KARLSTADS UNIVERSITET

Report DVGB06 - Tillämpad Programmering S13

Anders Nord - andno922@student.liu.se
August 18, 2013

This report will discuss how to create a simplistic game-engine in WebGL. Included features are:

- Shadow mapping
- Blinn-Phong shading model
- .obj-file loader
- Applying textures
- Simple bounding box collision detection
- Parallax scrolling

CONTENTS

1 INTRODUCTION

To create something as extensive as a game-engine is a great challenge. Where to start?

What is the next step? How will this affect future implementations in the code?

This report will address many common concepts in computer(game) graphics. Computer games have been played for a long time now. But to be able to just enter a website and get the fantastic graphics that WebGL can provide if implemented correctly is something new to many. Even though WebGL has been around for a while, now is the time web browser developers really have started to implement it as a standard feature. There is still a lot of fixes to be done and some web browsers need special treatment. For this project google chrome was used as a web browser.

For a look at the engine, visit http://andersnord.bitbucket.org/.

1.1 Structure of the report

First an overview of the project will be presented. The main steps will be examined superficially.

After that a more detailed explanation of the more technical steps will be provided.

2 Method

The game engine was implemented in WebGL using shaders. WebGL communicates with the graphics-card through OpenGL ES 2.0-standard shaders. To access these in html, JavaScript is being utilized. The matrix library used was glMatrix v0.9.5 [1].

2.1 Getting started

The first thing to do is choose some sort of design pattern to follow. This will make it much easier later in the project. In this project the MVC(Model View Controller) was applied. This separates the input, calculations and rendering. By doing this it is easier to keep track on what is going on and where.

The second thing is to start using some sort of Software development process. Online Kanban [2] was used for this project, which is a version of scrum.

The third thing would be to create some sort of time schedule for the project and update it once a week.

The fourth thing is to create the basic codebase for the project. In this case it would be setting up the shaders and compiling them, creating a render loop and get something basic to show up on the screen. See fig: 4.1.

The fifth thing is to use some sort of version control system designed to handle projects online. This project used bitbucket [11], but another popular choice is github [10].

2.2 A quick look at how the engine is built up

Fig: 2.1 is a screenshot from the final version of the engine. The flying islands in the background are 2D planes with attached PNG-textures. The whole background is just different 2D planes with textures on them. The player, tree, flying stone islands and the low-poly sphere are 3D objects.

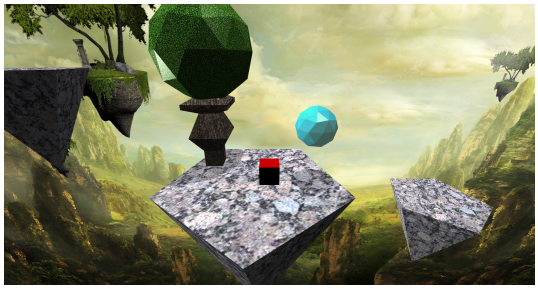

Figure 2.1: Screenshot from end product

2.3 Game control and physics

Keyboard keys are easily obtained through JavaScript. The player is then translated according to the input the player is giving. The physics are just simple gravity and friction.

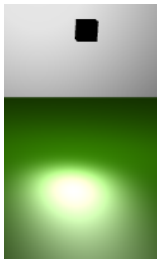

Figure 2.3: Specular highlight

2.4 Collision detection

The implemented collocation detection is a bounding box in two dimensions. Every object in the engine has a scaled invisible rectangle around them. All the mathematical calculations treating collisions are taking these into consideration, not the actual geometry of the object. See fig: 2.2.

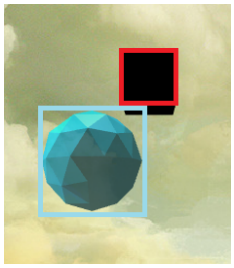

Figure 2.2: The rectangles are what is really colliding, not the 3D-models.

2.5 Light

For all of the 3D-objects a value is being calculated depending on a light source. The model used is Blinn-Phong-shading. This creates beautiful specular highlights, see fig 2.3

2.6 perspective, FOV and aspect-ratio

To get a sense of FOV(Field Of View), the perspective-matrix from the glMatrix library [1] was used . This also takes the aspect-ratio into account. See fig 2.4 and 2.5 for a comparison.

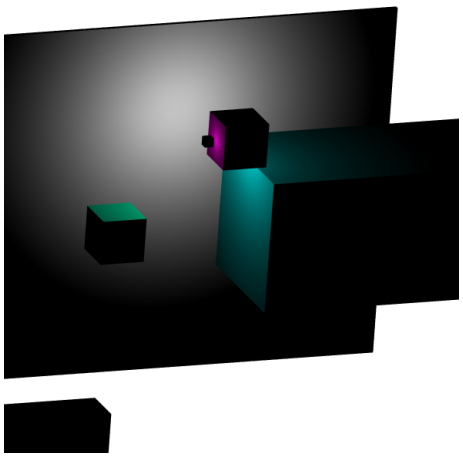

Figure 2.4: Before adding perspective matrix

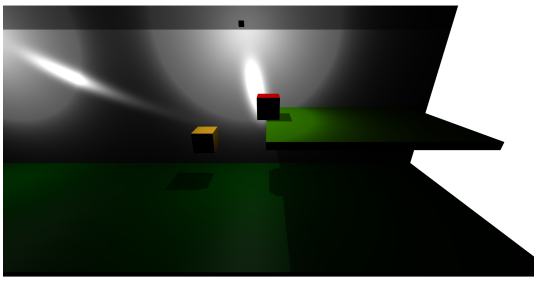

Figure 2.5: After adding perspective matrix

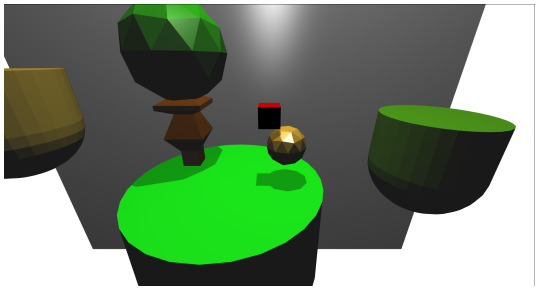

Figure 2.6: Before adding textures

2.7 Shadow mapping

This is a method often used in real-time applications to create shadows. First a depth-buffer-texture is being rendered via an FBO(Frame Buffer Object) from the light-source point of view. This texture is then being used to calculate if a vertex is being blocked and shadowed by any object in the scene.

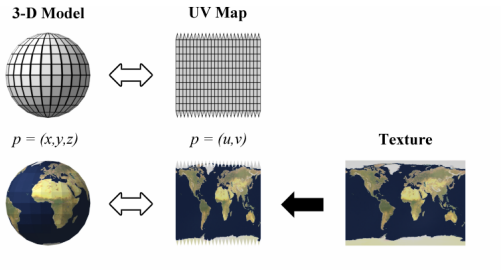

Figure 2.7: UV mapping [9]

2.8 Object files reader

The engine uses a simple .obj-file-reader. It has been specifically written for blender-files, but should be able to handle this format generally [7]. When using an .obj file with verticies, normals and UV-coordinates, three arrays according to the triangle-list in the object file, will be created.

2.9 Textures

Applying textures to an object is not very difficult. Without the UV-coordinates though, textures become really dull. What they do is telling the program which position in the texture a specific vertex should get its colour from, see fig 2.7. Textures in general really brightens up the experiance. Compare fig: 2.1 with fig: 2.6.

2.10 Parallax scrolling

This is a method where many different 2D planes are being placed with different distances to the camera. Then when side-scrolling it creates a feeling of depth and movement in the background. See fig: 2.8.

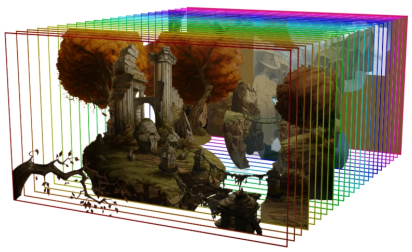

Figure 2.8: Parallax scrolling created with layers of planes. [8]

3 Detailed description

3.1 Collision detection

The collision detection is using bounding boxes. The reasons for this is that it is all about the question "is rectangle A inside rectangle B?". If this calculation was done for every polygon, it would be very computationally intensive. But for rectangles it is much faster, it is also easier to implement.

So the things needed to be kept in mind are: "Is A inside B?" and if it is, then "from which direction did it come?". To be able to answer these questions, all the edges of the boxes has to be checked.

The only thing to understand is that if, for example, rectangle A´s bottom edge is above rectangle B´s top edge, then we know there is no collision occurring. So by doing this check for all four sides, it is easy to know when an intersection between A and B is happening. If any of these four conditions are true, there is no intersection happening.

- A´s bottom edge is higher then B´s top edge
- A´s top edge is lower then B´s bottom edge
- A´s left edge is to the right of B´s right edge
- A´s right edge is to the left of B´s left edge

It also means that if none of these conditions are true, then an intersection is happening.

The next thing to check for is: "From which direction did it come?". This can be done by a simple check, "which edge is closest to which edge?". The green lines in fig: 3.1 indicate that the red box came from the upper right corner. By comparing the length of these two green lines it is possible to know that the rectangle came from above. See fig: 3.1.

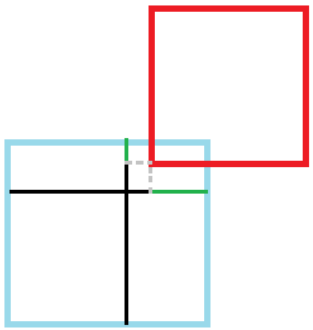

Figure 3.1: An intersection between two rectangles.

When this is known, the red rectangle gets moved in positive y-axis the length of the shortest green line + an arbitrary value.

3.2 Light

The implemented model is the Blinn-Phong shading model.

The method used is a mix between pixel shading and Blinn-Phong-shading. This is not pure Blinn-Phong-shading, because without calculating new normals accordingly to the Phong-shading method, it is called pixel shading.

The calculations are happening in the fragment shader, which means, for every pixel. This is also the reason why it is not calculated for the 2D planes. It takes a lot of extra power when the resolution is high. The planes receives the colour that the texture contains without any shading.

Because of using the fragment shader the pixel values gets interpolated between the vertices

in the triangle polygon, see fig 3.2.

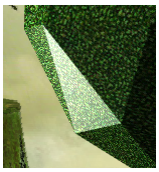

Figure 3.2: Pixel shading.

3.2.1 Blinn-Phong shading model

The model consists of 3 terms. The ambient , diffuse and specular light.

The ambient term is just an arbitrarily chosen value added to all objects so they wont be totally dark.

The diffuse term is the core of the model. This value will tell us how much the pixel should be lightened up. The diffuse value is being calculated by taking the dot-product between the vertex normal and the light direction. See Eq: 3.1.

$$Diffuse = Normal \cdot LightDir \tag{3.1}$$

For a visual example see fig: 3.3 where the small black box is a point light source.

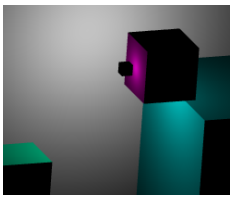

Figure 3.3: Diffuse term.

The specular term is being calculated by first creating a halfvector according to Eq: 3.2. LighDir and ViewDir are both normalized.

$$Hvector = Normalize(LighDir + ViewDir) \tag{3.2}$$

The dot-product between the halfvector and the normal is then being squared to the power of an exponent. The exponent α determines how the specular term affects the model. See Eq: 3.3.

$$Specular = (Halfvector \cdot LightDir)^{\alpha} \tag{3.3}$$

For a visual example see fig: 2.3 and examine fig 3.4. Notice in which direction the vectors are pointing.

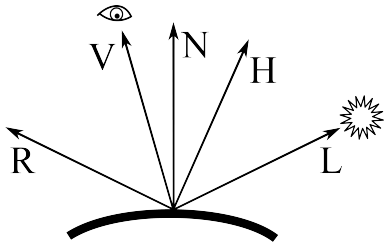

Figure 3.4: Vectors for calculating Phong and Blinn–Phong shading [4].

To get the specular light right when rotating the object, the normals have to be multiplied by the transposed inverse of the modelView-Matrix.

For a more detailed explanation and code example, visit [4].

3.3 Shadow-mapping

Primarily a depth-buffer-texture is being rendered via an FBO(Frame Buffer Object) from the light source point of view. This type of texture contains a value in each pixel which says

how far away from the point of rendering the objects are. It is a grayscale image, see fig: 3.5. This is the scene from the same time-step as 3.6.

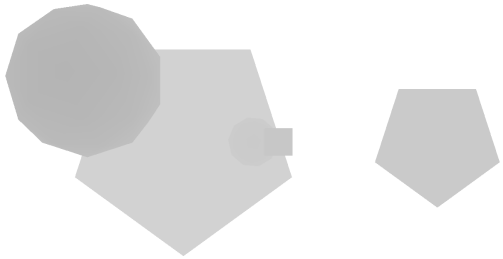

Figure 3.5: The depth texture rendered from the light source, looking down.

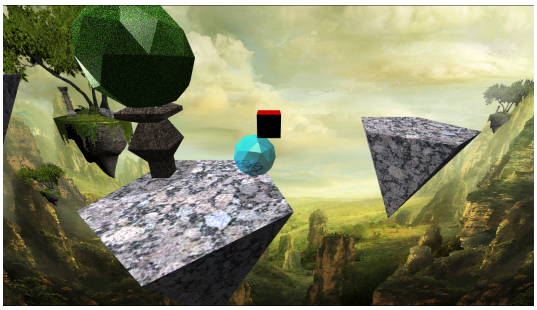

Figure 3.6: The game-view rendered at the same time as 3.5.

The pixels in fig: 3.7 that are behind the obstacles will be shadowed since they have a larger depth-value than the obstacle in the depth texture. So what needs to be done is that when rendering the scene from the game-view, every pixel has to be transformed from the view in fig. 3.6 to the view in fig: 3.5 and checked by its depth-value, the transformed vertex z-coordinate in our case.

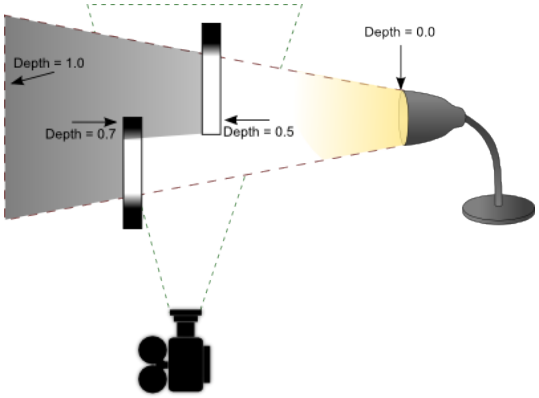

Figure 3.7: The basics of shadow mapping.

The vertex is first transformed by the light source projection matrix then the light source view matrix and finally by the modelview matrix. See Eq: 3.5.

$$V_L = M_P * M_V * M_M * V \tag{3.4}$$

Where

- M_P is the light source projection matrix.
- M_V is the light source view matrix.
- M_M is the modelview matrix.
- V is the vertex being transformed.
- V_L is the projected vertex from the light source.

When using this technique shadow acne, as seen in fig: 3.8, will often appear. This is because of numerical precision. The solution to this is adding a bias value to the z-coordinate. This solves the shadow acne problem but creates a new one called peter panning. See fig: 3.9. The shadow gets moved away from the object.

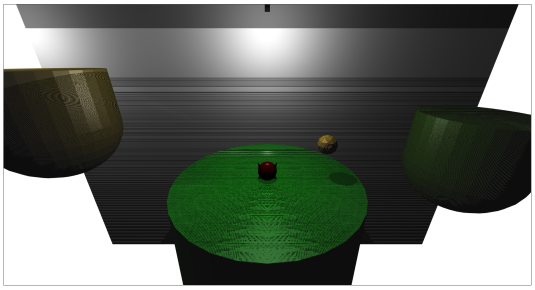

Figure 3.8: Shadow acne.

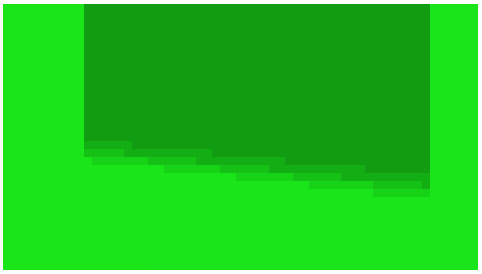

Figure 3.10: Poisson sampling.

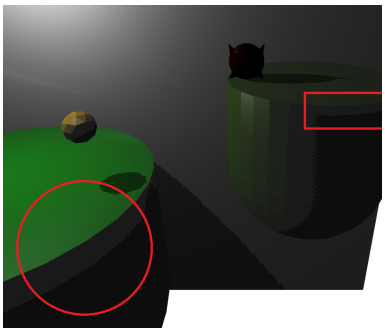

Figure 3.9: Peter panning.

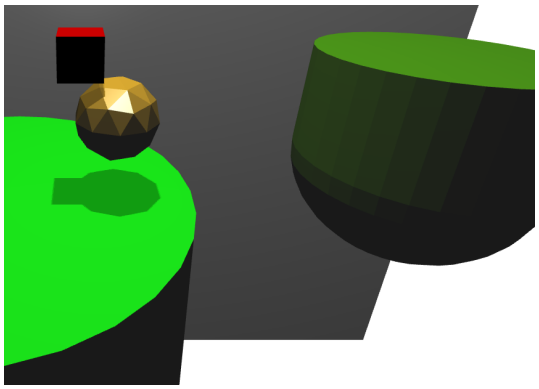

Figure 3.11: Final reuslt with shadows.

To create a bias value that fits the specific curvature, the value is depending on the slope. See Eq: 3.5 for a code example. This makes the peter panning disappear.

$$\begin{aligned} float \quad cosTheta &= Normal \cdot LightDir; \\ float \quad bias &= 0.004 * tan(acos(cosTheta)); \\ bias &= clamp(bias, 0, 1.0); \end{aligned} \tag{3.5}$$

To avoid aliasing, soft shadows are created with a method called Poisson sampling. It is basically a standard sampling of the neighbouring pixels from the depth texture. See fig: 3.10. The Poisson values are used to create a kind of randomness when sampling the pixels around the current pixel.

The poisson values were obtained from the source-code at [6].

For a detailed explanation visit [6] and [5].

4 Result

The following is a series of pictures which shows the development of the engine.

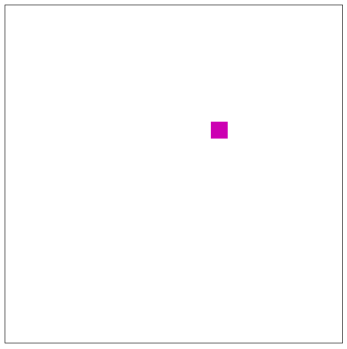

Figure 4.1: Week 1: Just being able to move a cube around.

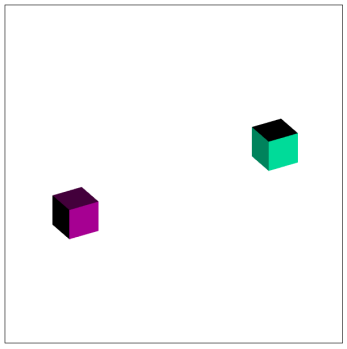

Figure 4.2: Week 2: Going to 3D and adding pixel shading.

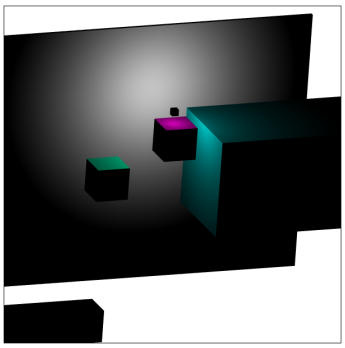

Figure 4.3: Week 3: Adding bounding boxes and distance depending light.

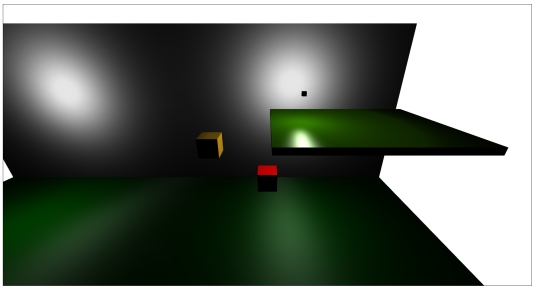

Figure 4.4: Week 4: Adding perspective matrix and specular light.

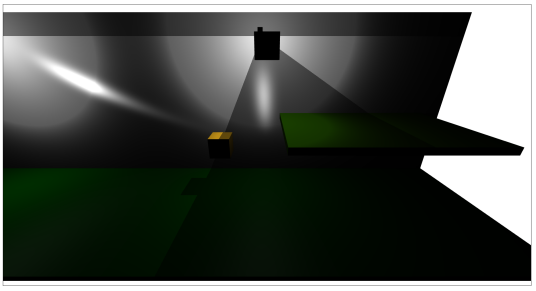

Figure 4.5: Week 5: Adding shadows.

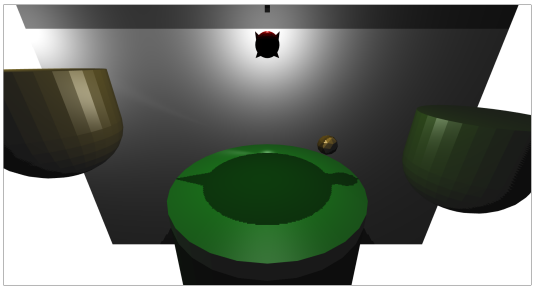

Figure 4.6: Week 6: Adding object loader.

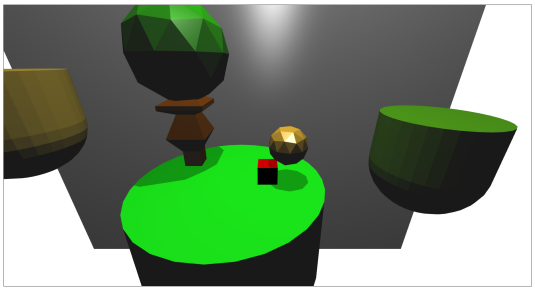

Figure 4.7: Week 7: Improved shadows.

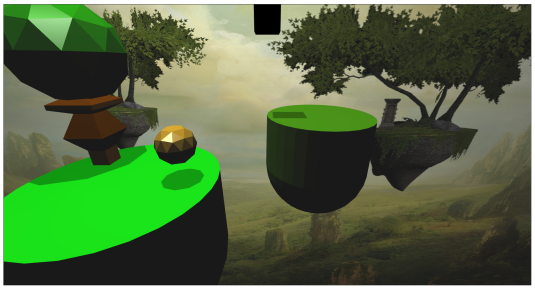

Figure 4.8: Week 8: Added textures.

Figure 4.9: Week 9: Added hovering islands and texture to all objects.

4.0.1 Computer specs

These test were run on this machine:

Processor: Intel Core i7 3610QM 2,3 GHz
Memory : 8 GB of DDR3 1600 MHz SDRAM
Graphics card: Intel(R) HD Graphics 4000 that has a graphics memory of 1792 MB
OS: Windows 8 64-bit

4.0.2 Performance measurements

Table 4.1: Performance results in resolution and FPS

Resolution	FPS
1280 * 720	66
1920 * 1080	62
3000 * 3000	20

5 Discussion

To summarize this project the theory have been consistent with what had to be done, but this does not mean that it was easy to implement. This is because of many reasons. It was the first time I really implemented something big in WebGL and I had to find and programme everything by myself. I could never really ask someone about anything technical so I had to look it up by myself.

The thing put most time into, probably, was the shadow mapping. It was really tricky to get get the desired result. I tried a lot of different methods and had to adept them to my likings. It was hard not to get any shadow acne and at the same time not get any peter panning. I also tried different number of Poisson samples and culling the front faces when creating the depth texture. But in the end I decided that these methods was not working out well enough.

In the shadow calculations I also tried to use the distance between the light source and pixel to get a better estimation. But when using this value it did not work out well at all, so the pixel z-value was used in the end anyway.

By using Kanbanize i could still have a pretty good overview of what had to be done and how to prioritize. To decide what had to be done next was still hard since I have no real experience of creating a game engine from before.

During the development a constant thinking about good and efficient solutions existed. In a game everything is happening in real time so it has to be very efficient code. Lots of times code had to be re-written and solutions had to be done differently to avoid lag. What it really is about is cheating as much as possible without anyone noticing it.

Sound was actually implemented partially. It wont be in the released version though until it works perfectly. This would make the experience better if, for example, a sound played every time the player jumped.

This has been a really rewarding project and has given insight into many different parts of computer graphics and graphical game related solutions.

References

[1] Matrix library, `https://code.google.com/p/glmatrix/wiki/Usage`.

[2] Matrix library, `http://kanbanize.com/`.

[3] Information and example of implementing bounding boxes. `http://devmag.org.za/2009/04/13/basic-collision-detection-in-2d-part-1/`.

[4] Information and example of implementing Blinn-Phong shading. `http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model`.

[5] A tutorial on how to create shadow mapping, `http://devmaster.net/posts/3002/shader-effects-shadow-mapping`.

[6] Another tutorial on how to create shadow mapping `http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/#Shadow_acne`

[7] Information about the .obj-format, `https://en.wikipedia.org/wiki/Wavefront_.obj_file`.

[8] Parallax scrolling, `http://en.wikipedia.org/wiki/Parallax_scrolling`.

[9] UV mapping, http://en.wikipedia.org/wiki/UV_mapping.

[10] Github, a version control system designed to handle projects online. https://github.com/.

[11] Bitbucket, a version control system designed to handle projects online. https://bitbucket.org.