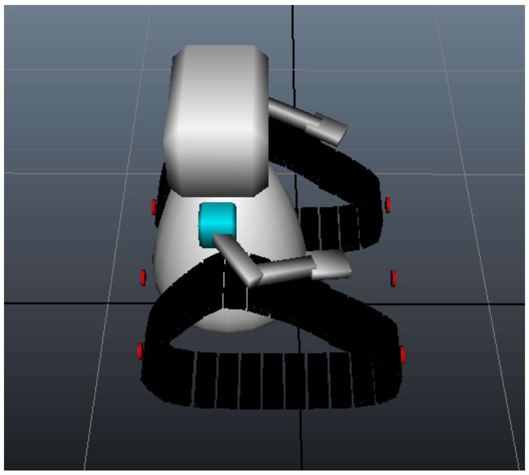


Figure 31: Showing the collision points of the robot.

As mentioned in 4.3 World Generation, the pipe and the obstacles are divided into segments. These segments contain about 10-15 obstacles each, and in order to reduce the number of collision checks needed, we only check the segment the player model is currently on. Since our narrow phase checks are made up of axis-aligned bounding boxes, we can afford to make the narrow phase check 10-15 times every frame without risking to impact the performance of the game noticeably. Since axis-aligned bounding boxes, as the name suggests, can only be axis-aligned, we had to compensate. Every obstacle in the game has a model matrix that represents where in the game world the obstacles are positioned. When an obstacle is loaded into the game, it is positioned at origin. At this point the obstacle is axis-aligned, and we may represent it as an axis-aligned bounding box. However, when we multiply it with its model matrix, it may be rotated. Since an axis-aligned bounding box cannot be rotated, we had to multiply the inverse of the model matrix for the obstacle we were about to check, with the points of the player model. This way, we move the points of the player model to origin, and we can represent every obstacle, even if it is not axis-aligned, with an axis-aligned bounding box. The advantage with an axis-aligned bounding box is that the computations needed to check whether or not a point is contained within the box are few. The axis-aligned bounding box is represented by two points - max and min. The max point represents the top-right furthest away corner. The min represents the closest bottom-left corner. The process of checking whether or not a point is contained within a axis-aligned bounding box is simple. A point is contained within the box if it is within the constraints of min and max.

5 Optimization

Even though mobile devices have been developed greatly during the last few years, performance is still an issue. Optimizing the code is often a time consuming part, but it is important to certify that the code is as close to optimal as possible. This often means that it is necessary to go against and break basic programming rules. For example, using public methods for accessing and setting private variables is seven times slower than directly referencinging them as public variables [31]. This chapter will explain some of the most common performace problems and how we solved them.

5.1 Vertical sync

Vsync, or vertical sync, is used for limiting the frame rate of an application to the refresh rate of the screen, and thus saving power by not calculating frames that would not be displayed anyway. Its primary use, however, is not to save power but to prevent screen tearing. Screen tearing is when two or more parts of different frames are shown in the same screen draw (see Figure 32). As good as vsync might sound, it comes with some problems. These problems involve the possibility of the application skipping entire monitor refreshes and thus reducing the visible frame rate by, at minimum, fifty percent.

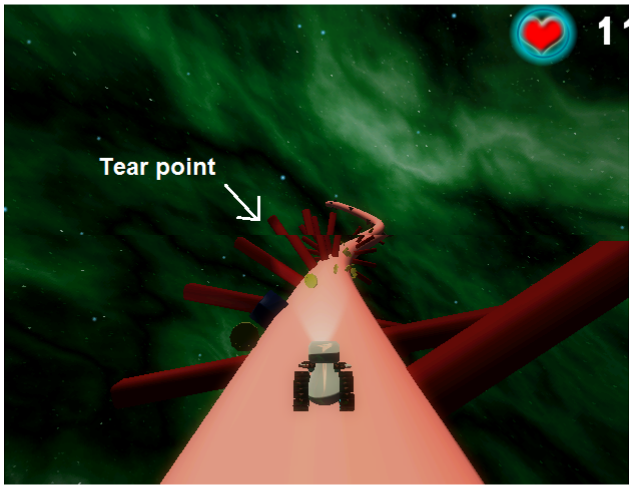


Figure 32: Screen tearing with one tear point.

Screen tearing happens when the actual frame rate is higher than the refresh rate of the device. If the actual frame rate is 90 and the refresh rate is 60Hz, the following will happen:

- In the first 10.67ms, the frame buffer will be filled.
- In the next 5.33ms, the application writes a new version of the first third of the screen to the frame buffer.
- After 16ms, the device takes the frame buffer and renders it to the screen.

This means that the first third of the screen will be from frame number two, while the two other thirds will be from frame number one. Thus, frame tearing appears.

There is a technique called double buffering which tries to mitigate the tearing problem. It does this by writing new screens to the back buffer instead of the frame buffer. When the back buffer is full, the buffers are swapped. Thus it will always pull completed screens. Or at least this would have been the case, if the swapping could never happen during rendering; but it can. Thus, it is possible that the monitor pulls part of the data from the wrong buffer, and again screen tearing will happen.

Vsync solves this problem by only swapping the buffers right after a monitor refresh has happened. With a frame rate higher than the monitor refresh rate, this is all fine. But if the frame rate goes below the actual frame rate, the following will happen:

- In the first 16 ms, ninety percent of the screen is rendered to the back buffer.
- Monitor refreshes itself after 16ms. The back buffer is not ready. Thus no new frame data is pulled.
- After another 1.6ms, the remaining ten percent of the screen is rendered to the back buffer. The back buffer is now full, and nothing can be written to it before the buffers have been swapped. Thus, it has to wait for the next monitor refresh, leaving 14.4ms of wasted time. After those 14.4ms, the monitor refreshes itself and the backbuffer is now empty and ready to be written to.

In two screen updates, only one new frame has been written to the screen. This means that the only possible frame rate is monitor refresh rate divided by N, where N is a positive integer. The gap between N = 1 and N = 2 is big compared to the other gaps. If the monitor has a refresh rate of 60Hz and the time to update and render a frame is between 1ms and 16ms, the frame rate will be 60 frames per second. But if the time to update and render is 17-32ms, the frame rate will be cut by half. Thus, one millisecond extra can cause the frame rate to drop from 60 to 30 frames per second [32].

5.2 Hardware

Android.com lists over 220 different devices using the Android operating system [33]. It is likely that not one of these devices has the exact same hardware specifications as any of the other devices. Because some devices are much stronger than others, it is hard to make a game which works good on all devices and still uses the full potential of each device.

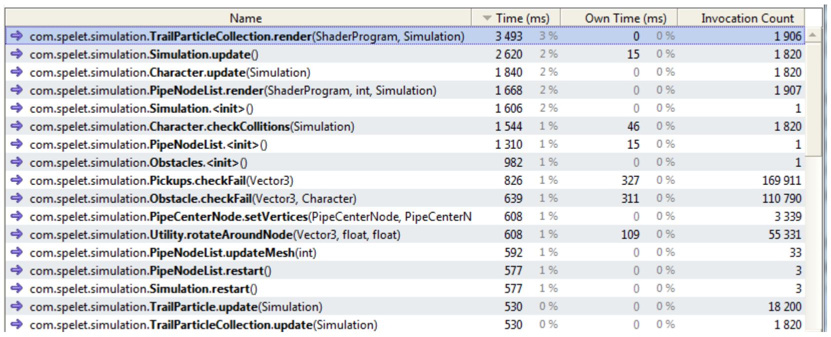
Because the hardware specifications are different between the devices, it is easy to understand that the problem with vsync, as described in 5.1 Vertical Sync, is likely to occur on at least a few out of the many devices. Thus, it is often a good idea for developers to set a minimum target device. Minimum target device means that the application is designed to work without stuttering or other problems on devices with hardware equal to, or stronger than, the minimum target. This, however, does not necessarily mean that the application will not work on devices with weaker hardware.

In our project, we consider the application to be working if the frame rate does not drop below 30 frames per second at any point. On the Samsung Galaxy S2, our minimum target device, we run the application with 40 to 60 frames per second. Each lap in our game loop takes between 12ms and 20ms, depending on the amount of objects being rendered. This means that it is the vsync problem that is reducing our frame rate. It also means that a device that renders fifty percent slower would still be getting 30 frames per second. 20ms*1.5 is still less than 32ms.

5.3 Bottlenecks

Locating the bottleneck of an application can be done in multiple ways. One of the simplests ways to locate them, is to reduce the workload on particular parts of the application. If the frame rate increases because of the reduced workload, the bottleneck has been found [34].

Profilers are great tools for finding bottlenecks in applications. We used a profiler called YourKit, which is free and fully functional, during an evaluation period and after that paid [35] . This profiler allowed us to profile the CPU, and check which methods required the most work from the CPU. A CPU profiler is best used to track game logic methods, and because all of our game logic is run from the class simulation, we can track methods run by this class and view the results (see Figure 33).



Name	Time (ms)	Own Time (ms)	Invocation Count
com.spelet.simulation.**TrailParticleCollection.render**(ShaderProgram, Simulation)	3 493 3 %	0 0 %	1 906
com.spelet.simulation.**Simulation.update**()	2 620 2 %	15 0 %	1 820
com.spelet.simulation.**Character.update**(Simulation)	1 840 2 %	0 0 %	1 820
com.spelet.simulation.**PipeNodeList.render**(ShaderProgram, int, Simulation)	1 668 2 %	0 0 %	1 907
com.spelet.simulation.**Simulation.<init>**()	1 606 2 %	0 0 %	1
com.spelet.simulation.**Character.checkCollitions**(Simulation)	1 544 1 %	46 0 %	1 820
com.spelet.simulation.**PipeNodeList.<init>**()	1 310 1 %	15 0 %	1
com.spelet.simulation.**Obstacles.<init>**()	982 1 %	0 0 %	1
com.spelet.simulation.**Pickups.checkFail**(Vector3)	826 1 %	327 0 %	169 911
com.spelet.simulation.**Obstacle.checkFail**(Vector3, Character)	639 1 %	311 0 %	110 790
com.spelet.simulation.**PipeCenterNode.setVertices**(PipeCenterNode, PipeCenterN	608 1 %	0 0 %	3 339
com.spelet.simulation.**Utility.rotateAroundNode**(Vector3, float, float)	608 1 %	109 0 %	55 331
com.spelet.simulation.**PipeNodeList.updateMesh**(int)	592 1 %	0 0 %	33
com.spelet.simulation.**PipeNodeList.restart**()	577 1 %	0 0 %	3
com.spelet.simulation.**Simulation.restart**()	577 1 %	0 0 %	3
com.spelet.simulation.**TrailParticle.update**(Simulation)	530 0 %	0 0 %	18 200
com.spelet.simulation.**TrailParticleCollection.update**(Simulation)	530 0 %	0 0 %	1 820

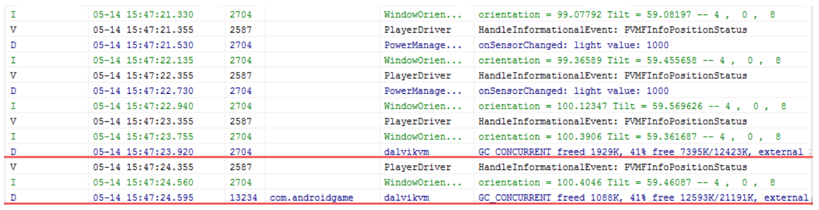
Figure 33: The output of the profiler YourKit, searched for simulation.

Because the profiler slows the system down, the times are not what they would normally be. This, however, does not mean that the information is useless. The methods that are only invoked once are run in the constructor and are therefore not of high interest for optimization. In this profiler data there is nothing that really stands out, because we have already spent many hours optimizing these methods to make them as fast as possible. For example, in the past we used collision detection for all obstacles and power ups, not just a smaller list with the nearby ones, this meant that the collision check took up to five times longer than the current one. This was easy to overlook when writing the code, but it was also easy to find that the collision detection was taking too long when a profiler was used.

There is another type of profiler, for the GPU, that could be used to profile OpenGL calls. These profilers are, however, considerably harder to set up than a CPU profiler is. We felt that the time spent on setting one up was not likely to make up for the gains. Therefore, we chose to optimize the rendering of objects manually, by looking through the shaders, the models, and the amount of objects.

5.4 Garbage Collecting

The CPUs of smartphones are not as strong as those used in modern desktop computers. Thus, so is the garbage collecting. Slow garbage collecting is one of the largest problems that the project came across during the development (see Figure 34).



```
I  05-14 15:47:21.330  2704                    WindowOrien...  orientation = 99.07792 Tilt = 59.08197 -- 4 ,  0 ,  8
V  05-14 15:47:21.355  2587                    PlayerDriver    HandleInformationalEvent: PVMFInfoPositionStatus
D  05-14 15:47:21.530  2704                    PowerManage...  onSensorChanged: light value: 1000
I  05-14 15:47:22.135  2704                    WindowOrien...  orientation = 99.36589 Tilt = 59.455658 -- 4 ,  0 ,  8
V  05-14 15:47:22.355  2587                    PlayerDriver    HandleInformationalEvent: PVMFInfoPositionStatus
D  05-14 15:47:22.730  2704                    PowerManage...  onSensorChanged: light value: 1000
I  05-14 15:47:22.940  2704                    WindowOrien...  orientation = 100.12347 Tilt = 59.569626 -- 4 ,  0 ,  8
V  05-14 15:47:23.355  2587                    PlayerDriver    HandleInformationalEvent: PVMFInfoPositionStatus
I  05-14 15:47:23.755  2704                    WindowOrien...  orientation = 100.3906 Tilt = 59.361687 -- 4 ,  0 ,  8
D  05-14 15:47:23.920  2704                    dalvikvm        GC_CONCURRENT freed 1929K, 41% free 7395K/12423K, external
V  05-14 15:47:24.355  2587                    PlayerDriver    HandleInformationalEvent: PVMFInfoPositionStatus
I  05-14 15:47:24.560  2704                    WindowOrien...  orientation = 100.4046 Tilt = 59.46087 -- 4 ,  0 ,  8
D  05-14 15:47:24.595  13234  com.androidgame  dalvikvm        GC_CONCURRENT freed 1088K, 41% free 12593K/21191K, external
```

Figure 34: Logcat displaying the pause of an application, due to garbage collecting.

Whenever an object is eligible for garbage collecting, work needs to be done by the garbage collector. An object becomes eligible if it is not reachable from any live threads or any static references. Cyclic dependencies are not counted as references, so if object A has a reference to object B and object B has a reference to object A and they have no other references, then both objects will be eligible for garbage collecting [36].

This means that it is important not to create unnecessary objects, and to reuse those already created. While old Android devices have to pause every time the garbage collector needs to work, new Android devices support concurrent garbage collecting. A concurrent garbage collector can collect garbage concurrently with the running applications, but needs to pause them to remove the collected garbage [37].

Many of the calculations for the game logic are made using vectors and matrices, and because there might be hundreds of the same calculations being run after each other every frame, it is possible to set these calculation vectors and matrices as static. For example, this means that even if there are multiple obstacles to check collision detection with, only one vector will be used, thus saving a significant amount of time. Pooling is another way to avoid trouble with garbage collecting. Pooling means that instead of removing the references to objects when they are no longer needed, they will be placed in a list of unused objects. These objects will then be used next time an object of the same type needs to be created. An example where pooling is used in the project is for the pipe. The pipe is generated in five segments, and each of these segments are in turn a list of 100 nodes. When the ship has passed one of these segments, it will have its nodes moved to become the new last segment of the pipe. This is done without creating any new objects.

5.5 Loops

When optimizing loops that iterate through collections, it is important to know that the best way to do this is different depending on the type of the objects in the collection, and the type of the collection. When the collection is a plain array of any type, enhanced for loops are to be preferred, but they are only slightly better than hand written counted loops. This can be compared to when the collection is an ArrayList or similar. Using an enhanced for loop in this case will create an iterator, and thus making the loop up to three times slower than using a hand written counted loop [31]. It is also important to remember that when iterating through an ArrayList, or similar collections, with an enhanced for loop, a new iterator will be generated each time to loop is initiated and discarded by the end of the loop. This iterator will need to be taken care of by the garbage collector, which in turn will cause the problems described in 5.4 Garbage Collecting. We tested this ourselves on a Samsung Galaxy S2 by writing an application that iterates an ArrayList with an enhanced for loop 10 000 times every frame. This caused garbage collecting to be called frequently; at least once for each time the code was run. We then rewrote the application so that it iterated the objects in the ArrayList using a normal for loop and, with this approach, garbage collecting was never called. Thus it is easy to see why and how high performance penelty enhanced for loops can be compared to normal loops.

Case 1, using enhanced for loop	Case2, using handwritten counted loop

ArrayList<Cat> cats = new ArrayList<Cat>(); for (int i = 0; i < 10; i++)
cats.add(new Cat());
applicationLoop:
for (int i = 0; i < 10000;i++) {
for (Cat c : cats) {
c.doStuff();
}
}
	ArrayList<Cat> cats = new ArrayList<Cat>();
	for (int i = 0; i < 10; i++)
	cats.add(new Cat());
	applicationLoop:
	for (int i = 0; i < 10000;i++) {
	for (int j = 0; j < cats.size();j++) {
	cats.get(j).doStuff();
	}
	}

5.6 Shaders

When writing shaders, it is important to remember that the fragment shader will be run for each pixel of each triangle, while the vertex shader will be run once per vertex. See 3.1.3 Shader Programs, for more information. The objects in Tisado vary in vertices between 24, for the most simple obstacles, and 90, for coins. In total, roughly 15000 vertices are used to render each frame. This can be compared to a full screen object in resolution 800x480 (Samsung Galaxy S2 resolution)[38]. Such an object contains 384000 pixels, and it is likely that many more will be sent through the fragment shader. This means that it is important to have as little code and maths as possible in the fragment shader and calculate as much as possible in the vertex shader. It is also important to remember that the GPU is faster with matrix and vector calculations than the CPU, and therefore as much maths as possible should be run in the vertex shader rather than calculated by the CPU.

5.6.1 Fillrate

Fillrate is a big limitation in the GPU of mobile devices. Fillrate is a word for how many pixels the GPU can write to the frame buffer each second [39]. In order to best avoid problems with fillrate, it is important to send as few pixels as possible through the fragment shader. This means that avoiding to render objects or pixels, that will not be shown in the final scene anyway, is of high priority. Avoiding sending objects to the GPU can be done using different culling techniques.

5.6.2 Depth Test

Depth testing is used for checking whether the pixels calculated by the fragment shader should be rendered or not. This is done by checking whether there is anything blocking the view from the camera to the object. If the depth test is disabled, all objects will be rendered on top of the previously rendered objects (see Figure 35).

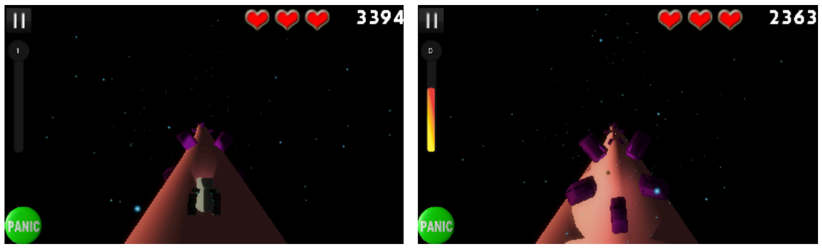


Figure 35: Visual difference of rendering with depth test (left) and without depth test (right). Sky sphere has been removed to visualize the impact of depth testing.

Many modern GPUs support a concept called early depth testing. This means that the depth testing will be run before the fragment shader is executed. Thus, it is possible to save performance [40].

A simple way of utilizing early depth testing, is to render objects in an order based on their distance to the camera. In Tisado we first render the player model, because the camera will always follow the player model and thus, it will always be visible. Second, we render the pipe because it is a large object and it takes up the most amount of screen space, apart from the sky sphere. We then go on with minor objects such as obstacles, pick ups and environment objects.

The sky sphere is rendered last because it will always be further away than the other objects.

5.6.3 Frustum Culling

Frustum culling is a useful tool used to avoid sending unnecessary data to the GPU. The idea is to check whether an object is within the camera's view. The camera's field of vision may look in various different ways, but generally, and in Tisado, it is a frustum of a rectangular pyramid (see Figure 36). For this reason, the cameras field of vision is often called the camera's view frustum. It is important to set a relatively big frustum for the camera to make sure that the object's near the far plane are small enough to not look like they are jumping from not visible at all to full size when they pass the far plane. The idea is to have the far plane just behind where the horizon would be. However, setting a too big frustum will result in a lot of objects, that are very small due to the distance, being sent to the GPU and rendered even though it is not necessary.

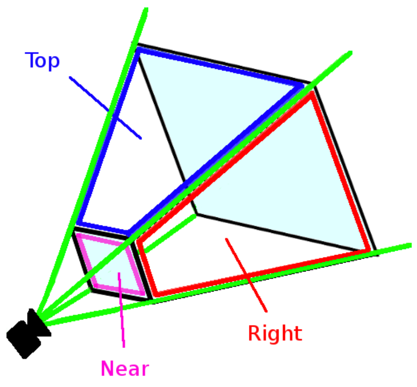


Figure 36: The frustum of a general camera.

If the object is not in the camera's view frustum, the object does not need to be sent to the GPU. Thus, saving time. Because the idea of this check is to save time, it is important that the actual check is very quick. To achieve a check as fast as possible, it is often a good idea to create a sphere or box around each object and check whether or not this geometrical form is inside the frustum. A check like this is less precise, but much faster than a vertex-perfect check would be. It is also possible to just check whether the centre of the object is inside the frustum. However, this can result in the object jumping into the screen. Imagine a cube twice the size of the frustum slowly moving towards the camera. If only the centre of this object is checked for being inside the frustum, the entire frustum will be filled with the cube when this finally happens [41].

In Tisado, however, we use the centre check approach even though it has some downsides. We do this because we need to optimize everything as much as possible to make sure the game runs as good as possible on all devices, and because all objects are relatively small compared to the frustum and the jumping will not be visible more than it would with normal frustum culling.

5.6.4 Back-Face Culling

Back face Culling is, unlike the Frustum Culling, made by the GPU. The idea here is to only fill the triangles that are facing towards the camera. Thereby, only half as many faces must be rendered. For example if you render a cube without any back face culling enabled, the inside of the cube will also be filled. This is generally not necessary in games. Thus, enabling back face culling is standard and should be used. Back face culling is responsible for the effect often seen in games in which, if the camera is inside a mesh, rather than seeing the insides of the mesh, the camera is able to look out of the mesh through what would be seen as a wall from the outside.

In order to check if the triangles are facing the camera or not, OpenGL checks whether the vertices of the triangles are appearing in an order that is clockwise or counter clockwise. One would think that it would be possible for the GPU to just check the normals of the vertices. This is not ideal, because the programmer might choose to set the normals in any direction, to create interesting lighting effects. Thus, if the normals were used, the faces would not always be rendered [42].

5.6.5 Occlusion Culling

Occlusion culling is the process of determining whether an object is hidden by another object from a certain viewpoint, generally from the viewpoint of the camera. It works much like depth testing, with the exception that Occlusion Culling is run before the objects are sent to the GPU. There are various different ways to do occlusion culling, but these ways are generally very expensive and are therefore not suited for a mobile game like ours. There are, however, special cases where occlusion culling can be very cheap and easy to do. For example, if the pipe in Tisado were always moving straight forward it would be possible to check the rotation of each object and compare that with the rotation of the camera. With coins, which are very small, it would be possible to check if the object is within cameras rotation plus 120 and camera's rotation minus 120 to determine whether it is visible or not. The number 120 is just an approximation, and this value would differ depending on the size of the objects. Minecraft is another example of where occlusion culling could easily be implemented and used with profit. If a block has all 26 neighbours, it does not need to be rendered [43].

6 Music and Sound

The resource demands of the music and soundeffects, in a smartphone game, must be kept as low as possible, because we had to consider the limitated storage and memory availible in the devices [44]. There are several different music formats to choose from, and it must be something that is suitable for Android, and more specifically libgdx, both in filesize and sound quality [45]. First, we will explain the availible implementation methods and music formats. After that, we are comparing them and showing our result.

6.1 Implementation

We were developing this game on an Android mobile phone with the help of a third party game library, Libgdx. To be able to make the best choice of the music formats within our limitations, we had to take into account that there were some different audio implementationts that was not covered by Libgdx. In the methods we used, we also tried out the Android native code to see which different audio formats that were availible to us.

6.1.1 Libgdx music class

The music class is built into the libgdx library. It represents a streamed audio file, and allows us to play, stop, pause, and adjust volume of the music. It is created via the libgdx audio class, with the method ‘new Music(FileHandle file)’, where the currently supported file extensions are wav, mp3, and ogg [46].

6.1.2 Libgdx Sound class

As with the music class, the sound class is also built into the libgdx library - but with the difference that this one is also giving us methods for setting the pan and pitch, allowing us to do more dynamic changes of the sound. The creation and supported file extensions are the same as for the music class [47].

6.1.3 Android native audio system

The Android native audio system provides hardware support for the audio. This allows us to use different audio formats than the one we are provided with from the libgdx audio class, and it also increases the quality of the sound. All of the above codecs for libgdx are supported, as well as many others [48]. In 6.3 Our Music and Sound, we focus on comparing libgdx’ audio formats with the successor of MPEG Audio Layer III (mp3), which is Advanced Audio Coding (AAC), we think this will give us the most benfits because it is the latest compression method after mp3.

6.2 Audio formats

Waveform audio, Ogg Vorbis and MPEG audio layer III are all formats compatible with Libgdx audio classes. Advanced Audio Coding is a part of the Android native audio system. All of these will be described and presented here as candidates for the audio formats in our game, a brief history of each format is also presented. There are still other formats availible to us, but as you will see in the results, none of them would qualify to be a competitor to these formats.

6.2.1 Waveform audio

The waveform audio file format is more commonly known as wav. Though a wav file can hold compressed audio, the most common wav format contains uncompressed audio in the linear pulse code modulation(LPCM) format. The standard format contains two channels of 44,100 samples per second and 16 bits per sample. Wav audio can also be edited and manipulated with relative ease of use with many open source softwares [49].

6.2.2 Ogg Vorbis

Ogg Vorbis is an open source digital audio encoding format that uses lossy data compression. It is used to store and play digital music and is roughly comparable to mp3 and aac. Ogg Vorbis has been designed to completely replace all old patented audio formats. It is completely free to use, and distribute, with no cost at all[50].

6.2.3 MPEG Audio Layer III

MPEG Audio Layer III, also know as mp3, is a patented digital audio encoding format using a form of lossy data compression. It means that a part of the audiosignal is lost when compressing with the mp3 algorithm. The mp3 patent is owned by Frauhofer IIS, and was developed in 1991, at the University of Hannover, Germany [49].

6.2.4 Advanced Audio Coding

The advanced audio coding, also more known as aac, is designed to be the mpeg audio layer III's successor. It is patented, but there are no licenses required to be able to stream or distribute content in aac format. AAC compresses an audio file to almost half the filesize compared to mp3, while still maintaining same audio quality. AAC has been standardized by ISO and IEC as parts of mpeg-2 and mpeg-4 specifications [51].

6.3 Our Music and Sound

When we compared the four audio formats, we started by sorting them out in which order we belived they would have the smallest filesize from the above data (wav, mp3,ogg,acc). A comparison between an mp3 and a wav file gave as result that the wav required 180KB, while the mp3 only needed 20KB of storage. When choosing between mp3 and acc, we relied upon a test from coolutils.com [52], which said that the acc was a both smaller in filesize, and had better sound quality compared to mp3. However, acc is not supported by libgdx, and due to the complexity of the native code, we decided to select mp3 instead [53]. Finally we saw that when mp3 was compared with ogg, ogg had a smaller filesize with the same quality [54]. One other difference was that ogg is open source, and mp3 is limited with licences by its patent. Therefore, the best choice is ogg, which we use for both music and sound effects.

Due to that the increasing music and sound effects, it was neccesary to do some class that handled them, so we made our own audiomanager class which allowed us to insert and remove instances of a music or sound class into a list. This simplified starting and repeating a particular music or sound effect, for example in the pipeNodeList class who is controlling all the obstacles and pickups (see Appendix A.3).

6.3.1 Music

Fruity Loops for PC was used to create the music in Tisado. We used a calm sound so that the players can identify themselves as a lonely robot in space. FruityLoops also allowed us to manipulate pitch correction, harmonization and time stretching which we used. We then exported it into the Vorbis Ogg format [50].

6.3.2 Sound effects

An example of sound effects is when the player picks up a coin or crashes into an obstacle. The most prefereble format would have been the Vorbis Ogg format because of the small filesize, but we decided that even the wav format works. The tradeoff with larger filesize against easier editing would not matter because all sound effects together were already small compared to an mp3 music file.

7 Results

The resulting product of this project turned out relatively true to our initial vision. The planned game play features were all implemented, though some of our graphical intentions had to be excluded for various reasons.

The final version of this project, is a fully playable game in which the player travels along a pipe and dodges objects to survive as long as possible, as well as collecting coins and power-ups. The game world is randomly generated for every new game session. However, a static seed, for the random number generator, can be set to force the same level generation each time.

The graphics include multiple light sources, per-pixel shading, reflections, particle effects, and post-processing. We also implemented animations and reflections for the player model, and a simple, but effective, form of collision detection for obstacles and pickups. Furthermore, the heavy focus on optimization during the second half of our project, allowed for inclusion of several effects that were unachievable before optimizing the rendering process.

8 Discussion

The outcome of the project is, visually, not as appealing as our initial ideas and goals were. The reason for this, is that it was simply not possible to fit as much in a mobile phone as we expected, without spending countless hours on optimization. Something that the big game developers can afford, but we could not. We are, however, happy with how the game turned out. Even though various aspects of the game play might need minor tweaks, the game logic meets all the requirement we decided on early in the development process.

Using libgdx instead of a more powerful tool for development, such as Unity, turned out good for us. We are pleased that we were given the possibility to chose our own framework, and we are happy that we chose libgdx. Doing so led to us understanding the fundamental parts of game development more than we think we would have if we used Unity, or any other tool equal to Unity.

The only real setback we had during the development of this project was that a member of our group, the 3D artist, decided to drop out of the group early on, due to unknown reasons. Apart from this, developing a project of this size, as a group, turned out great for us. We set up an SVN server early in the development and any code changes were committed to this server. Of course, there were times where collisions would be caused by trying to commit to the server. But, because we worked in smaller groups with each problem, these collisions happened rarely and when they did, they were easy to fix manually. The way the workload was divided between us during the weekly meetings was perfect, and even if some things took longer than expected, we ended up completing the features for our prototype releases in time.

Although we have yet to receive a more public opinion of how entertaining our product became, we are very satisfied with the overall experience it provides. As of now, the game could use some more work to reach its full potential, though we are firm believers that it will eventually be ready for commercial release.

9 Conclusion

In order to make a fun and visually appealing game, within a short time limit, it is important to have a good and simple game idea. It is also important to distribute the workload evenly amongst the participants of the project. To make sure that the application runs smoothly on smartphones, it is important to consider the different ways of adding each graphical effect. The different types of lighting, flat shading, Gouraud shading and Phong shading, all have their own, different, advantages and disadvantages, and it is important to compare these to each other, to find the best available solution for the game in question.

Such comparisons and considerations are important to go through for each of the different graphical effects.

Knowing the hardware and how to optimize for the given platform is important as well. For Android phones, there are few parts that should be given extra attention when optimizings. These parts include garbage collecting, public methods for accessing private variables, and fillrate. If the game contains advanced and complex logic, this too should be carefully monitored and optimized.

References

[1] Feeney, C. Android and iPhone now hog 91% of Mobile OS Market Share [Internet]. 2012 Mar 6 [cited 2012 May 12]. Available from: http://www.jumptap.com/android-and-iphone-now-hog-91-of-mobile-os-market-share/

[2] Ruddock, D. iPhone 4S VS. Android Posted: Even The Galaxy S II Is getting Smoked (For Now, At Least) [Internet]. 2012 [updated 2012 Jan 16; cited 2012 May 12]. Available from: http://www.androidpolice.com/2011/10/11/iphone-4s-vs-android-benchmarks-posted-even-the-galaxy-s-ii-is-getting-smoked-for-now-at-least/

[3] Anthony, S. iPhone 4S benchmarks: Twice as fast as the Galaxy S II, Droid Bionic [Internet]. 2012 Oct 11 [cited 2012 May 12]. Available from: http://www.extremetech.com/computing/99379-iphone-4s-benchmarks-twice-as-fast-as-the-galaxy-s-ii-droid-bionic

[4] Bachmann, S. Why we switched to libGDX [Internet]. 2011 [updated 2011 May 8; cited 2012 May 12]. Available from: http://thegreystudios.com/blog/?p=30

[5] OpenGL ES 2.0 [Internet]. 2012 [cited 2012 May 13]. Available from: http://developer.android.com/resources/tutorials/opengl/opengl-es20.html

[6] Zechner, M. Libgdx [Internet]. 2012 [cited 2012 May 12]. Available from: http://code.google.com/p/libgdx/

[7] Games Build with Libgdx [Internet]. 2012 [updated 2012 May 11; cited 2012 May 12]. Available from: http://code.google.com /p/libgdx/wiki/LibgdxGames?ts=1332862877&updated=LibgdxGames

[8] OpenGL Lighting or How Light Sources Work (Long, In-depth Tutorial) [Internet]. 2012 [cited 2012 May 12]. Available from: http://www.falloutsoftware.com/tutorials/gl/gl8.htm

[9] List of common shading algorithms [Internet]. 2011 [updated 2011 Jun 7; cited 2012 May 14]. Available from: http://en.wikipedia.org/wiki/List_of_common_shading_algorithms

[10] Akenine-Möller, T, Haines, E, Hoffman, N. Real-Time Rendering. 3d ed. Natick, MA: AK Peters ; 2008. Chapter 5, Visual Appearance; p.115.

[11] Gouraud, H. "Continuous Shading of Curved Surfaces," IEEE Transactions on Computers, vol. C-20, pp. 623-629, June 1971.

[12] Phong, Biu Tong, "Illumination for Computer Generated Pictures," Communications of the ACM, vol. 18, no. 6, pp. 311-317, June 1975.

[13] Basics of Light in 3D Computer Graphics [Internet]. 2010 [updated 2010 Feb 10; cited 2012 May 12]. Available from: http://robertokoci.com/basics-of-light-in-3d-computer-graphics/

[14] About Shader Programs [Internet]. 2007 Mar 7 [cited 2012 May 13]. Available from: http://idlastro.gsfc.nasa.gov /idl_html_help/About_Shader_Programs.html

[15] de Greve, B. Reflections and Refractions in Ray Tracing [Internet]. 2006 [cited 2012 May 12]. Available from: http://graphics.stanford.edu/courses/cs148-10-summer/docs/2006–degreve–reflection_refraction.pdf

[16] Guinot, J. The Art of Texturing Using The OpenGL Shading Language – Environment Mapping [Internet]. 2006 [cited 2012 May 12]. Available from: http://www.ozone3d.net/tutorials/glsl_texturing_p04.php

[17] Post-Processing Effects [Internet]. 2008 [cited 2012 May 12]. Available from: http://www.leadwerks.com/files/Tutorials/CPP/Post-Processing_Effects.pdf

[18] Akenine-Möller, T, Haines, E, Hoffman, N. Real-Time Rendering. 3d ed. Natick, MA: AK Peters ; 2008. Chapter 10, Image Processing; p.467-468.

[19] How to do good bloom for HDR rendering [Internet]. 2006 May 20 [cited 2012 May 13]. Available from: http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/

[20] Gaussian Blur Filter Shader [Internet]. 2008 Oct 11 [updated 2008 Oct 11; cited 2012 May 12]. Available from: http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/

[21] Hämäläinen, H. Bloom lib [Internet]. 2012 Jan 21 [updated 2012 Mar 29; cited 2012 May 13]. Available from: http://www.badlogicgames.com/forum/viewtopic.php?f=17&t=3131

[22] Welcome To The World Of 3D Computer Graphics [Internet]. 2011 [cited 2012 May 12]. Available from: http://www.squidoo.com/computer-graphics-introduction#module147978701

[23] Russo M. Polygonal Modeling: Basic and Advanced Techniques [monograph online]. Plano, Texas: Wordware Publishing; 2006 [cited 2012 May 12]. Available from: Books24x7.

[24] Müller-Prove M. Sketchpad [Master thesis]. Hamburg: Department of Informatics, University of Hamburg; 2002 [cited 2012 May 12]. Available from: http://www.mprove.de/diplom/index.html

[25] Parent R, Computer Animation [monograph online]. Kingston, Ontario: Elseveier Inc; 2002 [cited 2012 May 12]. Available from: ScienceDirect

[26] Henry D. MD2 file format [Internet]. 2004 Dec 19 [updated 2004 Dec 19; cited 2012 May 12]. Available from: http://tfc.duke.free.fr/coding/md2-specs-en.html

[27] Samsung [Internet]. 2012 [cited 2012 May 13]. Available from: http://www.samsung.com/ie/mobile/featured-applications/game-hub.html

[28] Nordling, C, Osterman, J. Physics Handbook. 8th ed.Lund: Studentlitteratur; 2006.

[29] Akenine-Möller, T, Haines, E, Hoffman, N. Real-Time Rendering. 3d ed. Natick, MA: AK Peters ; 2008. Chapter 17: Collision Detection

[30] Ericson, C. Real-Time Collision Detection. Amsterdam; Boston : Elsevier; 2005.

[31] Designing for Performance [Internet]. 2012 [cited 2012 May 12]. Available from: http://developer.android.com /guide/practices/design/performance.html

[32] Watson, B, David Luebke, "The Ultimate Display: Where Will All the Pixels Come From?" Computer, pp.54-61, August 2005

[33] Android [Internet]. 2012 [cited 2012 May 12]. Available from: http://www.android.com

[34] Akenine-Möller, T, Haines, E, Hoffman, N. Real-Time Rendering. 3d ed. 2008. Chapter 15; p.699-700.

[35] The Industry Leader in .NET & Java Profiling [Internet] 2012 [cited 2012 May 13]. Available from: http://www.yourkit.com/

[36] Javin, P. How Garbage Collecting works in Java. [Internet]. 2011 April 11 [cited 2012 May 12]. Available from: http://www.javarevisited.blogspot.se/2011/04/garbage-collection-in-java.html

[37] Bruno, E. G1: Java's Garbage First Garbage Collector [Internet]. 2009 Aug 21 [cited 2012 May 12]. Available from: http://www.drdobbs.com/jvm/219401061

[38] Galaxy S II [Internet]. 2012 [cited 2012 May 12]. Available from: http://www.samsung.com/se/consumer/mobil/mobil/smartphones/GT-I9100LKANEE-spec

[39] Woligroski, I. Fill Rates [Internet]. 2006 July 31 [cited 2012 May 12]. Available from: http://www.tomshardware.com/reviews/graphics-beginners-2,1292-3.html

[40] Early Depth Test [Internet]. 2012 [updated 2012 Apr 29; cited 2012 May 12]. Available from: http://www.opengl.org/wiki/Early_Depth_Test

[41] Akenine-Möller, T, Haines, E, Hoffman, N. Real-Time Rendering. 3d ed. 2008. Chapter 14; p.664-667.

[42] Face Culling [Internet]. [updated 2012 April 29; cited 2012 May 12]. Available from: http://www.opengl.org/wiki/Face_Culling

[43] Akenine-Möller, T, Haines, E, Hoffman, N. Real-Time Rendering. 3d ed. 2008. Chapter 14; p.670-677.

[44] Crooks, C. Mobile Device Game Development [monograph online]. Hingham, Mass.: Charles River Media; 2004 [cited 2012 May 12]. Available from: Chalmers Library.

[45] Zechner M. Libgdx Features Homepage [Internet]. 2011 [cited 2012 Apr 30]. Available from: http://libgdx.badlogicgames.com/features.php

[46] Zechner M. Interface Music [Internet]. 2010 [cited 2012 Apr 30]. Available from: http://libgdx.l33tlabs.org/docs/api/com/badlogic/gdx/audio/Music.html

[47] Zechner M. Interface Sound [Internet]. 2010 [cited 2012 Apr 30]. Available from: http://libgdx.l33tlabs.org/docs/api/com/badlogic/gdx/audio/Sound.html

[48] Zechner M. Interface Audio [Internet]. 2010 [cited 2012 Apr 30]. Available from: http://libgdx.l33tlabs.org/docs/api/com/badlogic/gdx/Audio.html

[49] Viers R. The Sound Effects Bible Studio City. CA: Michael Wiese Production; 2008

[50] Waggoner B. Compression for great video and audio. Oxford: Focal Press; 2009

[51] ISO/IEC 13818-7:2006, Information technology - Generic coding of moving pictures and associated audio information - Part 7: Advanced Audio Coding (AAC) [Internet]. 2006 [cited 2012 Apr 30] Available from: http://jaadec.sourceforge.net/specs/ISO_13818-7_AAC.pdf

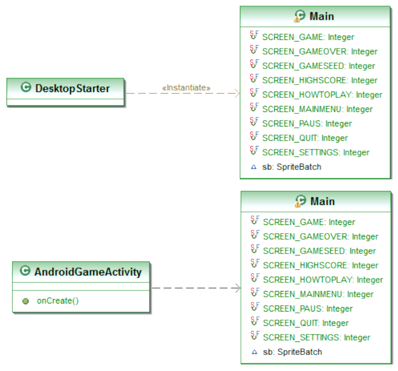
[52] MP3 VS AAC, or the best music file format for mobile phones [Internet]. 2012 [cited 2012 Apr 30]. Available from: http://www.coolutils.com/musicformobilephones

[53] Android developers Homepage [Internet]. 2012 [cited 2012 Apr 30]. Available from: http://developer.android.com/sdk/ndk/overview.html

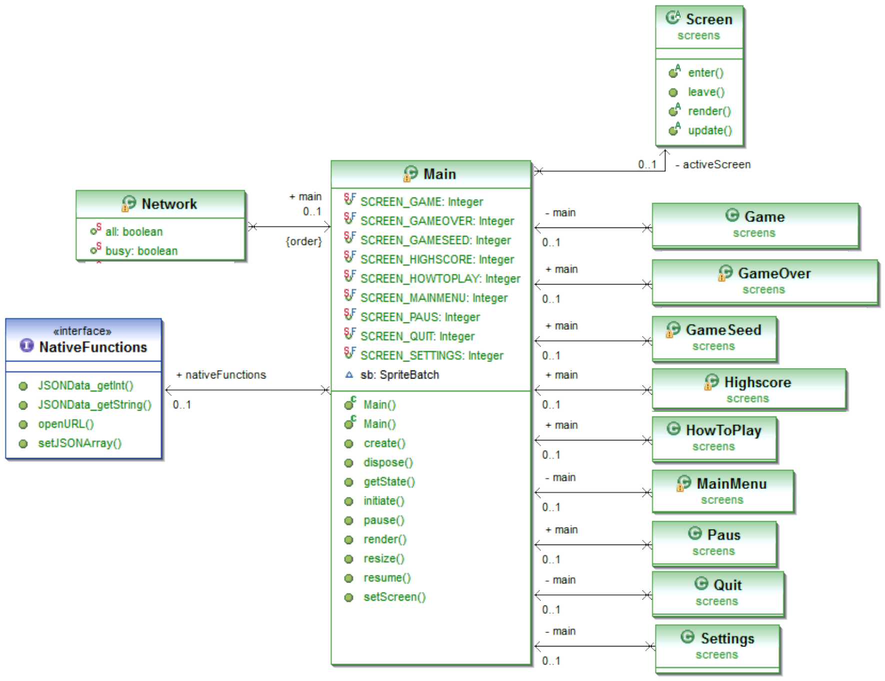
[54] Difference Between OGG and MP3 [Internet]. 2012 [cited 2012 Apr 30]. Available from: http://www.differencebetween.net/technology/difference-between-ogg-and-mp3/

A UML Diagrams

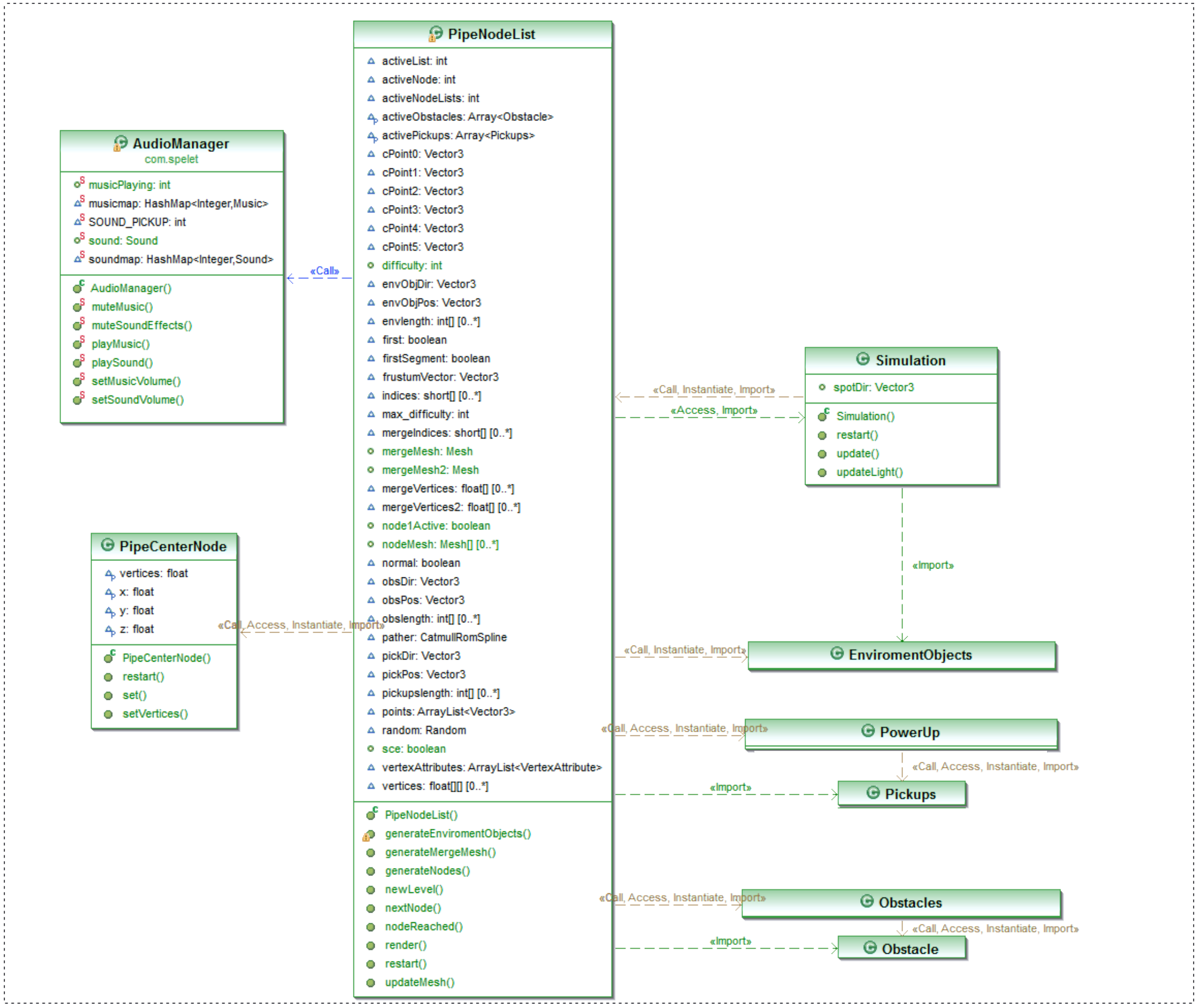
A.1 Libgdx Startup Classes



A.2 Main Class

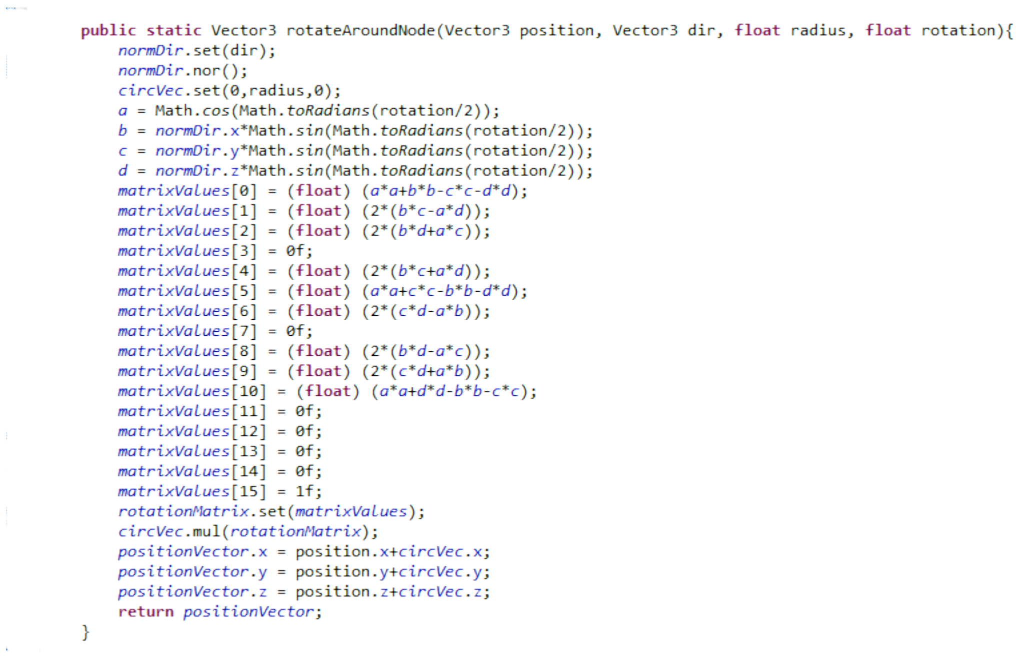


A.3 PipeNode Cass



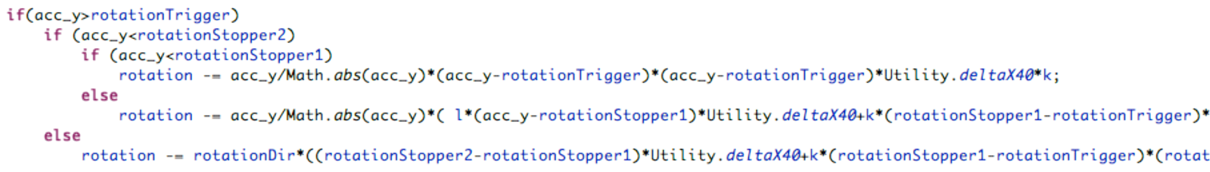
B Code Examples

B.1 Rotation Algorithm Aode



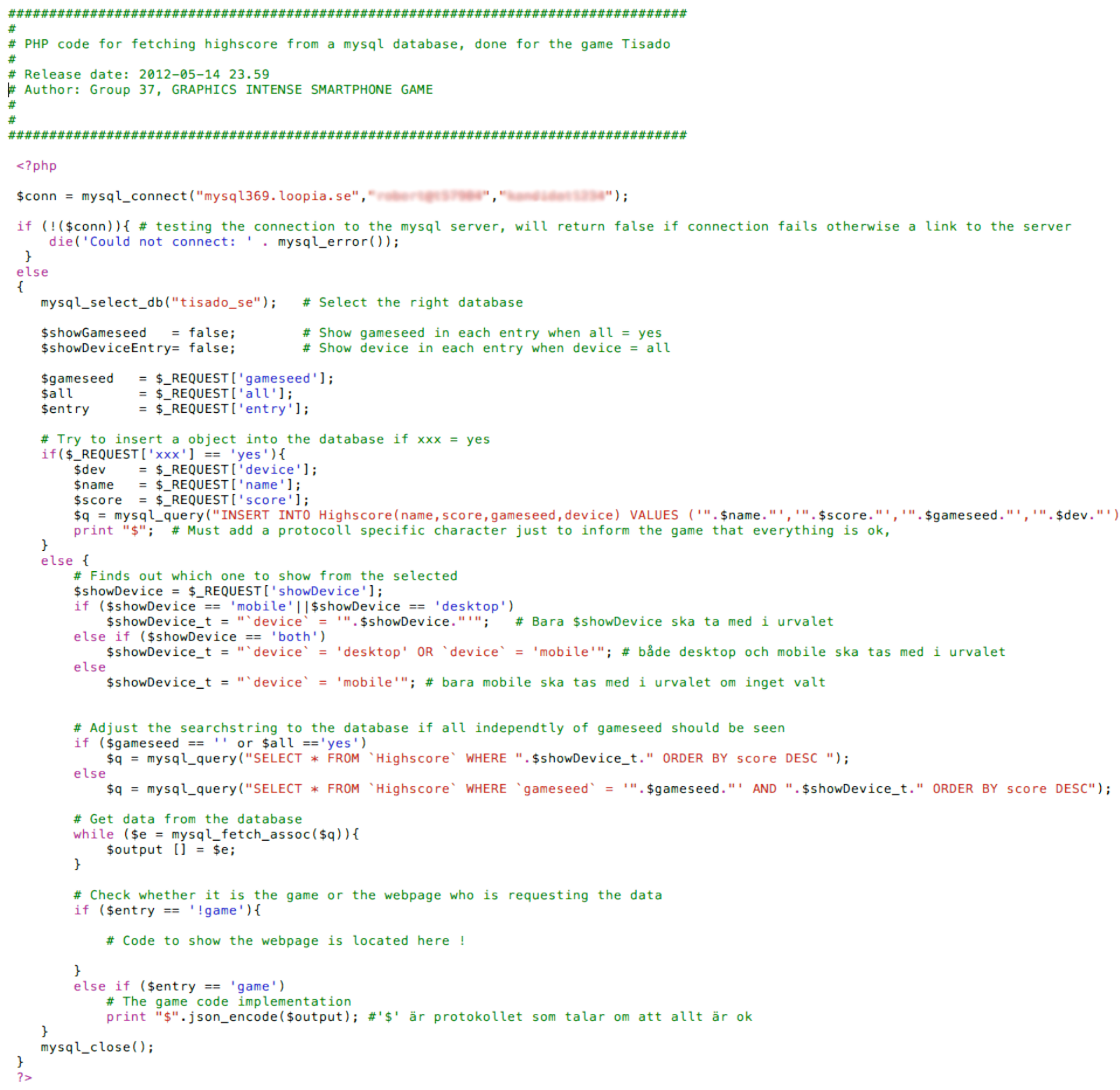
```
public static Vector3 rotateAroundNode(Vector3 position, Vector3 dir, float radius, float rotation){
    normDir.set(dir);
    normDir.nor();
    circVec.set(0,radius,0);
    a = Math.cos(Math.toRadians(rotation/2));
    b = normDir.x*Math.sin(Math.toRadians(rotation/2));
    c = normDir.y*Math.sin(Math.toRadians(rotation/2));
    d = normDir.z*Math.sin(Math.toRadians(rotation/2));
    matrixValues[0] = (float) (a*a+b*b-c*c-d*d);
    matrixValues[1] = (float) (2*(b*c-a*d));
    matrixValues[2] = (float) (2*(b*d+a*c));
    matrixValues[3] = 0f;
    matrixValues[4] = (float) (2*(b*c+a*d));
    matrixValues[5] = (float) (a*a+c*c-b*b-d*d);
    matrixValues[6] = (float) (2*(c*d-a*b));
    matrixValues[7] = 0f;
    matrixValues[8] = (float) (2*(b*d-a*c));
    matrixValues[9] = (float) (2*(c*d+a*b));
    matrixValues[10] = (float) (a*a+d*d-b*b-c*c);
    matrixValues[11] = 0f;
    matrixValues[12] = 0f;
    matrixValues[13] = 0f;
    matrixValues[14] = 0f;
    matrixValues[15] = 1f;
    rotationMatrix.set(matrixValues);
    circVec.mul(rotationMatrix);
    positionVector.x = position.x+circVec.x;
    positionVector.y = position.y+circVec.y;
    positionVector.z = position.z+circVec.z;
    return positionVector;
}
```

B.2 Rotation Speed Code



```
if(acc_y>rotationTrigger)
    if (acc_y<rotationStopper2)
        if (acc_y<rotationStopper1)
            rotation -= acc_y/Math.abs(acc_y)*(acc_y-rotationTrigger)*(acc_y-rotationTrigger)*Utility.deltaX40*k;
        else
            rotation -= acc_y/Math.abs(acc_y)*( l*(acc_y-rotationStopper1)*Utility.deltaX40+k*(rotationStopper1-rotationTrigger)*
    else
        rotation -= rotationDir*((rotationStopper2-rotationStopper1)*Utility.deltaX40+k*(rotationStopper1-rotationTrigger)*(rotat
```

B.3 Online Highscore PHP Code



```
###################################################################################
#
# PHP code for fetching highscore from a mysql database, done for the game Tisado
#
# Release date: 2012-05-14 23.59
# Author: Group 37, GRAPHICS INTENSE SMARTPHONE GAME
#
#
###################################################################################

 <?php

 $conn = mysql_connect("mysql369.loopia.se","[redacted]","[redacted]");

 if (!($conn)){ # testing the connection to the mysql server, will return false if connection fails otherwise a link to the server
     die('Could not connect: ' . mysql_error());
  }
 else
 {
    mysql_select_db("tisado_se");   # Select the right database

    $showGameseed   = false;        # Show gameseed in each entry when all = yes
    $showDeviceEntry= false;        # Show device in each entry when device = all

    $gameseed   = $_REQUEST['gameseed'];
    $all        = $_REQUEST['all'];
    $entry      = $_REQUEST['entry'];

    # Try to insert a object into the database if xxx = yes
    if($_REQUEST['xxx'] == 'yes'){
        $dev    = $_REQUEST['device'];
        $name   = $_REQUEST['name'];
        $score  = $_REQUEST['score'];
        $q = mysql_query("INSERT INTO Highscore(name,score,gameseed,device) VALUES ('".$name."','".$score."','".$gameseed."','".$dev."')
        print "$";  # Must add a protocoll specific character just to inform the game that everything is ok,
    }
    else {
        # Finds out which one to show from the selected
        $showDevice = $_REQUEST['showDevice'];
        if ($showDevice == 'mobile'||$showDevice == 'desktop')
            $showDevice_t = "`device` = '".$showDevice."'";   # Bara $showDevice ska ta med i urvalet
        else if ($showDevice == 'both')
            $showDevice_t = "`device` = 'desktop' OR `device` = 'mobile'"; # både desktop och mobile ska tas med i urvalet
        else
            $showDevice_t = "`device` = 'mobile'"; # bara mobile ska tas med i urvalet om inget valt


        # Adjust the searchstring to the database if all independtly of gameseed should be seen
        if ($gameseed == '' or $all =='yes')
            $q = mysql_query("SELECT * FROM `Highscore` WHERE ".$showDevice_t." ORDER BY score DESC ");
        else
            $q = mysql_query("SELECT * FROM `Highscore` WHERE `gameseed` = '".$gameseed."' AND ".$showDevice_t." ORDER BY score DESC");

        # Get data from the database
        while ($e = mysql_fetch_assoc($q)){
            $output [] = $e;
        }

        # Check whether it is the game or the webpage who is requesting the data
        if ($entry == '!game'){

            # Code to show the webpage is located here !

        }
        else if ($entry == 'game')
            # The game code implementation
            print "$".json_encode($output); #'$' är protokollet som talar om att allt är ok
    }
    mysql_close();
 }
 ?>
```