CHALMERS

The Jumping Dead

Development of a Graphics-Intense Smartphone Game

Bachelor of Science Thesis in Computer Science and Engineering

Emil Bryngelsson
David Dagson
Mathias Forssén
Jonatan Kilhamn
Nina Malm

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2013

The Jumping Dead
Development of a Graphics-Intense Smartphone Game

Emil Bryngelsson
David Dagson
Mathias Forssén
Jonatan Kilhamn
Nina Malm

Examiner: Sven Arne Andreasson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: The image on the cover can be seen show a typical frame from the game *The Jumping Dead*

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

This thesis presents the development of a graphics-intense smartphone game. With a team of five people for five months, a game engine is used to develop a 3D platform game for Android. The goal is to make the most of the graphics capability of the phone, while maintaining smooth gameplay at an acceptable frame rate.

To fulfil the requirements, a number of problems have to be solved. Light and shadows can be implemented in several ways, with different looks and costs. 3D modelling, texturing and animation is explored to learn to create good looking characters and decorations without overloading the GPU with triangles. The tools and effects used are tested with regards to the frame rate of the phone, and the results are documented in detail.

The resulting product is a platform game with animated models, interactive physics, and several types of light, shadows, and particle effects. The discussion is based around objective measurements of frame rate, in combination with the subjective opinions on the quality of visual effects used.

Based on the results, our conclusion is that a graphics-intense smartphone game can be made, but it has to be done with care. Considerations include the shader interaction with models and textures, minimising the number of triangles for rendering and storage in memory, and maintaining an even rendering and game-logic workload per frame to avoid stalling the game.

Sammanfattning

Denna kandidatuppsats behandlar utvecklingen av ett grafikintensivt spel för smartphones. Under fem månader arbetar fem personer med att utveckla ett 3D-plattformsspel till Android. Målet är att utnyttja telefonens grafikchip maximalt utan att det negativt påverkar flödet i spelet.

För att kunna nå målet så så behövde ett antal problem lösas. Ljus och skuggor kan implementeras på flera olika sätt, med olika utseende och kostnader. 3D-modellering, texturer och animationer undersöks för att kunna uppnå attraktiva figurer och dekorationer utan att överbelasta grafikprocessorn. Alla verktyg och effekter testades med avseende på frame rate på telefonen, och resultaten dokumenterades i detalj.

Den resulterande produkten är ett plattformsspel med animerade modeller, interaktiv fysik och flera olika typer av ljus, skuggor och partikeleffekter. Diskussionen är uppbyggd kring objektiva mätningar av bildfrekvens i kombination med subjektiva åsikter om kvaliteten på de visuella effekter som använts.

Utifrån resultaten är vår slutsats att ett grafikintensivt spel kan uppnås, men det måste göras med eftertanke. Hänsyn måste tas till, bland annat, hur shaders interagerar med modeller och texturer, minimering av det antal trianglar som renderas och sparas i minnet samt hur renderingen och spellogiken kan spridas ut på ett jämnt sätt för att förhindra att de negativt påverkar spelflödet.

Acknowledgements

We would like to thank our supervisor, Ulf Assarsson, for his pointers concerning graphical effects and his help with the report throughout the project. Further, we would like to thank the Department of Computer Science and Engineering for providing us with the smartphone that was used in the thesis. We would also like to thank bachelor groups 2 and 24 for providing us with invaluable feedback on the structure of our report. Finally we would like to thank the jMonkeyEngine and Blender communities for all their help and patience.

Contents

Dictionary

Some abbreviations, terms and definitions are stated here for reference and for ease of reading. Other technical terms are explained in the specific chapters where they are used.

the game - The game produced as part of this project.

the ghost - The in-game character controlled by the player.

the player - Refers to either the person playing the game or to the ghost, depending on context.

the phone - The phone used during development and for testing, a Samsung Galaxy S2 from 2011.

The Jumping Dead - The title of this project and name of the game. Most often refers to the game.

fps - Frames per second. A measurement of render speed often used in games.

resolution - Refers to the number of pixels on screen or on a texture, or the number of triangles of a 3D model.

cost - words like *costly*, *expensive*, and *intensive* are used in terms of processing power and memory unless otherwise specified.

Android - The Android OS for smart phones. Where version is concerned, refers to Android 4.1.2, which is the latest update for the phone, and was used during this project.

OpenGL - The Open Graphics Library. Where version matters, refers to OpenGL ES 2.0, the version used in Android on the phone.

JME - jMonkeyEngine 3.0 RC2. Refers to either of the SDK, the IDE or the game engine with this name.

shader - In computer graphics, a program that is run for each of a set of entities when rendering the scene. For the purposes of this report, the sets of entities are all vertices or all fragments.

vertex shader - A shader that is run for each vertex in the view, transforming their positions in three dimensions to positions on the screen.

fragment shader - A shader that is run for each fragment of the screen (commonly a pixel), performing any necessary calculations including determining the fragment's colour.

buffer - A temporary storage of data associated with each rendered fragment (commonly a pixel).

depth buffer - A buffer containing the distance between the camera and the closest surface at each fragment.

normal buffer - A buffer containing the normal of the closest surface point at each fragment.

colour buffer - A buffer containing the colour of each fragment. This is normally the buffer that is shown on the screen after rendering.

1 Introduction

The market of games for handheld devices has grown immensely the last couple of years. A recent study, conducted by the Entertainment Software Association, showed that the average household in the US owns at least one PC, smartphone, or dedicated game console (Entertainment Software Association 2012). The study also showed that a third of smartphone owners regularly use their phones to play games. There is therefore a large market for game developers in this field. This bachelor's thesis focuses on the development of a graphics-intense smartphone game for the Android platform.

1.1 Background

Computer graphics has for a long time been an important aspect of video games. Released in 1972, Pong was among the first video games produced for a mass market (Pongmuseum.com 2013). Computer graphics has seen major improvements since then and this development continues every year. The use of increasingly complex graphics in computer games calls for more advanced hardware, which can be seen as a contributing factor for the hardware industry to put out new and more powerful products at an increasing rate (Jörnmark, Axelsson, and Ernkvist 2005). Another field that is on the rise is the smartphone industry. Modern smartphones resemble pocket PCs more than regular phones and as hardware improves, so can the software developed for them as well. There are thousands of games available for smartphones and they, just like computer games, drive their hardware to support better and better graphics (Jörnmark, Axelsson, and Ernkvist 2005).

However, in contrast to desktop PCs, smartphones are built to be portable, which results in different requirements and restrictions on the hardware. Smartphones have much harder limits on their weight and size, and hardware for smartphones must be designed accounting for such constraints. Thus games developed for smartphones face different restrictions than the ones developed for PCs. For example, the amount of textures that can be cached in the GPU, the GPU speed, the CPU speed, and memory resources are considerably higher on modern computers compared to newer smartphones. Another important aspect is the difference in screen size and screen resolution between these categories of devices. As a consequence, visual details as well as flaws that would be visible on a computer screen are easily missed on a smartphone screen. These problems and restrictions are the focus of this thesis.

1.2 Purpose

The purpose of this project is to build a *simple yet functioning* and *graphics-intense* platform game for smartphones. Simple yet functioning means that the game will run with simple game logic and few features, but is in such a finished state that it could be played by someone outside the development group. Graphics-intense means that the game contains several graphical elements, such as 3D models, lighting, shadows, and particle effects, which are visually pleasing.

Furthermore, the aim of this thesis is to describe the graphical aspect of the development of the game, with focus on an analysis of different techniques available

for implementing different graphical elements.

1.3 Problem

Based on the purpose of the project, a problem specification was made addressing major concepts of the visual appearance of the game as well as the game concept and rules.

Object representation includes 3D modelling of objects, animation of the models, and texturing. The models and textures need to be adapted the screen size of the phone in order to limit their cost while still maintaining their attractiveness. Thus, Section 4 explores what level of detail is suitable for the device and the game concept used in this thesis.

Graphical effects are used to heighten the visual appearance of the scene, and includes implementation of several graphical elements which can be applied to represent the objects as well as possible. Using several graphical effects can result in an excessive use of the available hardware resources, which is why evaluation of different techniques in isolation as well as in combination with others is an important subject in game development. Section 2, Section 3, and Section 5 aim to answer the question of what is a good balance between the quality and the overall performance of the game.

Game logic is a subject where the physics and other rules of the chosen game domain are modelled. This also includes automatic generation of the level during gameplay, which involves adjusting the process in order to create a game which is neither too hard to play, nor so easy it bores the player. This is not the main focus of this report, but is covered briefly in Section 6 and Section 7.

1.4 Limitations

Developing a game is a large project which is normally not completed by five people in five months. Thus, two limitations were introduced to reduce the scope of the project.

Firstly, Android was chosen as the targeted platform since it would not be possible to develop the game for several platforms within the given time frame. The reasons for choosing Android over other platforms are the group members' previous experience with Java programming and that developing for Android has no associated costs.

Secondly, the development of *The Jumping Dead* was limited by the smartphone model, Samsung Galaxy S2, which was the one provided by Chalmers to be used for verification during this project. All software written had to be runnable on this device, and no claims can be made regarding other smartphone models.

1.5 Method

This section details how the project was conducted in order to achieve the visual appearance shown in the concept art in Figure 1.

Deciding which graphical effects should be implemented was an important aspect of this project. A literature study that analysed different graphical elements and

effects was made in order to facilitate the decision. The methodology used to choose which of those graphical elements and effects to be implemented, once they had been tested in the game, is covered in Section 1.5.1. In addition to the literature study, other tools were used to facilitate the development. The game was built entirely in the game engine jMonkeyEngine. The decision of which game engine to use, and the impacts of using jMonkeyEngine is further discussed in Section 7. Furthermore, Git, a version control system, was used to facilitate having multiple developers work on the project simultaneously. Finally, Blender was used for all the modelling work.

Figure 1 – Concept art for The Jumping Dead, used as a model throughout the development process.

1.5.1 Testing and comparing visual appearance

When evaluating the impact of different graphic effects, different perspectives were taken into account. Each feature included in the game needs to enhance the visual appearance while neither its computational cost nor its memory consumption are too high. The computational cost is measured in average frame rate over a few minutes. In order for an implementation of a graphic effect to be regarded as having an acceptable computational intensity, the average frame rate when it was applied needed to be at least 30 fps. The visual appearance is, on the other hand, measured using the subjective opinion of the group members.

If possible, these measurements are taken on the smartphone provided to the project. In other cases, such as when one implementation results in a black screen on the phone mentioned above but functions correctly on the computers used for the development, the measurements are done using computers to give an indicator of its performance and visual effect. In the later case, the name of the computer is stated in the text.

1.6 Description of the game

The Jumping Dead is a platform game in which a ghost jumps from platform to platform trying to get as far as possible. Jumping is accomplished by tapping the screen. The game ends when the ghost falls down, and the task is made harder by enemies pushing the ghost backwards. Power-ups can be collected to receive temporary benefits.

The setting is a stone church, which includes several elements to enhance the atmosphere. The walls are decorated with roman numerals indicating the player's progress, as well as windows, torches and plants. The enemies come in the shape of bats flying across the screen and wizards who fire colourful magic bolts at the ghost.

2 Light

Simulation of different aspects of light can be very important in games. Even very simple approximations of the behaviour of light help humans discern distances and the relative positions of objects. Colours can also set the mood in various ways (Küller et al. 2006). In a computer-rendered 3D environment, each object can be drawn in a uniform colour in order to be distinguishable from other objects in the scene. Introducing a more complex model for light, however, makes the scene look more realistic.

Several light models and techniques exist, ranging from models that trace individual beams of light as they are reflected and absorbed in the scene, which is too computationally expensive to be feasible for real-time rendering, (Filion 2011) to simplified models using a few constant colours for each object to approximate the illumination (Akenine-Möller, Haines, and Hoffman 2008).

The simplified model introduced above normally consist of three constant colours, namely ambient, diffuse, and specular (Wolff 2011). Ambient light is light with no direction which illuminates everything in the scene evenly (Akenine-Möller, Haines, and Hoffman 2008). This is supposed to approximate the real-world effects of light being repeatedly reflected in many diffuse surfaces in the environment (Akenine-Möller, Haines, and Hoffman 2008). Diffuse reflection models light that falls directly from a light source onto a surface (Akenine-Möller, Haines, and Hoffman 2008). Its strength is lowered as the light comes in close to parallel to the surface (Akenine-Möller, Haines, and Hoffman 2008). This corresponds well to reality, since the same amount of light is spread over a larger surface area if that surface is slanted away from the light source (Akenine-Möller, Haines, and Hoffman 2008). The concept of specular reflection is that most of the incoming light continues in the direction reflected in the normal (Akenine-Möller, Haines, and Hoffman 2008). The surface is thus drawn with different brightess depending on the position of the viewer. This can simulate shiny materials where one can see the reflection of a light source (Akenine-Möller, Haines, and Hoffman 2008).

2.1 Lighting techniques

There are a number of techniques to achieve lighting by using the simplified model described above. Firstly, light can be calculated by utilising either the fragment shader or the vertex shader, resulting in different computational costs and smoothness. Furthermore, light can also in some cases be precalculated when the illumination from a light souce can be predicted. This section will describe different techniques for modelling light.

2.1.1 Reflection models

Both diffuse and specular reflection use the fragment's surface normal in their calculations. However, as per Section 4.1, all surfaces are divided into flat triangles with sharp angles between them. If the entire triangle is considered to have the same normal, the resulting shape will have sharp division between edges. This is called flat shading (Akenine-Möller, Haines, and Hoffman 2008).

To make the surface seem smoother, the normal is defined differently for each vertex of the triangle and interpolated for all fragments in-between (Akenine-Möller, Haines, and Hoffman 2008). The method used in *jMonkeyEngine 3.0 (JME)* is called *Phong shading* (jMonkeyEngine 2013b) and simply interpolates the surface normal in every point from the three vertices. This means that the fragment shader has to perform light calculations for each fragment (Akenine-Möller, Haines, and Hoffman 2008).

Another method that is computationally cheaper is *Gouraud shading*. It performs the light calculations for the vertices only, and then lets the fragment shader interpolate the resulting colour for all points in-between (Akenine-Möller, Haines, and Hoffman 2008). Neither flat shading nor Gouraud shading renders specular highlights as accurately as Phong shading (Akenine-Möller, Haines, and Hoffman 2008). All three variants are shown in Figure 2.

Figure 2 – The same sphere rendered using, from left to right, flat shading, Phong shading, and Gouraud shading.

2.1.2 Precalculated light

The methods presented above can be used to determine the effects of light dynamically for each frame. If the light sources or the illuminated objects move, the lighting will change accordingly. However, this is not always necessary. If neither the light source nor the object to be illuminated moves, the light calculations will give the same results each frame.

In the latter situation, the illumination can be determined once and then re-used. This is done by storing precalculated values in a semi-transparent texture, which is brighter closer to the light source. Quadrilaterals with these textures are then placed between the light source and the object that is illuminated, which gives a realistic result if the object is flat.

There are other concerns to note when using precalculated light. Since the static texture does not look different when seen from different camera positions, it does not model the behaviour of specular reflections. Another consequence of using precalculated textures are that light calculations involving the light source can be omitted if no other objects than those the texture is applied to is affected by the source. Furthermore, if the objects the texture is applied to are not illuminated by other light sources, they can use an unshaded material. Such a material is exempt from all light calculations, and is always drawn in the same colour with a constant brightness (jMonkeyEngine 2013).

2.1.3 Background illuminated objects

A problem with using unshaded materials is that the objects for which it is used are not affected by background illumination. Thus, the objects appear brighter than other objects in space if the background illumination is not completely bright. Moreover, if the background illumination is altered, the colour of the object will not be updated, thus creating a colour contrast to the other objects.

One solution to this problem is to send a variable holding the ambient colour of the scene to the shader. During rendering, the shader then multiplies the colour of the object by the colour of the background illumination.

2.1.4 Multi-coloured light sources

Some light sources may have different colours in different directions. One example of this phenomenon from the game is the light transmission through the coloured window glass that causes objects to be illuminated with different colours depending on which parts of the window they are close to. If a light source with a single colour is used, the errors in colouring may be apparent.

Several methods for modelling the incoming light from multi-coloured light sources exist. A simple approximation of this phenomenon is to use several light sources with different directions and colours. However, this might lead to a large amount of light sources appearing simultaneously. Another option is to extend the colour attribute of the light sources with a texture describing its colours in different directions (Akenine-Möller, Haines, and Hoffman 2008). The relative positions of the point to be illuminated and the light source centre can then be used to obtain a coordinate to perform a texture lookup.

2.2 Light in jMonkeyEngine

In JME, support of a simplified shading model using ambient, diffuse and specular lighting is included in the engine. The object to be illuminated is set to use a material (see Section 4.3) that supports light (jMonkeyEngine 2013b). Then, when a light source is added to the scene, the lighting calculations are handled automatically.

There are several classes of lights implemented in JME. For background illumination, the game uses a *directional light*. As with ambient light, it reaches every point with the same colour and intensity, but it also has a direction. Lighting variations are introduced by adding *point lights*, which are shining outwards from one point in all directions, and *spotlights*, which have cone shaped ranges. Different placements of these types of light sources, along with precalculated light sources were tried. The following sections will describe the different alternatives that were implemented, and their effects.

2.3 Results

When modelling the light from the light sources in the game, several techniques were used to decrease the computational intensity of the game while aquiring the visual effects of light. These techniques included the functionality available in JME,

unshaded objects, precalculated lighting, and colour textures. Finally, a solution using unshaded materials being dependent on background illumination for the walls and windows, precalculated lighting for modelling the illumination by the torches and windows on the walls, point lights with a lower range for modelling the illumination from those lights on other objects than the walls, and colour textures to model the multi-coloured nature of the window lights. In this section, the alterations of the lighting models are depicted in sequence. Thus, the settings used for a specific measurement are always the last settings portraied with the alterations that are described along with the presentation of the measured value. Thus, the last described settings are the ones used in the game.

2.3.1 Performance with many light sources

Initially, point lights were placed in all torches and windows, and spotlights were used to brighten up the wizards, which added a high computational cost. In addition to providing more depth and variety to the scene, the lights gave the impression of activity, especially since the wizards' light sources followed them. With these settings, an average of 4.4 light sources, not counting the directional background light, was used simultaneously. Compared to the 47.8 fps obtained when only the background light was used, this lowered the frame rate to 18.8 fps.

Furthermore, if the wizards' spotlights were removed, the average number of light sources per scene was reduced to 2.4. As a consequence, the frame rate increased to 29.4 fps. Despite using less light sources, the presence of lights changed the visual appearance of the scene, as can be seen in Figure 3, which depicts the same scene with and without these lights. However, Figure 3b also shows how the light emitted from the fire of the torch reaches farther than seem realistic, thus not only affecting the ghost, but also the flower pot and the platforms. The reason for and a solution to this problem is described in Sections 2.3.3 and 2.3.4.

Moreover, measurements also showed that the range of the light sources as well as the frequency with which they appear affect the frame rate. For example, in order to execute at a frame rate of 35 fps or higher, a maximum of 0.9 point lights with radius of 75 JME world units[1] could be used, but if the radius was decreased to 50 world units, 1.9 point lights could appear simultaneously.

2.3.2 Shading of the background

Shading of objects is calculated for each fragment in JME, thus making the frame rate dependent on the size and number of objects for which light calculations are made. One large object, compared to the size of the platforms and other game objects, for which light is calculated is the background wall.

Making the wall *unshaded*, in other words not performing any light calculations for it, greatly improved the performance of the game. With the settings described above, where light sources were placed in torches and windows, but without illuminating the wall, the frame rate increased from 29.4 fps to 40.1 fps even if the average number of light sources per frame was increased to 3.2. Using the second

[1]The length unit used in the code. For reference, the platforms in Figure 4 are 24 world units long.

(a) Background illumination only.

(b) Light sources placed in torches and windows.

Figure 3 – Comparison showing the effect of placing light sources in the torches and windows.

option described in Section 2.1.3 enabled the wall to be affected by changes of the background illumination. These extra computations did not have any notable computational cost, as the game ran at 43.6 fps with an average of 3.0 light sources per frame. However, both these options removed the illumination of the light sources on the wall.

In order to solve the problem of the absence of illumination of the wall, precalculated light textures were added to the scene. The textures replaced the light sources of the torches, as described in Section 2.1.2, and were used to model the illumination of the wall by the window light. Thereby, the light sources of the windows were not removed since they were needed to illuminate the ghost and other objects passing nearby in the scene. The visual effects of the result is depicted in Figure 4, which compared to Figure 3b add a more prominent illumination of the wall by the light sources, but also gives a darker background of the scene. Due to this approach, the amount of simultaneously used light sources was decreased to 1.1, and the frame rate was increased to 50.6 fps. These techniques are described more in detail in Sections 2.3.3 and 2.3.4 below.

2.3.3 Modelling of torch lights

In the first implementation, the torches used point light sources illuminating both the wall and any passing objects. However, this modelling caused three different problems that resulted in modifications to the model. Firstly, JME's implementation of lighting requires that at least one vertex of a triangle needs to be within the range of the light source in order for any part of the object to be illuminated. This means that in order to affect the wall, the radius of the point light had to be at least 50 world units, which lead to high computational cost and lit objects farther away from the fire than what seemed realistic. Secondly, fire produces a flickering light in reality. When using an ordinary point light source, such flickering will not occur. Thirdly, as the previous results in Section 2.3.2 showed, computing the colour of the wall is computationally expensive and as a result the wall was made unshaded

Figure 4 – Usage of light textures and unshaded wall material for modelling the illumination of the wall by the light sources.

to increase the frame rate. As described above, this resulted in the torch light not affecting the wall.

In order to illuminate the unshaded wall with a light that more closely resemble real firelight, a precalculated light texture which was made to flicker by changing its opacity each rendered frame was used to replace the torch light source. The game ran at 50.6 fps when using this technique, which is considerably faster than using a light source, which ran at 29.4 fps, while also producing a more realistic result. However, as the texture only models the illumination by the fire on the wall, the fire will not illuminate the ghost or other objects that may move close to it.

To account for the illumination on nearby objects, a light source was added to the scene in addition to the texture. Since this light source only modelled the illumination of small objects close to the source, but not the illumination of the wall, it was given a smaller radius of 10 world units, which is the same size as the light texture on the wall. This option lowered the frame rate to 47.9 fps. The different models of the torch light is shown in Figure 5.

(a) A point light source with a 50 world unit radius.

(b) A pre-rendered lighting texture.

(c) A texture and a 10 world unit radius.

Figure 5 – Different variants of light from a torch.

2.3.4 Modelling of window light

The modelling of the windows was affected by the shape of the window, the brightness of the glass due to the sun light shining through the window, and the colourisation of the light being transmitted through the window glass. The following sections will describe the different alternatives that were examined when modelling the windows.

Multi-coloured window lights

Due to the light transmission through the coloured window glass, objects should be illuminated with different colours based on which part of the window they are close to. Since the light sources that are available in JME only can have one colour, they cannot be used as a realistic solution to this light behaviour.

As described in Section 2.1.4, an option for modelling this is to use multiple light sources. However, using a large amount of light sources was shown to be unfeasible for real-time rendering on the phone (see Section 2.3.1).

Another option is to use a precalculated wall lighting texture, which is applied to the wall. As this option replaces the light sources, it produced an apparent increase of frame rate to 53.4 fps. However, this option did not result in any colourisation of the light which illuminates other objects than the wall, such as the ghost or nearby platforms.

A third option that amends both these problems involves adding a texture describing the colours in different directions of the point lights as described in Section 2.1.4. In order to not substantially strain the GPU's texture cache, the used textures had a size of 4×5 pixels.

Due to the fact that some graphic cards only can transfer 32 floats into the fragment shader, some simplifications had to be made in order to be able to use the colour texture. Firstly, the specular colour had to be replaced with a specular brightness, which is a float describing the difference in intensity between the specular and diffuse colours. Thus, the specular colour was approximated in the fragment shader by multiplying the diffuse colour by the computed brightness, which may not give the correct specular colour. Another difference is the calculated of the distance vector between the point being shaded and the light source centre. Instead of using the centre of the light source and the position of the fragment, the calculation was made per-vertex in the vertex shader and then interpolated.

In this solution, the window closest to any moving illuminated object is found simultaneously as the ordinary traversal of the light sources, which is used by JME to find the colour and position of the light source closest to the different vertices of the object, thus not increasing the computational complexity of the application. However, this method occasionally selected the wrong window, thus resulting in the light calculations being made with the wrong colours. The static objects received the colour texture of the closest window when they were created, thus illiminating the possibility of receiving an incorrect texture. When this implementation was used for the ghost and the platforms, that is other objects were only affected by the standard colour of the light source, the game ran at 48.3 fps, which can be compared to the 52.7 fps that was measured when single coloured light sources of the same

size and frequency were used.

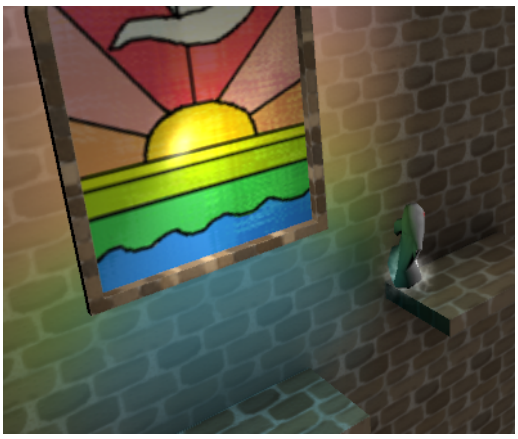
Correct shape of the light range

The window lights should be narrower horizontally than vertically due to the window shape, but the point lights have a spherical range.

The solution used was to place the light source in the lower part of the window, which resulted in it reaching farther below the window than above it. As a result, objects in the top half of the window would not be illuminated by the light source. Furthermore, the the coordinate used for the colour texture lookup had to be altered to compensate for the difference between the centre of the light source and the centre of the window. Due to this repositioning and the small horizontal size of the window, the radius of the point light could be decreased from 75 to 23.5 world units, which despite of the multi-coloured light source increased the frame rate to 54.3 fps. This frame rate is higher than the 52.7 fps measured using the larger, single coloured light source, but adds the visual effect of the platforms and the ghost being illuminated with different colours depending on their relative position to the window.

The difference between not using multi-coloured light and using it is shown in Figure 6, where the colurisation has an apparent effect on the platforms. The impact on the colouring of the ghost is more notable when it is moving compared to when seen in an image, as in Figure 6a. Thus, this colourisation is shown from an angle in Figure 6b, where the transition from green lighting to blue is notable.

(a) Multi-coloured window light seen from the front.



(b) Multi-coloured window light seen from the side.

Figure 6 – Effects on the ghost and platform illumination by a 23.5 world unit multi-coloured window light.

Shading of the window

If an ordinary shading model was used for the window, both the glass and the window frame were affected by other lights in the scene. As a consequence, this resulted in the glass being darkened by the directional light, as shown in Figure 6a above.

To amend this, the window could either be set as glowing or using an unshaded material. However, this lead to the brick frame being too bright, as with the wall in Section 2.3.2. This visual effect of using an unshaded material for the window frame is shown in Figure 7a.

Another option was to extend the shader used to make the wall illumination dependent on the directional light with an affection texture that describes how much different parts of the object should be affected by the background illumination. For the windows, a texture as small as 2×1 pixels could be used for describing that the brick frame should be affected, but that the glass should not. As portrayed in Figure 7b, this resulted in the glass of the window to appear to glow due to outside lighting while the bricks did not. However, when tested on the phone, the texture lookup always resulted in a value depicting that the brick frame should not be affected by the background illumination, thus leading to the same result as when using an ordinary unshaded material. In contrast to when using this option for the wall, the simplification of only taking the background illumination into account did not increase the frame rate, which was decreased from 54.3 fps to 52.3 fps.

(a) Use of the standard lighting material in JME, thus not taking background illumination into account.

(b) Use of an unshaded material with an affection texture. This option takes the changes of background illumination into account for the brick frame, while ignoring it for the glass.

Figure 7 – Comparison of unshaded window frames when taking background illumination into account and when not taking it into account.

2.4 Discussion

Initially, ordinary point light sources were used to model window light. This approach resulted in several visual problems as well as a very computationally expensive model. Firstly, the light sources had to have a large radius in order to illuminate the wall, which was both computationally expensive and illuminated objects farther from the light source than appeared realistic. Secondly, the light sources did not model colourisation of the light being transmitted through the church windows, nor did they model the characteristics of fire.

Instead, multi-coloured light sources positioned in the lower halves of the windows, smaller torch lights, and the use of light textures depicted reality better while

having a lower computational cost. The texture behind the church windows along with objects positioned in the upper half of the window were not illuminated by the light source were elements which lowered the realism of the model. However, the texture was still regarded to provide a visually pleasing effect, and the error of the lack of illumination was not prominent since objects seldom are positioned in that area of the window. However, these multi-coloured light sources only affects the ghost and the platforms. Thus, the wizards, plants, and other objects are coloured in the original light blue colour of the light source if they are positioned close enough to it. This decreases the realism further, but is not regarded as a problem since the ghost is the objects that the player focuses on. The multi-coloured nature could be activated for other objects as well, which should not lower the frame rate excessively based on the measurements taken when this light behaviour was activated for the ghost and the platforms, but this have not been tried. Moreover, the colour texture of a window which was not the one closest to the player was occasionally selected, which resulted in incorrect illumination of the ghost if the window closest to the ghost and the selected window were of different types. However, this seldom happened and was thus not regarded as a prioritised problem.

Furthermore, the simplifications made to calculations to accommodate for some graphics cards only being able to transfer 32 floats into the fragment shader produced no errors for the models used in the game, although the approximations could theoretically produce incorrect distance vectors and specular colours. This is because the specular colours of the platforms and ghost was a brighter variant of the diffuse colour, thus enabling the colours to be modelled correctly with only a diffuse colour and a value describing the difference in brightness between the specular and diffuse colours.

Using unshaded materials to take the background illumination into account proved to be an inexpensive alternative to shaded materials. An option to using an unshaded material to darken the window frame is to colour the texture for the window glass in a brighter colour. However, this will result in the colours of the glass appearing desaturated due to the grey background illumination. However, the phone used the affection texture incorrectly, but the result was regarded as better than letting the whole window being dependent on the background illumination. Finally, the use of an unshaded material for the windows was more expensive than using a material taking all lights into account, which was an odd result as this solution requires less computations. This may be a result of the problems with the measurement method discussed in Section 9.

3 Shadows

The model for light used in section 2, hereafter referred to as the simple light model, consists of three colour components per object, and have a problem in that it tends to produce a flat result (Wolff 2011). After having implemented lights, shadows can be added to the model in order to add visual appeal and realism. However, the representation of shadows in computer graphics is very often a concern separated from illumination. In reality, shadows are a consequence of the fact that light does not pass through opaque objects. There are physically correct rendering models, but compared to the simple light model, those run much slower and are known to be unfeasible for real-time rendering (Filion 2011).

For the simple light model, used in JME, shadows are also crude approximations. Two types of shadow models were considered for The Jumping Dead: *drop shadows* and *ambient occlusion*. Drop shadows are the darker area cast by an object occluding the light source. Ambient occlusion is on the other hand a model of shadows that are caused by nearby objects (Wolff 2011). There are other types of shadows and shadowing techniques than these two, but none of them were considered for The Jumping Dead.

This section will outline previous work and knowledge on drop shadows and ambient occlusion. Then follows a description of the implementation of drop shadows in the game, followed by the same regarding ambient occlusion. Both these results also include some discussion along with the conclusions that can be drawn.

3.1 Drop shadows

To create a drop shadow, the basic task is calculating which objects are lit by a given light source, and which are occluded. This information can be sent to the fragment shader, which can then draw the shadowed fragments in a darker colour. There are different methods that can be used to calculate which points that should be shadowed, and they will be explained in this section.

Before moving on to different methods to do so, one thing should be noted. Because of the very simplified model of light, this section only concerns *drop shadows*, i.e. shadows that fall from one object onto another. There are other shadow-like effects that are captured when using ray-tracing, but not when using the simple light model (Akenine-Möller, Haines, and Hoffman 2008, pp. 284–285). An approximation of these can be achieved through *ambient occlusion*, which is described in section section 3.2.

3.1.1 Projection shadows

For shadows of meshes on planes, there is a very basic method called *projection shadows*. Each object casting a shadow is projected through the position of the light source onto the plane to be shadowed. This gives a two-dimensional object which can be rendered as a shadow. One of the drawbacks of this method is that only planar surfaces can be shadowed. Furthermore, those planar surfaces must be determined beforehand; the program cannot by itself determine where to cast the

shadow. The restriction to planes also means that objects in general can not cast shadows on parts of themselves (Akenine-Möller, Haines, and Hoffman 2008).

3.1.2 Shadow-mapping

Another method is to shadow each fragment in space based on the information in one or several *shadow maps*. Such a map is generated each frame by rendering the entire scene one from the viewpoint of each light source instead of the camera. Instead of drawing to the screen, the program only marks points as illuminated or shadowed based on whether they can be seen from the light. Then, in the ordinary render pass from the camera's viewpoint, this information is used to determine whether to shadow each point.

One drawback of this approach is that the resolution of the shadow map — that is the number of bitmap elements used in the first render pass — limits the smoothness of the shadows (Akenine-Möller, Haines, and Hoffman 2008). Since many points may be close to the camera but far from the light source, using the same shadow map resolution as the screen resolution means some shadow edges become rough. At the same time, increasing the shadow map resolution means computing time is spent on determining the shadow status of several points, far from the camera, that all end up in the same pixel on the screen.

A way to partially mitigate this, which is used in JME, is *parallel-split shadow mapping*. The idea is to split the view frustum into segments of different depth, and then using a different shadow map resolution for each segment. This way points close to the viewer receive high-resolution shadows, and less time is devoted to points farther away.

3.1.3 Shadow volumes

Another method entirely is to use *shadow volumes*. It is based on the idea that the total shadow cast from an object can be seen as an infinite volume with one face towards the light, and faces on all sides running parallel with rays cast from the light source. To determine if a point is shadowed, simply check if it is inside at least one shadow volume. This is done by following a path from the camera to the point of interest, and count the number of shadow volumes entered and exited. Advantages of shadow volumes include the fact that the shadows are drawn in the same resolution as the rest of the scene (Akenine-Möller, Haines, and Hoffman 2008).

3.1.4 Results

A complete implementation of drop shadows is available in JME, including variants for shadows from directional lights, point lights and spotlights. As previously mentioned, this implementation uses parallel-split shadow mapping.

Tests were made with shadows only cast by the ghost and the platforms, and only received by the platforms and the wall. The visual results are showed in Figure 8. Using a high-resolution map, 4096×4096 pixels, the shadow edges are very sharp. A lower resolution, 512×512 pixels, results in jagged shadow edges. These sawtooth-like corners are the pixels of the shadow map, as that map is layered on the scene

from the viewpoint of the light source. Using the lowest resolution, 128×128 pixels, the shadows are also blurred-out which mediates the jagged look that would otherwise result.

This shadow rendering is computationally expensive. With other settings such that the laptop computer used for testing ran at an average of 68 fps [2], test cases similar to Figure 8 were tried. The 128×128 pixel shadow map version ran at 44 fps; the 512×512 one at 38 fps, and a 2048×2048 pixel shadow map version ran at 31 fps. The biggest one from Figure 8, 4096×4096, made the game freeze for about a minute and then continue at less than one frame per second.

On the phone, all test cases resulted in a black screen as soon as the actual game was started. The application would keep running, as evident by the music playing,

Finally, another problem was noted when testing on a computer using the JME implementation. The drop shadows would not be shown through any transparent object. The plant models are an example of transparency in the game, since they are made by drawing a texture of leaves on transparent background to a flat surface. Through the leaves the wall could be seen, but any shadow cast on that wall disappeared.

3.1.5 Discussion

The purpose of adding shadows to the game was to achieve a more realistic look and help the player see the platforms clearly. The restriction was, as with all other graphical effects, that the game would run on the phone at a satisfactory frame rate. A working implementation of ambient occlusion was already present before drop shadows were thoroughly investigated. Since the latter would then be brought into the game when it already contained lights and ambient occlusion, the change would only add a small amount of realism. For that reason, the inclusion of drop shadows was abandoned as it seemed to be a very time-consuming task.

The only implementation that was tested was JME's own. However, writing a

[2] It should be noted that the game runs differently on different computers, and so these numbers only tell anything in relation to each other. In this section, the laptop model used was a Macbook Aluminium Unibody, 2009.

(a) Resolution: 128×128 pixels. **(b)** 512×512 pixels. **(c)** 4096×4096 pixels.

Figure 8 – Drop shadows when using jMonkeyEngine's default implementation set to different shadow map resolutions.

new implementation of shadow mapping did not seem worthwhile. This was because this effect and numerous others showed that the GPU's texture cache, where such a shadow map would have to be stored, could not handle very big textures. The results shown in Figure 8 indicated that the resolution of any shadow map providing satisfactory shadows would be past the limit.

The other variants for drop shadows mentioned above, projection shadows and shadow volumes, did not seem to be good candidates either. As the forays into ambient occlusion detailed in Section 3.2.2 made clear, creating and handling semi-transparent shadow objects and placing them on the walls was tedious and complicated enough when those shadows were not to move or change shape. Shadow volumes was also deemed too complicated to implement compared to other effects.

Drop shadows are easy to achieve in JME, but that implementation does not work on the Android platform. There may be a way to adapt the implementation specifically for Android, but that would have to wait to another project than this one. If no other shadowing effect were used, the scene would greatly benefit from including drop shadows. However, in this case ambient occlusion had already increased the level of realism greatly meaning drop shadows were not a priority.

3.2 Ambient occlusion

Ambient occlusion is an approximation of light attenuation due to occlusion by nearby objects, such as the shadowing in corners, edges between floors and walls, and in wrinkles of clothing (Wolff 2011). It is known to add a noticeable amount of quality, depth, and realism to the rendered image (Wolff 2011).

There has been research in the area previously, which has resulted in a number of techniques for modelling ambient occlusion with different amounts of accuracy and time consumption. As the research often is conducted on computers whose hardware resources, such as GPU texture cache, GPU speed, and CPU speed, are considerably higher than those available on smartphones, some techniques created for real-time rendering are not feasible for that purpose on smartphones.

3.2.1 Ambient occlusion techniques

Accurate modelling of the accessibility[3] of a point can be done by tracing rays emanating from the point and directed in its upper hemisphere (Wolff 2011). If a ray collides with any surface within a certain distance from the point, the surface occludes the point in that direction (Wolff 2011). The accessibility of the point is then proportional to the fraction of rays that are not occluded (Wolff 2011). Figure 9 portraits this concept.

As this method involves tracing a large number of rays from each point, it is known to be very computationally intensive (Wolff 2011; Akenine-Möller, Haines, and Hoffman 2008). Thus, much of that computation is avoided or approximated in real-time rendering (Akenine-Möller, Haines, and Hoffman 2008). Three categories of different methods available for real-time rendering of ambient occlusion are presented below.

[3]The amount of ambient light a point receives (Wolff 2011).

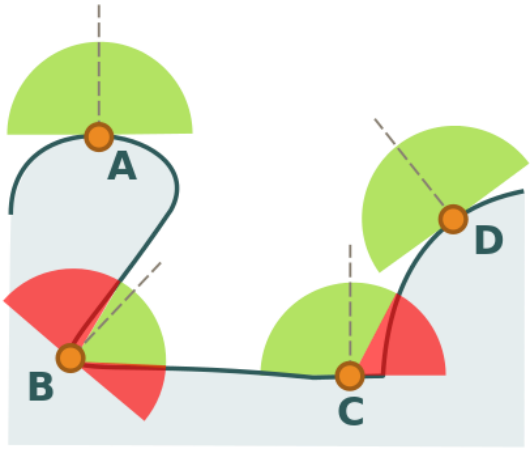


Figure 9 – Concept of the ambient occlusion model in 2D space. The light blue area represents an object while the white area represents air. The circles are different points that are evaluated, the dashed lines their normals, and the semicircle their upper hemispheres. Rays that are occluded are directed in the red parts of the hemisphere, while rays that are not occluded are found in the green parts. Points A and D will not be occluded at all, while C will have a lower accessibility value, and B will be very occluded.

Precomputed ambient occlusion

With the simplification that all indirect light is equally distributed in space, ambient occlusion is independent of the position of the light sources (Wolff 2011; Filion 2011). This allows precomputation of the accessibility factors, which are stored in a texture (Wolff 2011; Akenine-Möller, Haines, and Hoffman 2008).

However, as ambient occlusion is dependent on nearby objects, caution must be taken for objects that move relative to each other and when objects are deformed (Wolff 2011). The faults recurring from the simplification of not modelling such occlusion for rigid bodies are not always perceived by the user (Akenine-Möller, Haines, and Hoffman 2008). If this simplification does result in detectable errors, the ambient occlusion from one object on others can be stored in a low resolution cube map or a texture, and then projected onto the scene when the object moves (Landis 2002; Kontkanen and Laine 2005; Malmer et al. 2007). Additionally, the occlusion from one object on itself due to movement of its body parts require precomputation of several poses (Akenine-Möller, Haines, and Hoffman 2008; Kontkanen and Laine 2006). Both solutions for moving objects are very storage expensive (Akenine-Möller, Haines, and Hoffman 2008), which can become a problem when rendering on smartphones.

Screen-space ambient occlusion

Screen-space ambient occlusion is a category of techniques operating in screen-space. Thus, surfaces which are not visible from the camera do not contribute to the calculated accessibility value. Fortunately, this is seldom causing considerable problems for the visual perception (Filion 2011).

Both Filion (2011) and Mittring (2007) have proposed screen-space methods

where rays are traced against the depth buffer instead of several directions in the upper hemisphere. However, their approaches for extracting sample points and performing calculations differ. Filion (2011) has proposed extracting 2D samples in the upper hemisphere of the point by using a normal buffer. Mittring (2007) on the other hand considers 3D points from the whole sphere with the centre in the evaluated point, thus having to approximate which samples are not in the upper hemisphere of the point.

Since the main bottleneck when using screen-space ambient occlusion is the use of the GPU's texture cache system (Filion 2011), Filion's use of the normal buffer in addition to the depth buffer can cause problems. To amend the extensive use of the texture cache system, the depth buffer can be downsampled to a sixteenth of the original size, which often produces pleasing results (Filion 2011).

Both methods have problems with different artefacts, such as banding artefacts or noise, due to the small sample rate and non-randomised sampling (Akenine-Möller, Haines, and Hoffman 2008; Filion 2011). The problem can be amended by using a semi-randomised sample pattern and post-process blur (Akenine-Möller, Haines, and Hoffman 2008; Filion 2011), but those solutions cause higher computational cost and in some cases strain the GPU texture cache system.

An alternative to these techniques has been presented by Luft, Colditz, and Deussen (2006). The method subtracts a blurred version from the original depth buffer, which is inexpensive but only resembles ambient occlusion (Akenine-Möller, Haines, and Hoffman 2008). However, a pleasing image can be produced with correct settings (Akenine-Möller, Haines, and Hoffman 2008).

Volume-based ambient occlusion

Volume-based ambient occlusion is a category of techniques where the accessibility value is estimated analytically by modelling each object as a collection of volume shapes.

Bunnell (2005) used disks positioned at each vertex of the objects. The main issue with this method is that it requires a large amount of discs even for small scenes, which leads to a large memory consumption for storing information about them.

Evans (2006) proposed a method based on distance fields[4] covering the whole scene. The method produces visually pleasing results, but due to the size of the field, it is only suited for small volumetric scenes.

Another option not dependent on the geometric complexity was presented by McGuire (2010). The method includes extruding dynamic bounding volumes from objects, and later traversing them similar to how shadow volumes are traversed during shadowing (McGuire 2010). For many scenes, the result is comparable to ray traced occlusion and can be substantially faster than Mittring's method while less error-prone (McGuire 2010). The main disadvantage of the method is that it uses both normal and depth buffers during calculations (McGuire 2010).

[4] A distance field stores a value for each point on a plane or in a volume. The sign of the value states if the point is inside an object or not, and the magnitude is equal to the closest object boundary (Akenine-Möller, Haines, and Hoffman 2008).

3.2.2 Results

For use in the game, implementations of precomputed ambient occlusion and screen-space ambient occlusion were made. The implementations were initially tested on an Acer TravelMate 5520 computer for an indicator of their speed and accuracy, and later tested on the phone for final comparison. In short, precomputed ambient occlusion was regarded as the best option in the end, and is thus the alternative used in the game.

Precomputed ambient occlusion

The first implementation included precomputed ambient occlusion stored in textures for each object and additional quadrilaterals representing the ambient occlusion on the wall and platforms caused by the object. Due to this approach, the implementation does not model the occlusion caused between moving objects or body parts.

To minimise the use of the GPU's texture cache system, two different simplifications were made. Firstly, the ambient occlusion textures were minimised to a size smaller than the size of the objects shown on screen. Secondly, if the object used a colour texture, the ambient occlusion texture was merged with it before rendering. As a consequence, the occlusion factor was multiplied by both the ambient and diffuse colour, which produced a slightly too dark result.

Another problem noted from the implementation was overdarkening. This occured when two occlusion quadrilaterals overlapped, or when the ghost was positioned in the air or in front of a window. The latter occurence had two reasons. Firstly, the floor quadrilateral was not temporarily removed when there was no platform positioned below the ghost. Secondly, if the ghost ran in front of a window, the window glass was shadowed.

Predictably, the implementation added a sense of depth to the scene. As can be seen in a comparison between Figures 10a and 10b, this resulted in the platforms being easier to distinguish.

Screen-space ambient occlusion

The second implementation tested was the ambient occlusion functionality Bouquet provided as a post-process for jMonkey in 2010.

As a consequence of the alterations of the colour buffer after the scene is rendered, not only the ambient colour term, but also diffuse colours, specular colours, and light sources are darkened. An example of this can be seen in Figure 10c, where the light-emitting fire is shadowed.

Furthermore, the computational complexity of the technique was too high for real-time rendering on the TravelMate laptop. To lower the complexity, Bouquet's implementation was altered to only calculate ambient occlusion in real-time for pixels close to moving objects, and using precomputed ambient occlusion for fixed objects. While the alterations led to a considerable increase of speed from 18 fps to 28 fps on the laptop, it was still too slow for real-time rendering. Additionally, the alterations introduced further over-darkening at pixels at which both ambient occlusion textures

and screen-space ambient occlusion was used. Moreover, when tests were executed on the phone, the GPU's texture cache was overused, resulting in an execution error. Finally, as JME does not support post-processing when compiling for Android, the technique resulted in a black screen even if the calculations were performed.

(a) No ambient occlusion model. Performance: 57.1 fps on the laptop, 52.7 fps on the phone.

(b) Precomputed ambient occlusion. Performance: 54.3 fps on the laptop, 49.5 fps on the phone.

(c) Bouquet's screen-space ambient occlusion. Performance: 18 fps on the laptop, not runnable on the phone.

Figure 10 – Comparison of different ambient occlusion techniques.

3.2.3 Discussion

The precomputed method uses accurate models, and thus yields realistic results for static objects. If it is used, then accessibility values should be precomputed for the occlusion generated by moving objects on other moving objects, and an implementation of correct projection of such occlusion should be made. However, the lack of such support was remotely visible during gameplay.

There are also several proposals of solutions to the problems with over-darkening. The incorrect shadowing by the ghost on the air can be solved by sending the position of the ghost and the occlusion texture to the shaders used to shading the platforms instead of using a quadrilateral. The shaders can then use the distance between the the ghost and the point being shaded to evaluate if the ghost is occluding the platform and to calculate the coordinate for a texture lookup of the occlusion texture. The occlusion of the window glass can be solved in with a similar technique by sending the occlusion texture to the wall and window shaders. The problem with over-darkening where wall occlusion textures from objects overlapped was solved by taking caution to move objects whose wall occlusion may overlap further from each other

Nevertheless, the technique involves a time consuming manual process when computing and storing accessibility values for each object. Screen-space methods have the advantage of not being dependable on the used objects, but demanded more resources than were available on the smartphone. Thus, although widely used for real-time rendering, screen-space ambient occlusion proved too computationally expensive for use on the phone. Other post-processing alternatives that are faster exist, but care must be taken to ensure that they do not overuse the GPU's texture cache system and that they are fast enough.

One solution to the over-darkening of light emitters due to the post-processing is to compute the screen-space ambient occlusion technique during rendering. Then, the accessibility value can be multiplied by only the ambient colour. It also introduces the option for each pixel to choose whether to use the accessibility value from a texture or calculate it in real-time. However, this requires a suitable technique with regard to the hardware resources available on the phone.

4 3D modelling

The models used in computer graphics are built up from polygonal *meshes* and their associated properties. The components of a mesh are introduced in Section 4.1 and illustrated in Figure 11. This chapter then continues to describe some techniques used for creating, colouring, and animating a mesh. This is not intended as a detailed how-to guide, but such guides can be found in the references. Nor is it meant as a complete overview; the topic is huge and the concepts covered here are limited to those that were used in the finalised game.

4.1 Primitives

Model geometry is made up of primitives. The smallest unit is the *vertex* which is a set of attributes describing a point in space. In the simple case this means three coordinates x,y and z in some coordinate system. The notion of vertex may also contain additional information tied to the point, for example colour.

A set of vertices make up an *edge* or a *polygon face*. Edges and faces could in theory also have color attributes associated to them but in practice data is rather inferred from the corner vertices by interpolation.

Polygons in turn make up polygonal meshes which are used as geometric models. Triangles are generally faster to process than other polygons, and for this reason polygon faces are split into triangles for real time rendering.

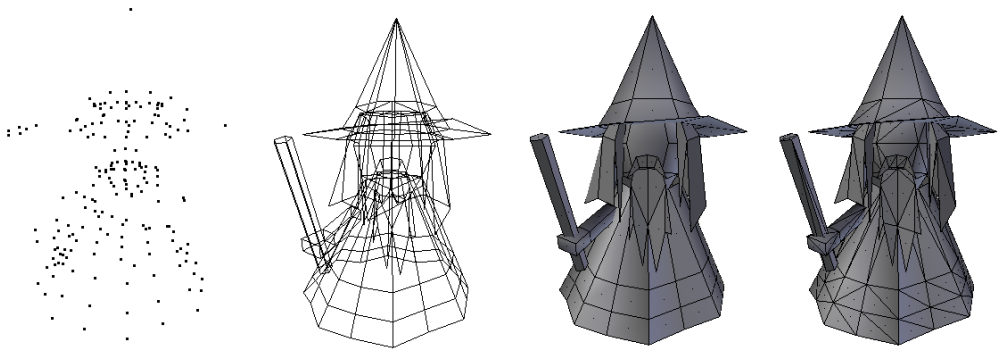


Figure 11 – The primitives making up a wizard mesh. From left to right are vertices, edges, polygonal faces, and triangles.

4.2 Mesh editing

A mesh can be created programmatically (Anders 2010) or using *mesh editing* software. For this project, *Blender* was chosen because it is free to use (The Blender Foundation 2013), and was recommended by the JME community(Vainer 2013c). This section covers basic operations in 3D modelling, cloth simulation, and smooth shading as applied with Blender.

Some of the most basic tools in Blender are *translation, rotation, scaling, extrusion*, and *subdivision* (Flavell 2010). These operations are applied to the vertices, edges, and faces of a mesh. Starting from a set of standard geometries including

cubes, spheres, and cones, many physical shapes can be roughly imitated using only these tools. An example is given in Figure 12.

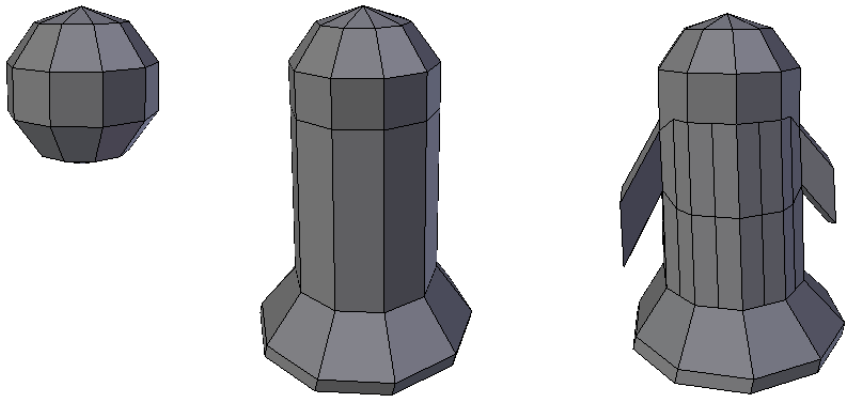


Figure 12 – The ghost mesh is formed from a sphere by translation, scaling, extrusion and subdivision.

Cloth modelling is one application of soft body dynamics, in this case for imitating the look and physical characteristics of cloth (Mullen and Coumans 2008). This is particularly useful when creating a ghost mesh. Blender includes a physics engine that handles the calculations, providing the designer with a flexible way to model cloth without delving into physics (Mullen and Coumans 2008). An illustration is provided in Figure 13. In this case, the soft body simulation is temporary and only used to deform the mesh. When a satisfying shape is found, the mesh is modified to become rigid again.

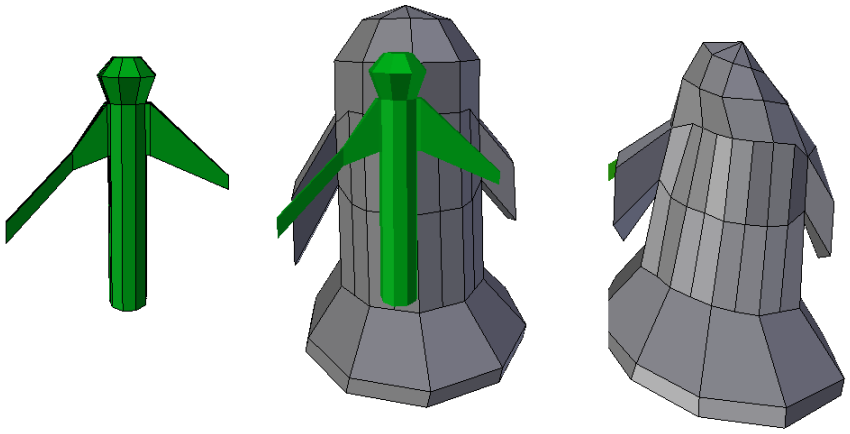


Figure 13 – Cloth modelling. A supporting mesh (green) is placed inside the cloth to control the deformation. A physics simulation is run where the cloth falls down onto the support.

Smooth shading can be enabled for a mesh or parts of it while modelling. This changes the appearance of the surface through shading without increasing the number of triangles. Without smooth shading, a very high triangle resolution would be required to achieve the same smooth look. The effect of activating smooth shading

for the ghost model is shown in Figure 14. The specific shading algorithm that will be used depends on the context in which it is used. JME uses Phong shading as seen in Figure 2.

Figure 14 – Smooth shading applied to the ghost mesh to give the surface a smooth look. Note that the outer edges are still very sharp.

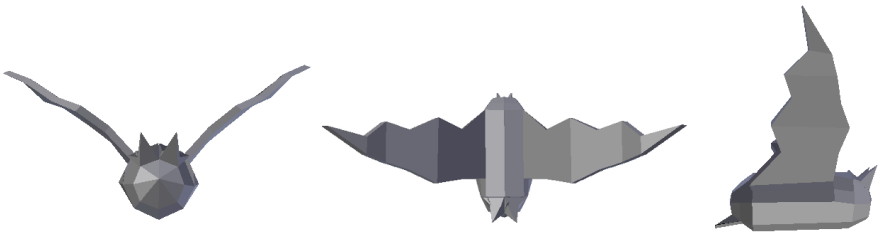
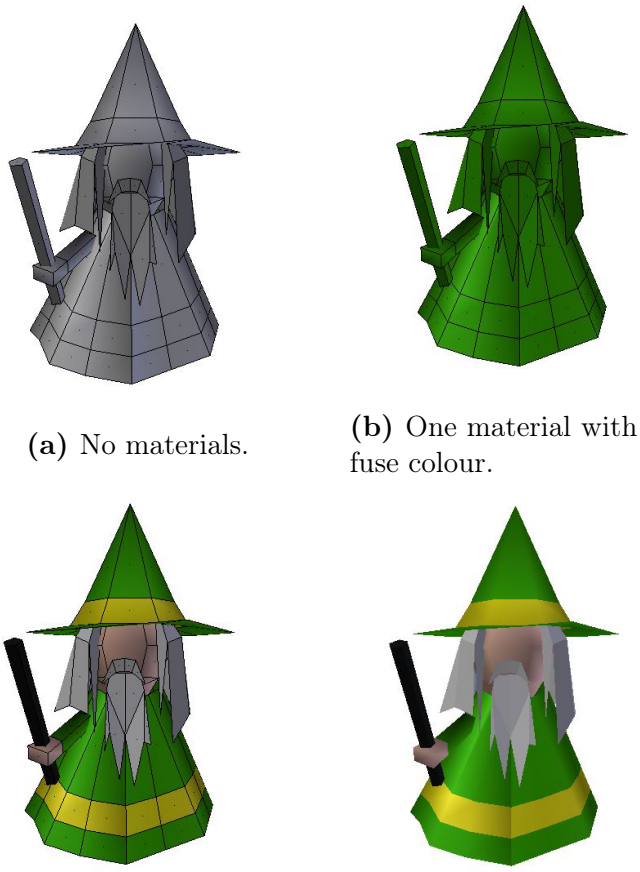


Figure 15 – The bat mesh, flat shaded.

4.3 Colouring and texturing

When the shape of a mesh is set, the next step is to add visual information to the surface. This can be accomplished by the use of materials, textures, and UV maps. A material is a set of parameters added to a model to provide information about surface colour, transparency, textures, and shading (Akenine-Möller, Haines, and Hoffman 2008). This information is used to render the object as intended on screen. JME supports definitions for both *unshaded* and *shaded* materials (Vainer 2013b). A model with unshaded material will be drawn identically regardless of any light sources in the scene, whereas the colour of a shaded material can be affected by several layers of light (see Section 2). An illustration applying coloured materials can be seen in Figure 16d.

Texturing means drawing a 2D image onto the surface of a model. This allows for a high amount of visual detail without the cost that comes with increasing the number of triangles. In the mesh editor, a UV map image is created and an image editor can then be used to draw the texture.



(a) No materials.

(b) One material with diffuse colour.

(c) Five materials with different diffuse colours, assigned to groups of faces.

(d) The same materials as in (c), with the edges between faces hidden.

Figure 16 – Materials and diffuse colour applied to the wizard mesh.

UV mapping is a method in which the faces of a mesh can be mapped to a 2D image. Each vertex in the mesh is assigned a UV coordinate that can be displaced, providing control of how a texture is drawn on a mesh (Flavell 2010). This is commonly used to create skins for game models. UV coordinates are so named because they represent the 2 dimensions of the UV map, as opposed to the 3-dimensional XYZ coordinates of the mesh. One way to visualise the work process is to imagine a mesh made of paper, which is cut open and unfolded on a flat surface, then drawn onto and reassembled.

4.4 Animations

Computer animation is a broad term that can refer to many techniques for creating moving images. This chapter describes two different techniques and a comparison between them.

The first one is animation by transforms. The affine transforms of translation, rotation, and scaling are three basic operations in 3D graphics(Akenine-Möller, Haines, and Hoffman 2008). JME provides methods to apply these transforms to a mesh,

enabling run-time animation by code. Each of the transforms can be applied with an arbitrary factor to any of the 3 axes, creating many possible combinations, however these animations are simple and often repetitive.

The second method is *bone animation*. This is a common technique used to animate game characters and objects, enabling more advanced motions than the ones covered so far. To set up a bone animation, an *armature* (skeleton) of *bones* is made and rigged to connect with the vertices of the mesh. Transformations on the armature will then deform the mesh accordingly (Flavell 2010). Animations are made by putting the armature in different poses and saving these as *key frames*. The game engine calculates interpolated positions in between frames, forming a complete recording of motion. Bone animations can be created in Blender and then imported by JME (Vainer 2013a).

4.5 Results

This section describes the models created during development and some notes on performance impact.

4.5.1 Mesh editing

The game uses 11 meshes in total: three for characters and eight for other objects. The ghost, wizard, and bat were made in Blender with the techniques described above, and the results can be seen in Figure 14, Figure 16d, and Figure 15. The final ghost mesh was made following four iterations of earlier models, experiments with cloth, and group discussions on how to make it anatomically correct. The wizard went through four iterations, and the bat only one. After several re-makes using successively fewer triangles, the final character meshes contain 206, 269 and 288 triangles, respectively.

The first version of the wizard has a hat made with high resolution cloth, counting 10944 triangles. The whole mesh counts 13260. When used in the game, the phone runs out of memory and the game crashes after 20-90 seconds during test runs.

The mesh components for torches and power-ups were made from standard geometric shapes scaled to a suitable size, see Figure 17 for power-ups. The platform, window, and plant meshes mainly contain flat shapes, serving as frames for textures. The non-character meshes are not very interesting from a mesh editing perspective; the models made from them will instead be displayed in later sections.

4.5.2 Colouring and texturing

The ghost, wizard, and bat all use materials with diffuse colouring and smooth shading enabled. This gives them their base colour, and for the bat nothing else is added on top. End results as seen in-game are illustrated in Figure 8, Figure 20, and Figure 23.

The game includes eleven textures used with models, all of which were hand-drawn using various image editors. There are two textures each for windows and plants. Each power-up has a standard volume mesh with a texture on it, shown in Figure 17.

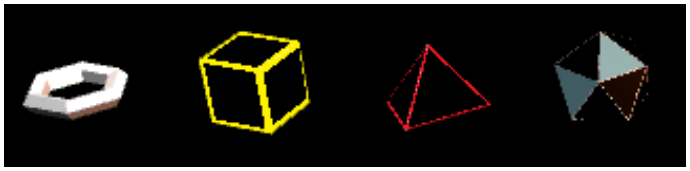


Figure 17 – The powerup meshes with UV mapped textures.

The background wall and platforms use a texture made to be repetitive, as seen in Figure 4. It creates the illusion of a large seamless image, although the brick texture used is only 52x58 pixels. The plant texture, despite its irregular shape, is displayed nicely on a rectangular mesh thanks to its use of transparency: Figure 18. An illustration of the in-game result is shown in Figure 19.

Large textures caused the game to crash because the GPU ran out of texture cache. As with models, early textures had an unnecessarily high level of detail for their purpose. Later iterations were scaled down to save memory, and this could be done without losing visual quality since the screen of the phone is so small.

UV mapping is implemented for the wizard and ghost textures for facial features, as seen in Figure 20. It was also used to achieve the wrap-around look on the platforms, as seen in Figure 21.

4.5.3 Animations

There are many methods of animation, and two have been successfully implemented in The Jumping Dead. Wizards are animated by rotation, turning them to face towards the ghost. Power-up graphics were designed with animations by rotation and scaling. The design concept of all power-up animations is displayed in Figure 22.

Bone animations were implemented for the ghost, the bat, and the plant, illustrated with Figure 24 and Figure 23. Some are played in a loop when no other animation takes place: the ghost walks by default and the bat flaps its wings. The ghost jump animation is activated when initiating a jump by the player pressing a key. Collisions trigger the plant animation and the ghost flinching. Finally, the invulnerability animation for the ghost is activated on the state change that occurs when touching the power-up.

As implemented in the game, animations have a visual function and an informative function, but no logical consequences. Each animation is the result of a state, never the cause. Disabling all animations would make the game less interesting, though collisions, jumping, and other interactions would still occur.

Two test runs were made to estimate the impact on performance. The highest number of animated models that occurs naturally in the game is a scene with the ghost and three bats on screen. This scene was tested with and without animations, and no significant difference in frame rate could be noted.

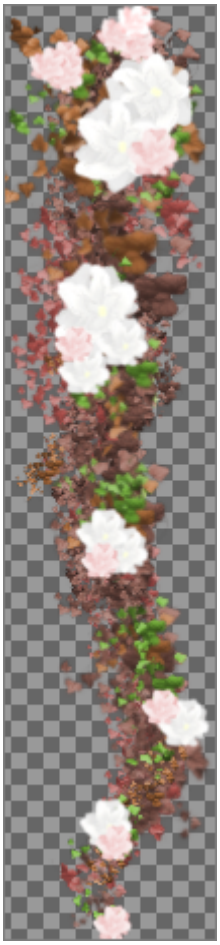


Figure 18 – The plant textures make use of transparency to display on a rectangular mesh, where the unused pixels are seen through. The checkboard pattern represents transparent pixels. There are two different plant textures with the same function.

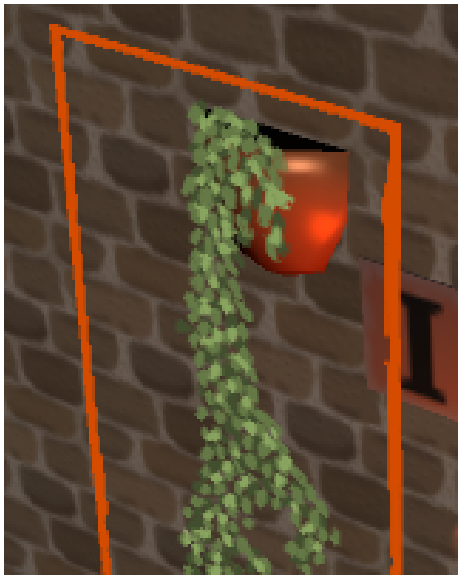


Figure 19 – The plant model close up to show the plane's position relative to the scene. The texture's edge is coloured red to show it more clearly.

4.6 Discussion

Mesh editing

The final wizard model uses roughly 1/50 times the number of triangles compared to the first. However, it makes better use of those triangles, with hair, a beard and coloured clothes. These design details are computationally very cheap and the final wizard ends up looking better in game. One problem with the 13620 triangles is that the potential detail level was not noticeable the way it was used in the game.

Attempts were made to create a mesh for a 3D plant, but that proved to be very difficult with the tools known at that point. It can be done with a mesh editor but would require other tools, and likely a high triangle count to look good.

Colouring and texturing

Manual skinning and UV mapping the whole mesh was tried early in the project, however it was disregarded because drawing textures require a lot of time, and continually changing the meshes means redrawing those textures. Later, when the final meshes were settled upon, skins could be made with confidence that they would be kept. The easier solution was to add colour through materials which is both fast and insensitive to mesh changes. Materials are assigned to faces and if the shape of those faces change, the colour stays the same. In the case of remaking the whole model, it is just a matter of keeping a few RGB-colour values and re-applying them which can be done in minutes. UV mapped textures were thus mainly used for flat objects, that is walls, platforms, plants and windows, and for power-ups which have regular shapes not subject to change.

Another advantage of material colours is the ability to change colours at run-time by assigning a new RGB-value to the diffuse method of the material. This was used to re-dress the rapid-firing wizards in black and red instead of green and

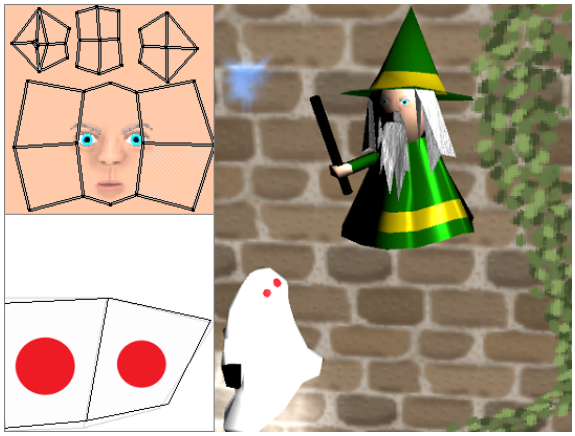


Figure 20 – UV mapped character skins. The polygon shapes correspond to faces of the mesh. The polygons are displayed to illustrate the concept, and are not part of the actual textures drawn on the mesh.

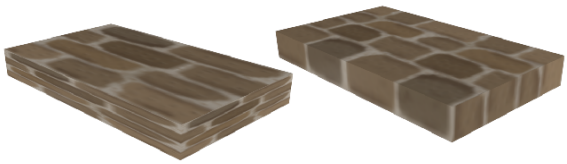


Figure 21 – Left: Default texture mapping on a platform meant that the texture was scaled and drawn onto each face. Right: This was corrected with manual UV coordinates for a better look.

blue, to distinguish them as more dangerous. Switching textures at run-time is of course also possible, but requires drawing, importing, and keeping those textures in memory.

Textures tend to look flat when drawn on low-polygon models and flat surfaces since the shape of the surface is unaffected. There are many techniques used to improve texture looks and remedy this problem, collectively named *bump mapping*(Akenine-Möller, Haines, and Hoffman 2008). Bump mapping would have been a nice addition to improve the look of walls, platforms and plants, but this requires using shaded textures. To improve performance, shading was disabled for the wall, platforms, and plants, and thus it was not an option to use bump mapping on most textures in the game.

Colours were mainly specified in Blender as part of the model file and then exported for use with JME. The recommendation from the JME community is to rather do such configuration inside JME because different shaders are used, and appearance in Blender during design may be different from the final appearance in the game. This was never an issue in this project, and it was far easier to configure these in Blender. However, for a project with greater needs for detail and perfection, and using more types of shading, the recommendation should be followed.

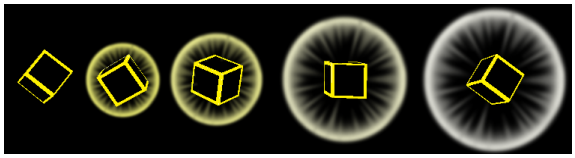


Figure 22 – Animation of the invulnerability power-up. A cube model is rotated on the three axes and a single particle image is scaled by a periodic function of time. The animations for the other three power-ups are variations using the same techniques.

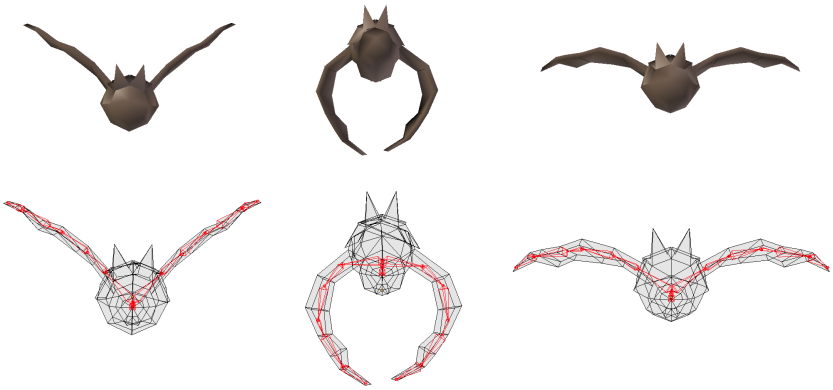


Figure 23 – Bone animation of the bat. The armature, placed inside the mesh and displayed in red, is seen in the key frames of the bat flight animation.

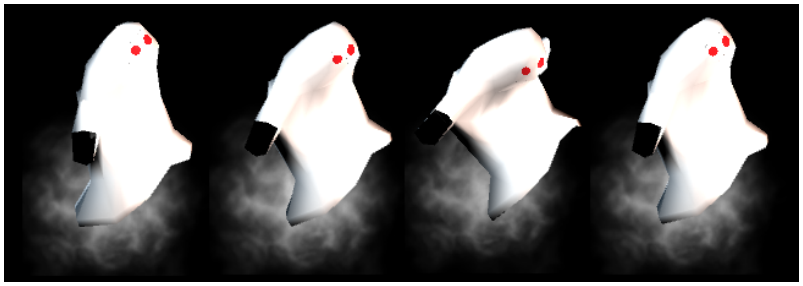
Animations

There is some overlap between the above definitions of animation and movement. The movements of game characters are in fact translations but these are such basic game elements that we do not think of them as animations. However, when comparing early wizards that lack movement to the ones in the finalised game, the latter feel much more alive thanks to translation and rotation movements. Another overlap is that the ghost has two bone animations that could have been made with transforms instead: a Z-rotation and a Y-scaling. Coding them would have been faster than recording with key frames, but having animations for one character accessed by different methods would be impractical.

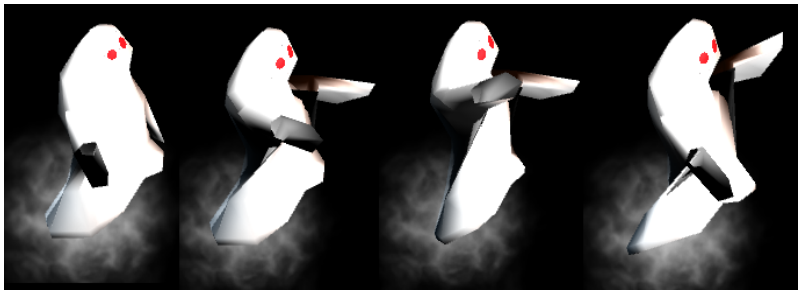
Character models

When modelling, each of the main characters provided different challenges, based on their role in the game seen from a design perspective. The concept used for the ghost does not require much colouring or texturing, but being the main character, the ghost interacts with most other objects and needs ways to visualise these interactions and states. This is reflected in the many animation sequences available to the ghost and the randomised choice of three different collisions. Its different power-up states is displayed by the color of the particle effect attached to the ghost.

The wizard has a focus on visual variety and detail. He wears a five colour skin and a particle effect for a total of six associated colours. He has detailed face and



(a) Flinching animation, played when the ghost is hit by an enemy.



(b) Frenzy animation, played when the ghost is invulnerable.

Figure 24 – Four images each of two different bone animations made for the ghost. Note: These are screenshots from gameplay, not to be confused with the key frames of the animation.

beard textures. His spells have about 60 possible colour combinations, and he can change the colour of his clothes and particle by sending one or two RGB values to his redress method.

The challenge with the bat was to make it look realistic. Although the whole game has a cartoony style, this animal corresponds to something real and should make an effort to imitate that. No one complains if a ghost moves a bit weirdly, but wings flapping is something else. Thus, the main challenge with the bat was creating the mesh and the animation. Photos of real bats were used as concept art and much attention was paid to individual vertices and scale of body parts, which were not as important for the other characters. The skeleton and animation were inspired by videos of real bats and birds flying. The bat was, more than any other model, restricted by the hunt for low triangle count, since the body and wings could have been made to look much better if given more vertices.

5 Particle systems

A particle system is an image-based animation tool. It consists of a group of *particles*, visible objects that move and change according to a set of rules (Akenine-Möller, Haines, and Hoffman 2008). They are a good complement to 3D models, often used to represent things that are non-solid, transparent, constantly change shape, or several of the above (Akenine-Möller, Haines, and Hoffman 2008). A few examples are fire, waterfalls, smoke, and explosions.

This section briefly describes how particle systems in computer graphics work, and moves on to detail their implementation in the game.

5.1 Basic concepts regarding particles

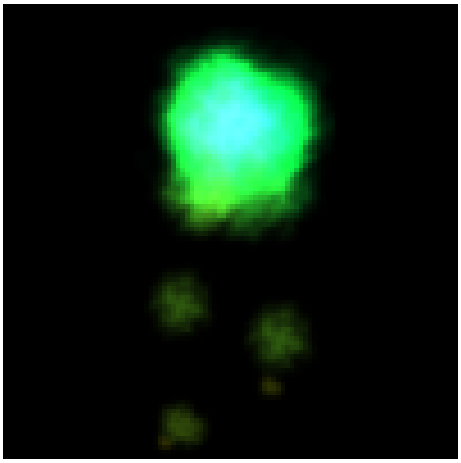
Each particle is represented by a 2D *sprite*, an image or collection of images ranging from a single pixel to a sequence of animated billboards (Akenine-Möller, Haines, and Hoffman 2008). A simple particle system consists of a *generator*, a point in space where the particles are created. The particles usually move away from the generator, sometimes randomly, and fade away with time. The sprite may be animated and change between several images. By varying parameters such as the movement pattern, speed, colour, size, and lifetime of the particles, many different effects are achieved. Some examples of particle systems created using JME are seen in Figure 25.

5.2 Results

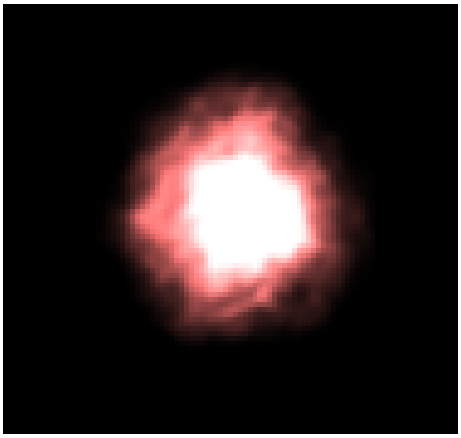
JME has built-in support for particle systems, which resulted in that the implementations of the objects that constisted of particle effects were easy. A particle emitter object can be created and attached to the scene graph, and with no further configuration be visible in the scene. Numerous details can then be configured to get the desired appearance, from colour, size, and the billboard image to the particles' movements and animation settings.

Several particle designs have been implemented, as seen in Figure 26. The wizards' fireballs were the first, followed by the magic spark on their wands. The ghost leaves a semi-transparent trail of clouds, and an intense effect surrounds the power-ups. Torches come in different shapes and colours and most of them are based on one particle emitter with some random variables added for visual variety. The different things modelled with particles may seem unrelated from a user perspective, however, on the inside they are to a large extent identical, with mainly size and colour variations.

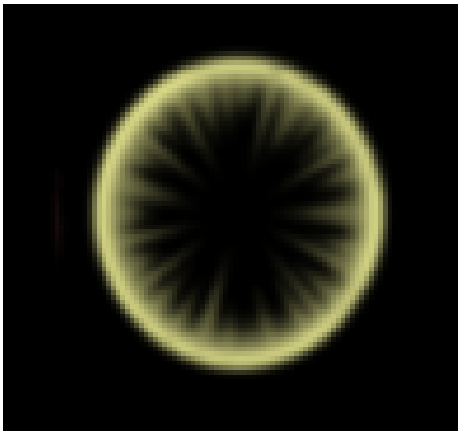
The particle systems used have not made a noticeable impact on the game's performance. The illustration in Figure 26 may be considered to display a high amount of particles on screen for the game, and more extensive testing has not been done.



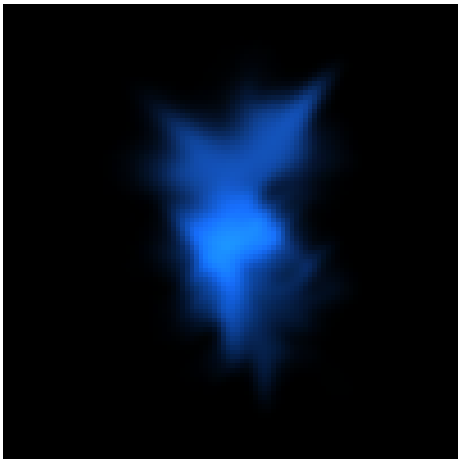
(a) Dripping goo, using a decreasing particle size and high gravity.



(b) Explosion-like glow, using big particles, no gravity and small initial velocity.



(c) Force field, using a texture on a single particle growing in size.



(d) Sparks, using few particles with big, spark-like textures.

Figure 25 – Examples of particle systems.

5.3 Discussion

The game benefited greatly from the inclusion of particle systems. This is evident from how basic some of the objects using particles for their appearance are – decorative torches and wizard spells in particular – and how hard it would be to replace them with 3D models.

Compared to rendering bigger models, higher-resolution textures, or performing an extra rendering pass to render shadows , the computational resources claimed by the particles in the game are minor. Since the same texture is used for every particle of the same type, the GPU cache will not be strained. Since the particles cover a small area of the screen and do not require for example lighting computations, they will not push the GPU itself to its limits either.

We have not deliberately tried to increase the number of particles to reach the limits, because for this game the number of particles currently used is enough. More particles would not have added much the the visual appeal, as a church can only

Figure 26 – In this image from the game, particle systems are used for the ghost, the power-up, the torches, the wizard's wand, and the spell.

hold so many torches and sparkling wizards before it becomes too much.

6 Game logic

The purpose of this project is to create a game, and this section will detail the specifics of that game. Although the focus of this thesis is the graphics of The Jumping Dead, which is only tangentially related to its mechanics, the design of those mechanics were in the end a substantial part of the project.

The results in this section are presented at face value, and without much discussion. This is because the design decisions concerning the game logic are generally not supported by anything other than the group's own judgment. The purpose of this section is instead to give the reader a working understanding of the game, in order to follow in the examples given in other sections.

6.1 Main gameplay

The main gameplay of The Jumping Dead is running and jumping. The ghost's movement is mostly handled through the physics engine *JBullet* (jbullet.advel.cz 2008), which is integrated in JME. The engine makes sure that the ghost is pulled down by gravity and does not fall through platforms. It runs in discrete time steps, called *ticks* (jMonkeyEngine 2012). However, it automatically updates the ghost's position between each tick based on its stored velocity value. The following algorithm, run before each tick, determines what that velocity will be:

1. Check if the player model is on the ground. If it is, set the forward velocity to the value given by the current running speed.

2. If the jump button has been pressed since the last tick, set the upwards velocity to the standard jump speed.

3. If the jump button has been released since the last tick, cut the jump short: if the upwards velocity is bigger than the *cutoff value* (61% of the initial jump speed), set it to the cutoff value.

4. If the player was hit by an enemy since the last tick, overwrite all previous velocity with an upwards speed the same as the standard jump speed, and a backwards speed equal to 71% of the running speed.

5. Pass the new velocity to the physics engine, which applies gravity and forces from collisions, and then updates the player's position.

In the above, the current running speed is based on the current difficulty level – the speed increases by 3% of the starting speed every three seconds.

6.1.1 Jumping

The jump routine is the same as the one used in the *Sonic the Hedgehog* game series (Mercury [pseud.] 2013), illustrated in Figure 27. When the screen is tapped, the vertical speed of the ghost is set to a predefined jump speed. If the tap is held, the ghost continues along the upprmost trajectory until gravity brings it down again. However, if the tap ends the vertical speed is immediately set to a predefined lower

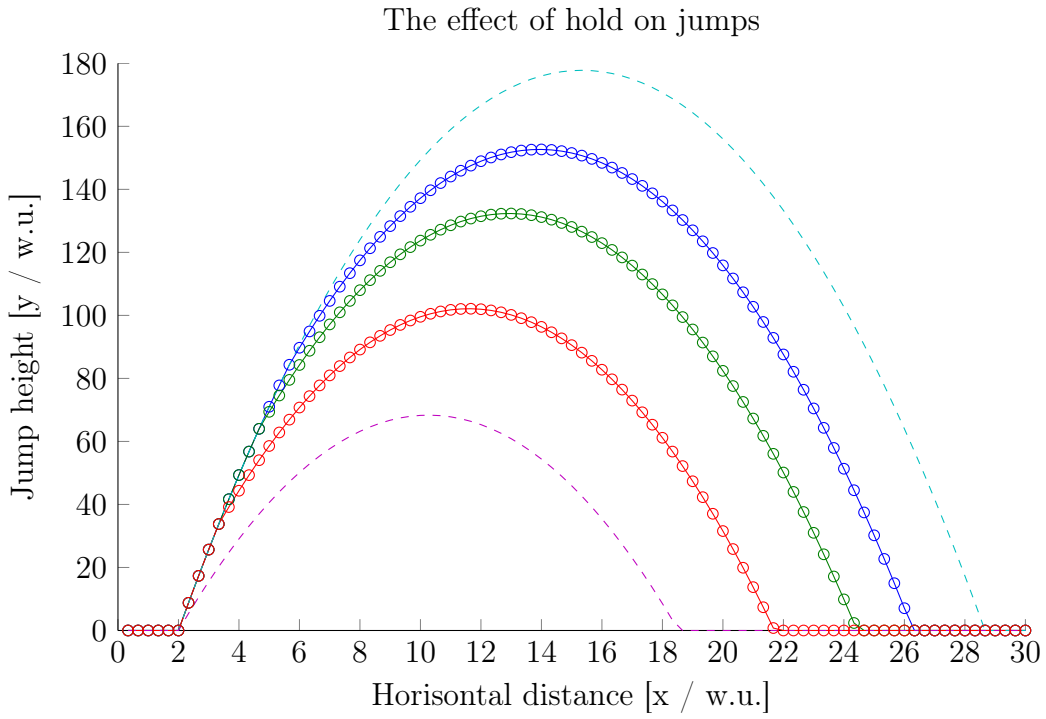


Figure 27 – Plot showing the *Sonic* jump algorithm.

value, unless the vertical speed has already fallen below this treshold. This game uses a lower speed of 61% of the initial jump speed. This algorithm allows for different jump heights.

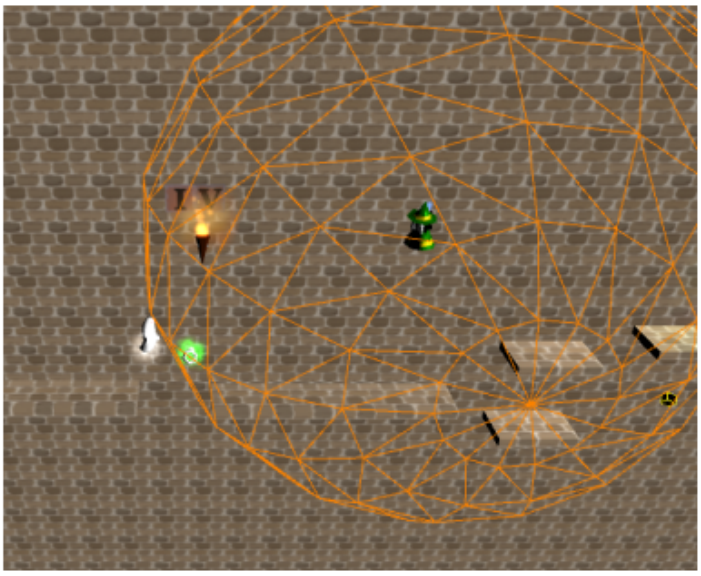
The ghost's speed can change, and this also changes the velocity and acceleration values used for jumping and falling. The reason for this is to keep the jumping and falling trajectories the same shape regardless of speed. Specifically, the velocities given when jumping and being pushed back by enemies increase along with the ghost's speed, and the acceleration due to gravity grows as the square of the ghost's speed. The speed increases along with the difficulty over the course of the game, and can also be affected by power-ups.

6.2 Collision detection

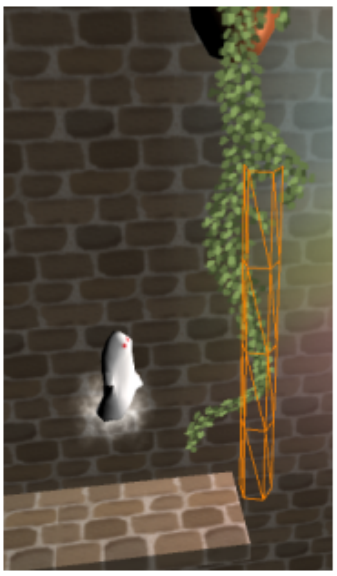
In addition to handling the interaction between ghost and platforms, the physics engine is used for collision detection. Any object may be registered as a *collision listener* to receive collision data from JBullet. In the game, one collision listener receives all collision data and alerts the colliding objects to react accordingly, for example when the ghost collides with a bat, or a magic bolt with a platform.

JBullet also supports non-physical objects, called *ghost objects*[5]. They can pass through other objects freely, but doing so will still alert the collision listeners. This is utilised for the enemies in the game, most readily apparent for the wizard. Each wizard has its own ghost object in the shape of a sphere, shown in Figure 28. As long as the player is inside the sphere, the wizard will turn towards it and attack it

[5]Not to be confused with the main character of the game, only known as 'the ghost'.



(a) The aggression range of a wizard.



(b) The activation area of the plant.

Figure 28 – Two collision shapes, ordinarily invisible, shown in orange. Left: the aggression range of a wizard. Right: the activation area of the plant.

with regular intervals.

All other objects that interact with the player, such as bats, power-ups and plants, also use ghost objects. These are generally in the shapes of spheres or cylinders, which means they crudely approximate the shapes of the objects themselves. When the ghost object touches the player, the appropriate reaction occurs regardless of the speed or direction of the collision. These reactions include the player being pushed backwards from a bat, or a plant swinging to the side when the player passes by.

6.3 Power-ups

The Jumping Dead features four different types of power-ups that the player can pick up. These are *Invulnerability*, *Double jump*, *Speed up* and *Speed down*, and their names accurately describe their effects when picked up. They all use spherical ghost objects to determine when the player has touched them.

The *Invulnerability* power-up makes the ghost invulnerable for five seconds. In that time, attacks against the ghost have no effect. The bats and magic bolts react like they do on a normal hit, but the ghost is not knocked back.

The *Double jump* power-up gives the ghost the ability to jump again in mid-air. This is done by tapping the screen a second time. The ghost can jump once while airborne, and must touch the ground before being able to do it again. This ability does not go away with time, instead it stays until the ghost is hit by an enemy.

From their names, the *Speed up* and *Speed down* power-ups sound like each others' opposites. However, their implementation is quite different. *Speed up* gives an increase in speed, equal to 33% of the speed at the start of the game, which can

bring the player over the ordinary maximum. It is, however, temporary and the speed will decrease by the same amount after five seconds or when hit by an enemy. Slowing down on the other hand is a one-time event: the speed is set back by 33% of the starting speed, and the normal acceleration starts over from the new speed. In addition, picking up a *Speed down* up ends any *Speed up* currently in effect.

When a powerup is in effect, the ghost's appearance changes slightly depending on the specific powerup. The trail of mist the ghost leaves behind either becomes green, bigger, or lingers for a longer time before dissipating. When *Invulnerability* is taken, the entire screen becomes brighter and the ghost switches from its ordinary walking animation to an alternative one (see Section 4.4).

6.4 Level generation

The level is generated in sections, with each section consisting of a piece of wall with platforms attached. On each section is also a variable number of enemies, power-ups and decorations. All these features are randomised independently from each other.

The level generator places platforms based on a platform layout chosen randomly from several available layouts. The position of the last platform in a section determines that of the first platform in the next. This ensures that the distance between two platform is never so big that the ghost cannot clear it. The level generator also checks the height of the last section against upper and lower bounds, so that the upper and lower edges of the wall cannot be seen.

Some platform layouts determine which enemies show up as well, but for the rest the enemies are randomised separately. The possible enemies on a section are one wizard, one bat or three bats flying in formation. If the section has a wizard, the type of wizard is also randomly chosen (see Section 6.5).

The decorations available are windows, torches, plants and milestones. Each section holds either a window or a torch, but not both. 60% of all sections also have a plant. Which variety of window, torch or plant is used is also randomised. Finally, every third section has a milestone showing the player's progress in roman numerals.

There are a number of parameters used by the level generator, such as the default distance between platforms, frequency of certain types of enemies and power-ups, and heights of the various decorations. Their values have not been determined in a systematic manner, but rather by continually tweaking them during development; keeping what worked and changing what did not.

The game world does not exist very far away from the player. As seen in Figure 29, only about five times the visible level is kept in the program's memory at all.

The part of the level left behind cannot immediately be removed since the player can be knocked backwards repeatedly so. Moving too far ahead is not a problem, since the distance between the player and the right edge of the level is what triggers the generation of more content.

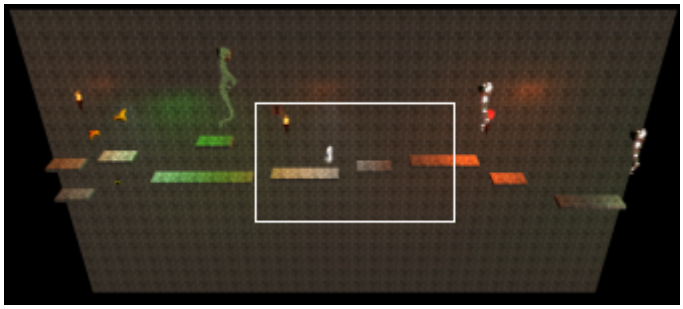


Figure 29 – The entirety of the level existing at a given time, as seen from a greater distance. The white rectangle represents the area seen by the player.

6.5 Hazards

The three types of hazards in the game are bats, wizards and the wizard's magic bolts. The wizards in turn come in three variants, differing in placement and attack patterns. As previously stated they all use ghost objects to determine when to interact with the ghost.

All hazards move during the game, and all such movement is accomplished in the same way. Every time the screen is updated, the model is moved a small distance, proportional to the time elapsed since the last update. This is the same way that JBullet performs the ghost's movement, but it is handled separately and not necessarily at the same rate. This movement is completely separate from the animations described in Section 4.4.

Of the three types of wizard, one is set apart from the other two. The two ordinary wizards fly above the ghost and fire magic bolts straight at it. In contrast, the third type of wizard flies in the foreground and fires at a point just ahead of the ghost, calculated as follows. The ghost's current velocity is interpreted as a distance vector, giving the position (relative to the ghost) where the ghost will be in one second. The height component of this vector is scaled by a factor of 0.3, and the resulting position is the target for a fireball whose speed is chosen so that it will hit in one second. If the ghost then keeps on moving forward, it will often be hit by the fireball. The reason for scaling down the height component of the new position is that in the middle of a jump the ghost temporarily has a very large upwards or downwards component of its velocity. Using that vector without change results in the foreground wizards firing magic bolts far above any point the ghost will ever reach.

7 Using a game engine

At an early stage, it was decided that the game was to be written using the game engine jMonkeyEngine (JME). It is a free, open-source 3D game engine for programming written in Java, which was the group's favoured language. JME also provides a number of features that were deemed useful, including basic implementations of textures, lights, shadows, and particle effects; a physics engine; and support for building projects for Android.

7.1 Choosing a game engine

Using a game engine of this type heavily influences the level of the programming. Building a small, simple game, one could feasibly have started on a lower level, directly using the OpenGL libraries needed for Android graphics. Another way lies with higher-level engines like Unity, where even more of the work concerns designing the scene and objects, and less concerns classical programming. For this project, the 'middle way' was chosen.

The decision to use JME in particular came from it being the first candidate found that fulfils all of several wishes from the group. Firstly, it uses Java, and at about the desired level of programming. Secondly, it provides implementations of the sought graphical effects that are extendable (even more so thanks to the open source code) yet would be 'good enough' if there turns out to be no time to improve on them. Thirdly, it is free to use.

7.2 Adapting to jMonkeyEngine's framework

Any choice of engine or framework limits things like code structure and workflow to some degree. An engine that loads assets and draws objects to a screen must have those assets and objects organised in some way. The particulars of this organisation can certainly influence the developers' choices for how to represent the internal workings of their game. This was definitely the case when developing The Jumping Dead.

When building a game, there are in principle two different systems that need to interface with each other in some way – the model (the game logic) and the view (the visual representation). There are examples of things on both sides that do not concern the other side – purely decorative background objects not represented in the game logic, or an invisible line that when crossed triggers an enemy attack.

The model can, in principle, be handled in any way imaginable, as long as a link is kept that updates what is shown on the screen. Depending on the specific game, the internal representation of game objects might be very simple. In a platform game with 2D logic all information needed about a static rectangular platform can be stored in four floating-point numbers. Designing this model is an important step in creating the game code. The programming principle called Separation of concerns (Hürsch and Lopes 1995) suggests that the model and the graphical representation should not be made too dependent on each other.

7.2.1 Spatials and controls

Working in JME, however, at first stymied the group when it came to model and view separation. Built around the scene graph, all objects that are drawn on the screen are so-called *spatials*. Their appearance, including position and rotation, are stored internally and can be manipulated from the running code. To each spatial can be attached a *control* object, which contains code that affects the spatial each time the game updates.

This framework means all object behaviour – the logical representation – end up in control classes, tightly connected to the spatials – the graphical representation. The classes JME uses to represent things like vectors and geometric transforms, as well as the routines used to activate and deactivate sections of code, are all part of this package deal. During the process, this sometimes felt like a constraint to work around. Creating other data structures to store certain information and then porting it into the JME spatials-and-controls system were actually considered. Not because the system in place is downright bad, but because it at times seems awkward and unfamiliar. However, with more experience using the engine, or with more active guidance from programmers with such experience, it might have proved a powerful tool.

7.2.2 Loading assets and scenes

One of the appeals with using JME during this project was the ease with which assets, e.g. models, textures, and sounds could be imported into the game. This can be done at any time during the program, anywhere in the code. However, only after working with The Jumping Dead for over half the project did the group realise the potential of the *scene composer* tool the JME development kit offers. It allows for designers to load multiple models and carefully design a scene, and then save the entire scene to be loaded at once when the game is run. This also allows for batching models together to optimise both rendering and physics behaviour.

The downside to this realisation was that one of the earliest design choices for The Jumping Dead had been to use a procedurally generated level. Only a few objecs are generated at a time, and their exact placement is randomised which prevents designing the scene beforehand. Unfortunately, the project had run so far that it was deemed too late to change this basic design, and so the JME scene composer was never used.

7.2.3 Conclusions

The way this project turned out, it cannot be said what would have gone differently using another engine. Even less so if something lower-level than a complete game engine, like Libgdx, had been used. It is certain, however, that many things would have gone differently. A lot of work would have been saved on trying to work around a system that was, in a way, built for another type of game than The Jumping Dead. On the other hand, such a basic thing as loading a Blender-created model and its texture into the game, which is done with a few lines of code in JME, could have been a big undertaking in itself using other tools.

7.3 Issues on Android

When developing in JME, the game runs on the computer by default and requires a special setting to export to the Android device for testing. More than once, features which ran perfectly fine on the computer caused glitches or even crashes on the smartphone. This section outlines a few example situations that had to be solved.

7.3.1 Physics

The physics in the game partially uses a physics engine called JBullet, which is integrated with JME. It consists partly of *native code*, that is code written and compiled in another language and integrated into the Java classes. The standard native libraries used for most of JME's deployment options does not run on Android devices, so there is alternate native code specifically made for Android. However, there turned out to be some differences between this code and the ordinary JBullet code used when testing on the computer. This discrepancy caused many errors that were difficult to debug.

For example, the game uses *ghost objects*, which are non-physical volumes that report when an object passes through them which in this game is used to detect when the player is in close range of a wizard. The implementation used for the player's collision shape worked as intended on the computer, but on the phone it could not pass through the ghost objects. The collision that was supposed to raise an event but let the player through instead behaved as if it were solid. This problem was eventually solved by writing a new PlayerControl class, building on a more general rigid body behaviour.

A similar issue appeared with the code concerning checks of whether the player is on the ground of a platform. This is done by means of a *ray test*, that is checking if anything in the path of a given ray. A ray is drawn from the character to a point just below their feet, and if it intersects anything the program concludes that the player is on the ground. The length of this ray should be as small as possible, so that the player will not be classified as on the ground when actually in the air just above it. However, the value which other JME classes used that worked fine on the computer, turned out to be too small on the Android device. The player model could rest on the ground but claim that it was airborne, and thus would not start walking forward. It seems that the Android version of the physics engine works with a lower resolution, so that the length of the ray was lost in rounding errors.

7.3.2 Graphics

In JME there are built-in implementations of a number of concepts of game graphics. This includes material properties, lighting, a scene graph, particle effects and post-processing. Unfortunately there are some problems with support for Android in the graphical implementations as well. According to jMonkeyEngine (2013a) some effects are simply not supported and others drastically lowered the frame rate on the phone.

Many of these had to do with the graphics cache on the phone. The error 'out of graphics memory' appeared many times for different reasons. Advice from the

JME community along with more detailed testing made clear that loading any one texture too large in size gave this error. This occurred for example at times when too complex models, with high-resolution textures, were used. It also effectively prevented the inclusion of any kind of post-processing, as such effects require storing and manipulating the entire screen image. This includes the screen-space ambient occlusion, detailed in Section 3.2.2, as well as bloom filters and motion blur effects [6].

[6]Both bloom filters and motion blur was brought up as candidates for inclusion but were never a high priority, and no substantial efforts were devoted to them.

8 Results

The most tangible result of this project is the game, which utilises a 3D environment with dynamic lighting, and functioning game logic as described in the purpose. The game contains several textures and animated 3D models created by the group members. On the contrary, the movement and collision detection of the models are controlled by a physics engine which was not part of the development of this game.

However, the game is stalled when new sections of the level are generated. This results in the game not running smoothly, although the last measurements, when all final effects were activated, gave an average frame rate of about 50 fps (see Sections 2.3 and 3.2.2).

In addition to the game, the project also resulted in many insights in computer graphics. The described techniques have been used in the computer graphics field previously, but this project has evaluated their suitability for smartphone games. In some cases, the measurements indicated what was expected from the literature studies, while other results contradicted them. Some of the results presented here have resulted in recommendations, given in Section 10, for other developers in similar situations.

Using large light sources to model the light from torches, windows, and wizards was too computationally expensive to be used in the game. Instead, the final version of the came uses precalculated light textures to model the illumination of the wall. Furthermore, the main visual appeal of lighting regarding other game objects comes from light sources smaller than the ones originally included. The light from the windows was also associated with colour textures in order to model the colourisation of light being transmitted through the church windows.

While lighting greatly improved how realistic the game looks, shadows are needed to give a sense of depth. Both drop shadows and ambient occlusion were investigated, and both greatly improved how realistic the game environment looked. However, the implementation of drop shadows did not transfer to the phone, which is why it is not included in the final version of the game. The ambient occlusion included in the game is prerendered, a method not unlike the embedded lights mentioned above.

Several animated 3D models are used in the game. A main character with six animations has been made, as well as two enemies, two decoration models, and four interactive power-ups. In addition to the main character, one enemy and one decoration uses skeletal animation. Some initial models used too many triangles, which caused performance problems, especially regarding GPU memory, and were replaced by low polygon models to amend these problems. These models were enhanced by UV mapped textures, which were scaled down to a size smaller than that of the corresponding objects shown on screen in order to relief the strain of the GPU's texture cache system.

Another technique used to either model objects or improve existing objects are particle systems, which had a negligible impact on performance. Various uses of these included modelling of fire and magic, conveying of status information, and intensifying effects. In order to make the scene more interesting, random parameter variations were used by the systems.

9 Discussion

Overall, the finished product matches its expectations. However, there were issues that were not foreseen, which have increased the awareness of the limitations of the device itself, the game engine used, and to some extent the working process that was adopted. Looking back, with the knowledge that have now been acquired, changes could have been made to make the development process smoother. There are also game features and content, though not critical for the playability of the game, which, given more time, could be implemented.

9.1 Project management

From a management perspective, a more efficient way to structure the work would probably have been to better use the individual strengths of the development team. As it were, most members of the team could work on the areas that interested them the most, which in itself can be seen as a good incentive to produce quality work. However, this resulted in a substantial amount of time being spent learning new subjects, as the areas of interest did not always correlate to previous expertise. Whether this was the optimal approach is still up for debate. Furthermore, a better distribution of tasks within the team could probably have benefited the development.

However, one thing which is agreed upon is that this project could have benefited from better planning and structure. For instance, producing concept art at a much earlier stage would have helped steer the development in the right direction sooner, leaving more time to work on implementation. A similar observation can be made about how the effectiveness of the development could have been improved by better planning the structure of the code. This could have been done by more extensive use of Unified Modelling Language (UML) diagrams.

9.2 Game development

All that is considered core gameplay features have been implemented. However, there is a list of content that could be implemented given more development time. Items on this list include new features such as high scores, achievements, and more objects in the level. Furthermore, a solution to the problem with the game being stalled when new sections of the level are generated has not been found, which means that the game cannot be released in its current state. An attempt to solve this issue has been made, which involved separating the level generation into a separate thread. This improved the performance, but the game was still occasionally stalled.

A perhaps larger undertaking, though one which is considered to be important, especially if the game is to be released to a larger audience, is to optimise the game to run on more devices than the one used for this project. This would include solving problems, such as adapting assets so that they can be used in many different resolutions. Furthermore, adding the option for the user to adapt the gameplay from a settings menu, changing features, such as sound levels, overall difficulty of the game, and control scheme would be important if the game is to appeal to a larger audience. It is also believed that a user study could have benefited the project, especially in order to be able to release the game to a larger audience. Although

the subjective opinions of the group members can be used as an indicator for what is the best option in a decision regarding the game concept or design, the group members cannot represent the whole possible user group for the game.

Developing the game with a single smartphone model as the target device had both advantages and disadvantages. Using a target device greatly limited the number of different hardware setups and Android version which needed to be considered. Additionally, testing was made more consistent since the same mobile device was always used. However, as mentioned previously it causes issues when the game is run on other models that have different hardware. A solution to this problem could have been to have the game recognise the hardware of the device it is installed upon and then adapting certain settings, which could have made it possible to run the game on a more wide variety of devices.

A problem, which occurred at later stages when more extensive game testing was conducted, was that no effort had been made to facilitate testing of the game. Features that could have been implemented to help this include pressing a button to spawn a certain type of enemy, turn on a certain power-up, or pause the entire game. The lack of this functionality, combined with the fact that the level was randomly generated, made it unnecessarily hard to reproduce bugs when needed. Maybe even implementing a level editor, implying a way to save and load specific levels, would have been a good idea despite the early decision to use random level generation in the final game. Another area which was made harder by the randomisation than it had to was the framerate testing as no two test were exactly the same. This could have been solved by using the same seed for the randomisation each time causing the level to always look the same. However, it would not have been a perfect solution as some extreme cases might have been missed.

9.3 Graphics

Developing a game gives a first-hand experience of the difference certain graphic effects make when implemented. For the untrained eye, the importance of for example ambient occlusion for the realism of a 3D scene can be hard to grasp. The improvement was the most prominent with the introduction of lights and shadows, but was definitely present for animations and particle effects as well. Though some of the implementations were not perfect, such as the prerendered shadow that can still be seen when the ghost jump between platforms, they still go a long way in enhancing the visual aspect of the game, especially when many times the only other option was to have no effect at all due to the limitations of the phone. Especially interesting is how far graphics go towards making the game have such a high quality that it can be released. Without changing any of the game logic, without adding more content to the level, the presence of torches with particles and lighting improved the quality of the game tremendously, reminding us that digital games is very much a visual medium.

10 Conclusions

The purpose of this project was to build a simple yet functioning and graphically intense platform game for an Android smartphone. The Jumping Dead is the result of the work that the group has invested towards fulfilling the purpose. During the development, issues were encountered and solved. These have been summarised in a condensed format in the following paragraph, which can be seen as a rough set of recommendations when creating a similar project.

The development of The Jumping Dead showed that some effects had a much larger impact on overall performance than others. Especially dynamic light, shadows and post processing effects had such a drastic impact that they were either cut back or skipped entirely. However, embedded lighting and texture-based ambient occlusion were shown to yield similar visual results at a much lower cost. Such methods could serve as satisfactory replacements for the more computationally intensive versions.

By adapting the detail of models to screen size, and using textures rather than triangles, performance could be improved without losing visual quality.

Particle systems are highly customisable and computationally cheap, making them very useful for a graphics-intense smartphone game. In The Jumping Dead they were used to model fire, dust, magic spells and power-ups. Furthermore, JME provides an easy-to-use implementation that runs on Android without problems.

Furthermore, the use of a preexisting game engine freed up time which could be spent on other areas of the development. However, it is important to realise that while game engines undoubtedly serve as good foundations for a project, they are not tailored to a specific project, which can cause issues. In the case of this project, the engine-specific implementations of certain effects did not work on the phone. Not having complete control over those implementations was an obstacle in the way of improving on them. For future projects, we recommend that the choice of framework or game engine is made after serious consideration of the advantages and disadvantages of the alternatives.

References

Akenine-Möller, T., E. Haines, and N. Hoffman (2008). *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., p. 1045. ISBN: 987-1-56881-424-7.

Anders, M. (2010). *Blender 2.49 Scripting*. Olton, Birmingham, GBR: Packt Publishing Ltd, p. 292. ISBN: 978-1-84951-040-0.

Bouquet, R (2010). *Screen Space Ambient Occlusion for jMonkeyEngine 3.0. jMonkeyEngine.org*. URL: `http://jmonkeyengine.org/2010/08/screen-space-ambient-occlusion-for-jmonkeyengine-3-0/` (visited on 05/05/2013).

Bunnell, M. (2005). "Dynamic Ambient Occlusion and Indirect Lighting". In: Pharr, M. *GPU Gems*. Upper Saddle River: Addison-Wesley, pp. 223–233.

Entertainment Software Association (2012). *Essential facts about the computer and video game industry*. URL: `http://www.theesa.com/facts/pdfs/esa_ef_2012.pdf` (visited on 04/15/2013).

Filion, D (2011). "Principles and Practise of Screen Space Ambient Occlusion". In: Lake, A. *Game Programming Gems*. Boston: Cengage Learning.

Flavell, L. (2010). *Beginning Blender, Open Source 3D Modeling, Animation, and Game Design*. New York, NY, USA: Apress, p. 448. ISBN: 978-1-4302-3126-4.

Hürsch, W. L. and C. V. Lopes (1995). *Separation of Concerns*. Tech. rep.

jbullet.advel.cz (2008). *JBullet*. URL: `http://jbullet.advel.cz/` (visited on 06/06/2013).

jMonkeyEngine (2012). *JME3 Documentation: Physics Listeners*. URL: `http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:physics_listeners` (visited on 04/22/2013).

— (2013a). *JME3 Documentation: Android Support in the jMonkeyEngine*. URL: `http://jmonkeyengine.org/wiki/doku.php/jme3:android` (visited on 05/16/2013).

jMonkeyEngine (2013b). *JME3 Documentation: Material Definition Properties*. URL: `http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:materials_overview#phong_illuminated` (visited on 05/14/2013).

jMonkeyEngine (2013). *Material Deinition Properties*. URL: `http://hub.jmonkeyengine.org/wiki/doku.php/jme3:advanced:materials_overview` (visited on 05/10/2013).

Jörnmark, J., A.-S. Axelsson, and M. Ernkvist (2005). "Wherever Hardware, There'll be Games: The Evolution of Hardware and Shifting Industrial Leadership in the Gaming Industry". In: *DiGRA 2005: Changing Views: Worlds in Play, 2005 International Conference. June 16–20 2010, Vancouver*.

Küller, R. et al. (2006). "The impact of light and colour on psychological mood: a cross-cultural study of indoor work environments". In: *Ergonomics* 49.14, pp. 1496–1507.

McGuire, M. (2010). "Ambient Occlusion Volumes". In: *Proceedings of High Performance Graphics 2010. June 25–27 2010, Saarbrucken*.

Mercury [pseud.] (2013). *Sonic Physics Guide: Jumping*. URL: `http://info.sonicretro.org/SPG:Jumping` (visited on 05/15/2013).

Mullen, T. and E. Coumans (2008). *Bounce, Tumble, and Splash! : Simulating the Physical World with Blender 3D*. Hoboken, NJ, USA: Wiley, p. 400. ISBN: 978-0-4703-9272-0.

Pongmuseum.com (2013). *Atari PONG*. URL: `http://www.pongmuseum.com` (visited on 04/17/2013).

The Blender Foundation (2013). *Blender.org*. URL: `http://www.blender.org` (visited on 04/17/2013).

Vainer, K. (2013a). *Creating assets in Blender3d*. URL: `http://jmonkeyengine.org/wiki/doku.php/jme3:external:blender` (visited on 04/18/2013).

— (2013b). *How to Use Material Definitions*. URL: `http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:material_definitions` (visited on 04/19/2013).

— (2013c). *Models and scenes*. URL: `http://www.jmonkeyengine.org/wiki/doku.php/jme3:advanced:3d_models` (visited on 04/19/2013).

Wolff, D. (2011). *OpenGL 4.0 Shading Language Cookbook*. Olton Birmingham: Packt Publishing Ltd.

Secondary sources

Cited in Akenine-Möller, Haines, and Hoffman (2008):

Evans, A (2006). "Fast Approximations for Global Illumination on Dynamic Scenes". In: *SIGGRAPH 2006 Advanced Real-Time Rendering in 3D Graphics and Games course notes*. SIGGRAPH.

Kontkanen, J and S Laine (2005). "Ambient Occlusion Fields". In: *ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*. SIGGRAPH, pp. 41–48.

— (2006). "Ambient Occlusion for Animated Characters". In: *Eurographics Symposium on Rendering*, pp. 343–348.

Landis, H (2002). "Production-Ready Global Illumination". In: *SIGGRAPH 2002 Course Notes, Course 16: RenderMan in Production*. SIGGRAPH, pp. 87–102.

Luft, T., C. Colditz, and O. Deussen (2006). "Image Enhancement by Unsharp Masking the Depth Buffer". In: *ACM Transactions on Graphics* 25.3, 1206–1213.

Malmer, M. et al. (2007). "Fast Precomputed Ambient Occlusion for Proximity Shadows". In: *Journal of Graphics Tools* 12.2, pp. 59–71.

Mittring, M. (2007). "Finding Next Gen: CryEngine 2". In: *SIGGRAPH 2007 Advanced Real-Time Rendering in 3D Graphics and Games course notes*.