CPS3231 - Computer Graphics
Arkanoid 3D Assignment

Aidan Cauchi

Faculty of ICT
University of Malta

19/01/2020

Contents

Chapter 0

Introduction

This report contains the high level overview describing how the tasks in the assignment specification were implemented. It is split into a number of chapters, each describing how the respective task was carried out. The source files are supplied along with this report. The intention of the report is to give the reader a bit of intuition before looking at the source files.

Chapter 1

Task 1

This chapter contains the relevant information explaining how task 1 was completed. This includes an overview of the map generation, the ambient lighting and textures used.

1.1 Map Generation

The map is generated using the *arkanoidMap.generateMap(columns, rows, gl, scene)* and is set up as follows:

- The map is generated starting from (0,0) and grows to the right.
- All entities in the game are children of the map node in the scene graph. They are placed in the scene relative to the map.
- Its transform is then updated, where $x = -centre$. Setting its transform to the negative value of the centre ensures that the map now appears symmetric and all other blocks defined relative to the map are now easily transformed to the centre of the screen.

Given the knowledge above, Figure 1.1 explains how the blocks are generated. Since everything is relative to the map, the x coordinate can simply be increased by the block width and the offset on for every block until the maximum number of blocks per row is reached. Once this happens, the x coordinate is reset, and the y coordinate grows downwards, taking into account the row number and the height and offset of the blocks. After all the breakable blocks are generated, the border blocks are generated using the same principle, taking into account the width and offsets for all blocks. Finally, the ball and the paddle are generated and put at the bottom centre of the map. While this outlines the general idea, the *gameMap.js* file contains the code that explains it in more detail.

In addition to the blocks, a single directional light source is also created, looking at (0,0,-1).

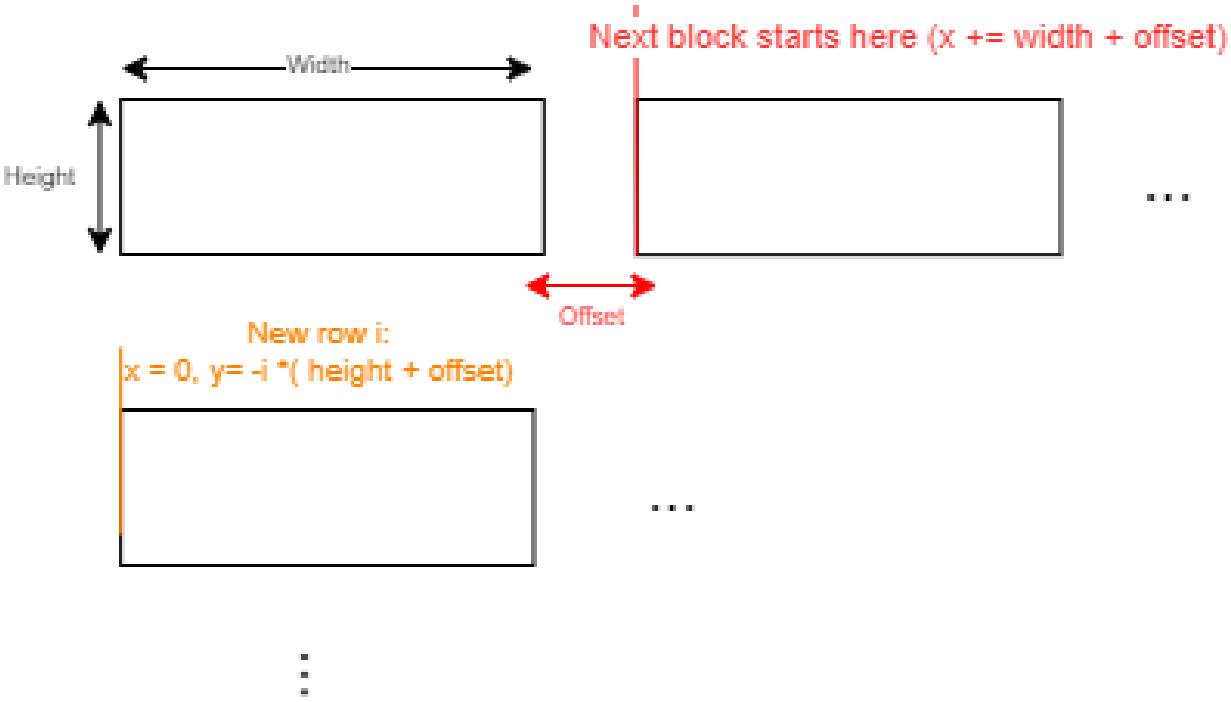

Figure 1.1: Outline for generating blocks

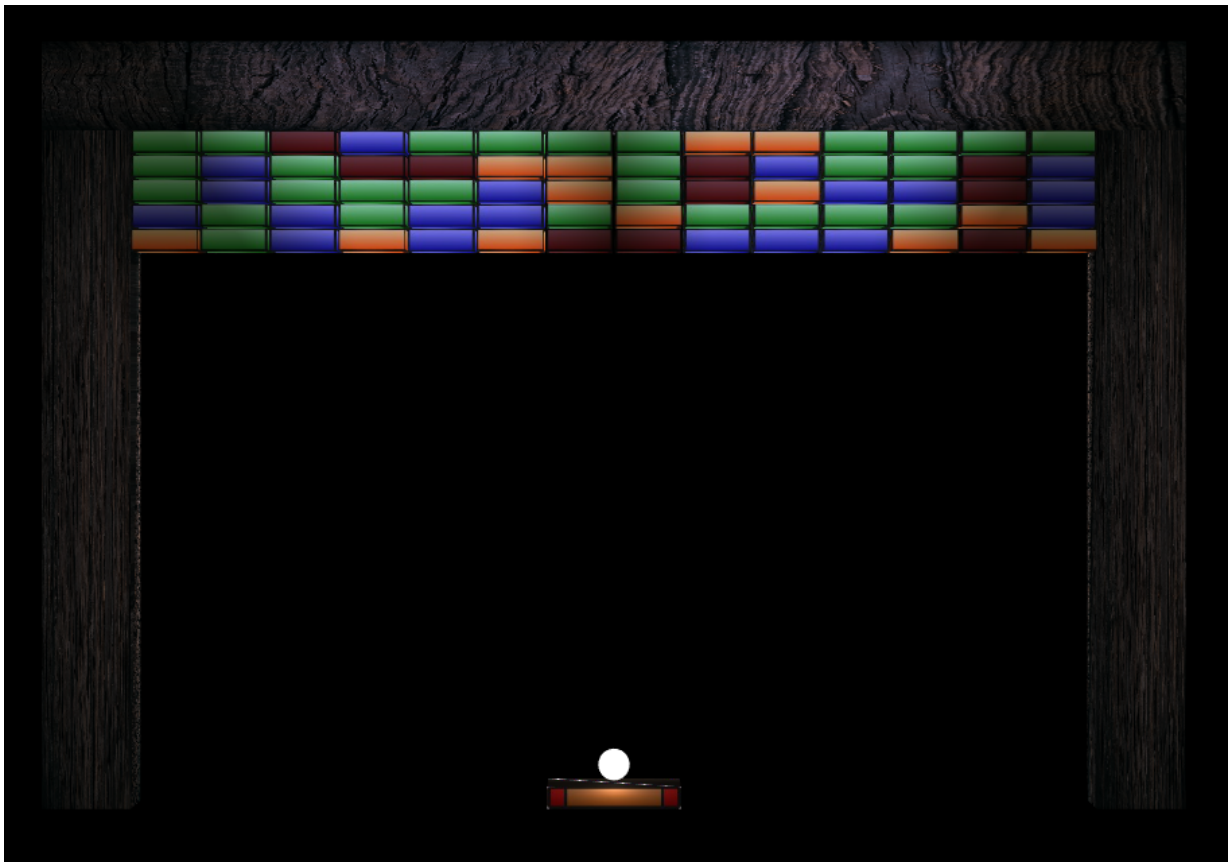

Figure 1.2: A fully generated map

1.2 Textures

The textures used can be found within the assets folder. These are also found in the *texture.js* file in base64 format. As can be seen by Figure 1.2, the blocks emlpoy a diffuse shading model while the ball employs a specular one, hence the shininess. Before generating the blocks, the textures are loaded in memory. The paddle, borders and ball always have the same texture. However, the blocks are randomly assigned materials every new game. A random number is generated which during each block generation, which corresponds to the index of the material to use in the materials array.

```
for(var j = 0; j < blockColumns; ++j)
{
    var matIndex = Math.round(Math.random() * 3);

    var blockModel = new Model();
    blockModel.name = "Block" + blockId++;
    blockModel.index = block.index;
    blockModel.vertex = block.vertex;
    blockModel.material = materials[matIndex];
    blockModel.compile(scene);
```

Figure 1.3: A snippet of the block generation loop.

The authors of the textures can be found in the footnotes below [1] [2].

1.3 Model Generation

Prior to placing an object in the scene, the object must be generated. The *generateBlock(positions, normals, colours, uvs)* and *makeSphere(centre, radius, h, v, colour)* methods are used to generate block and sphere models respectively. The implementations can be found in the *brick.js* and *gameMap.js* (near the end) files respectively. The code explains it better than the writer can in words.

[1] https://opengameart.org/content/basic-arkanoid-pack
[2] https://unsplash.com/photos/6vvIBTvL90A

Chapter 2

Task 2

This section explains how the camera movement was implemented, and also describes how user input was read.

2.1 Reading User Input

In order for the player to interact with the game, event listeners were used. Specifically, the **'keyup'** and **'keydown'** event listeners were used, which are called when a user presses a button or releases it respectively. Furthermore, the game makes use of a dictionary containing the state of each button called '*keys*'. Each entry in '*keys*' contains the key mapped to true or false, depending on whether it is currently pressed or not.

```
document.addEventListener('keydown', (e) => {
  switch(e.key)
  {
    case 'q': keys['q'] = true ;break;
    case 'e': keys['e'] = true ;break;
    case 'w': keys['w'] = true ;break;
    case 's': keys['s'] = true ;break;
    case 'r': keys['r'] = true ;break;
    case 't': keys['t'] = true ;break;
    case 'd': keys['d'] = true ;break;
    case 'a': keys['a'] = true ;break;
    case ' ': {
      keys[' '] = true ;
      e.preventDefault(); //to prevent scrolling downwards
    };break;
  }
});
```

Figure 2.1: A snippet of how the 'keydown' event listener works

2.2 Camera Movement

In order to move the camera, the game makes use of four variables: *theta*, *theta_min*, *theta_max* and *speed* (found in *render.js*). *theta_min* and *theta_max* specify the minimum grazing angle and the maximum grazing angle respectively. When the user presses '**S**', *theta* starts increasing with the value of *speed* per frame until *theta_max* is reached. Conversely, when '**W**' is pressed, it is decreased until *theta_min* is reached. This creates the interpolation between the top and grazing angle (70°). Pressing '**Q**' sets theta to 0 and '**E**' sets theta to 70°. Pressing '**R**' increases the speed variable and '**T**' reduces it (up to 0.005). On each new frame, the camera is rotated along the x axis with the current value of theta.

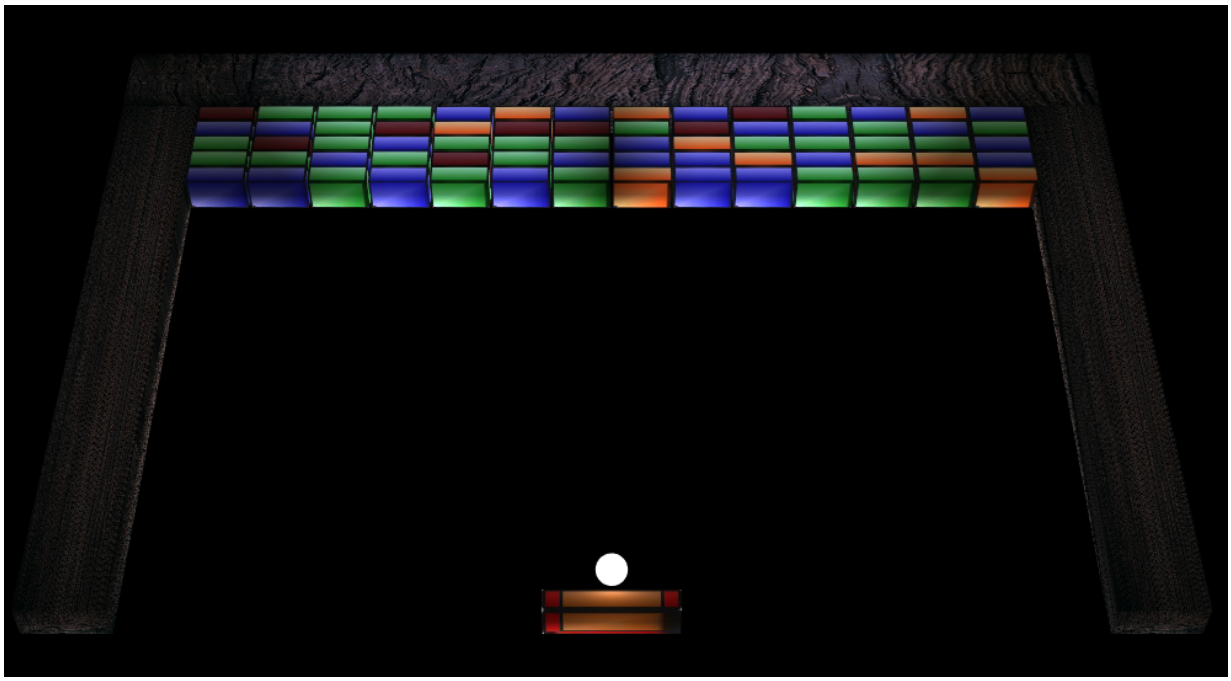

Figure 2.2: The map shown through a rotated camera

Chapter 3

Task 3

This chapter contains a high level description of how the physics system works, which consists of collision detection, player input and ball reflection.

3.1 Physics Body

In order to be able to perform physics calculations, an object called a *physicsBody* was used. A physics body could either be a rectangle (brick), sphere or the paddle. By making use of *physicsBody.generateBoxBody()*, *physicsBody.generateSphereBody()* and *physicsBody.generatePaddleBody()*. Physics bodies are simply invisible objects in the scene, where collision detection tests happen on the physics body themselves, rather than the shape they represent. Given they are represented as objects in the world, this makes it easy to transform them around in the scene, using the world space coordinate system.

The *node* object in the scene graph was then modified to also hold a reference to a *physicsBody* object along with the *nodeObject*. Another variable was added called *isDestructible* which is used for breaking bricks (explained later on).

3.2 Movement

In order to move about, physicsBody objects can also make use of a vector called *direction*. Every time the animation callback is invoked, the distance travelled is calculated, making use of a *speed* variable and *deltatime* to calculate the distance travelled. Afterwards, the distance is multiplied by the direction vector, where the result is then used to transform the object from the previous point to the new point. The model which the physicsBody represents is also updated with this new position.

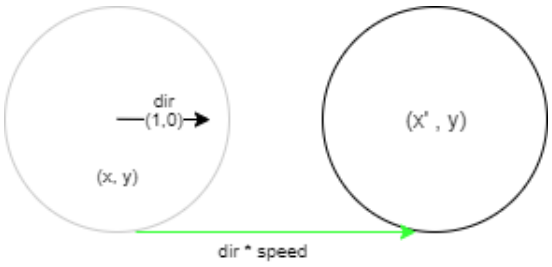

Figure 3.1: An example showing how the ball is transformed every frame

For the paddle, the principle is the same but the direction vector is modified when the player presses

'**D**' to go right and '**A**' to go left. Moreover, the longer the **spacebar** is held when the ball is attached to the paddle, the higher the speed (up to a limit).

3.3 Intersection

In order to check whether a physicsBody has hit another one, the 'node' object in the scene graph was updated to have a method called *Node.getIntersectingNode(physicsBody)*. This then checks whether the supplied body intersects with the node or any of its children. The intersecting node is returned or null otherwise. This can be found in the *scene.js* file along with the scene graph. The intersection test is carried out in every frame, by the required objects, using the animation callback function in the scene graph.

There are 3 types of collisions that can occur:

1. Sphere - Sphere collision
2. Box - Box collision (for laser powerup)
3. Sphere - Box collision

For the first case, the distance between the centres of the spheres is calculated. If this is less than the sum of the radii then a collision has occurred. For the second case, all sides of the boxes are checked to see whether they lie in the physicsBody of each other. If this is the case, a collision has occurred. For the last collision, the maximum x and y coordinates of the sphere or the box are calculated. The distance from the ball to this new coordinate is calculated and if it lies inside the radius of the ball, a collision has occurred.

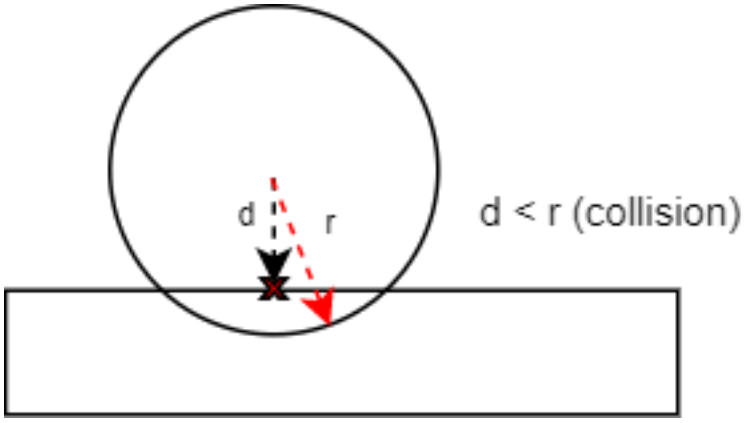

Figure 3.2: Sphere - Box collision

3.4 Bounce

When the ball collides with a physicsBody, it will then proceed to travel in the reflected direction. The way this is calculated is as follows:

- If the ball hits the bottom or the top of the brick, its y direction is reflected.
- If it hits the side of the brick, its x coordinate is reflected.

This is quite simple to implement as if the ball is going left and hits the top or bottom, it will still keep going left but now also goes downwards or upwards respectively. The same idea applies for the sides.

Before a ball bounces off, it will check whether the body it collided with is destructible, using the *isDestructible* variable mentioned before. If this is the case, it will deactivate the block and decrease the number of blocks left until the games ends (it will also check some other variables, as will be seen in the powerup section). Else it would simply bounce off.

Figure 3.3: Screenshot of the game after some bounces

An exception to this rule is when the ball hits the paddle. The ball is not reflected off of the centre of the paddle but rather a point slightly below it.

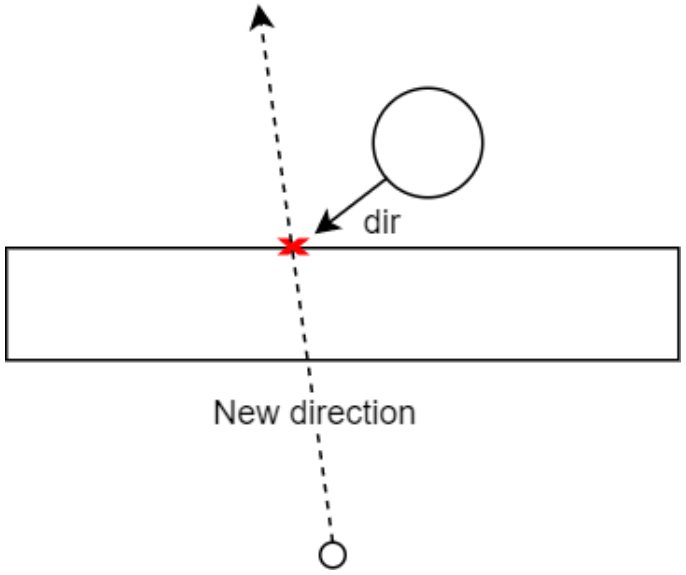

Figure 3.4: Paddle bounce mechanics

3.5 Ending the game

Below the paddle, there is an invisible physics body called the fail block. Upon colliding with the fail block, the ball will reset and the lives left are decreased. If the balls goes outside 3 times, the game freezes signalling a loss.

Chapter 4

Task 4

This chapter briefly explains the changes brought about in order to handle additional illumination.

4.1 Shader Updates

The same shader used in class for other projects was used however with some slight modifications. The data that was used for one light source was now expanded into an array of 4, since that's the total number of lights in the scene. Then the vertex and fragment shader loop through all four light sources, performing the necessary calculations and storing the results in the respective index of the light sources. The *light.js* was also modified a bit, adding an *inUse* flag stating whether the loop should consider the light source at that moment in time.

```
varying vec3 vColor;
varying vec3 vNormal;
varying vec3 vEye;
varying vec3 vLight[4];
varying vec3 vLightAxis[4];
varying vec3 vLightWorld[4];
varying vec2 vTexCoords;
```

Figure 4.1: Snippet showing the modified shader variables

4.2 Lights Added

The types of new lights included two spotlights and one point light. The callback of the point light is that its transform is updated to the transform of the ball in order to follow the ball. On the other hand, the callback of the spotlights sets their transform to the paddle transform, with some offset to the left and right in order to put them at the side. The map lit up with the the new light sources can be seen by Figure 1.2, as previously shown. Furthermore, the code for the light generation is found in *gameMap.js* in the *setUpLights()* method while the shader code is found in *index.html*

4.3 Scene Graph

Figure 4.2 shows the scene graph of the game to accommodate for the new light sources.

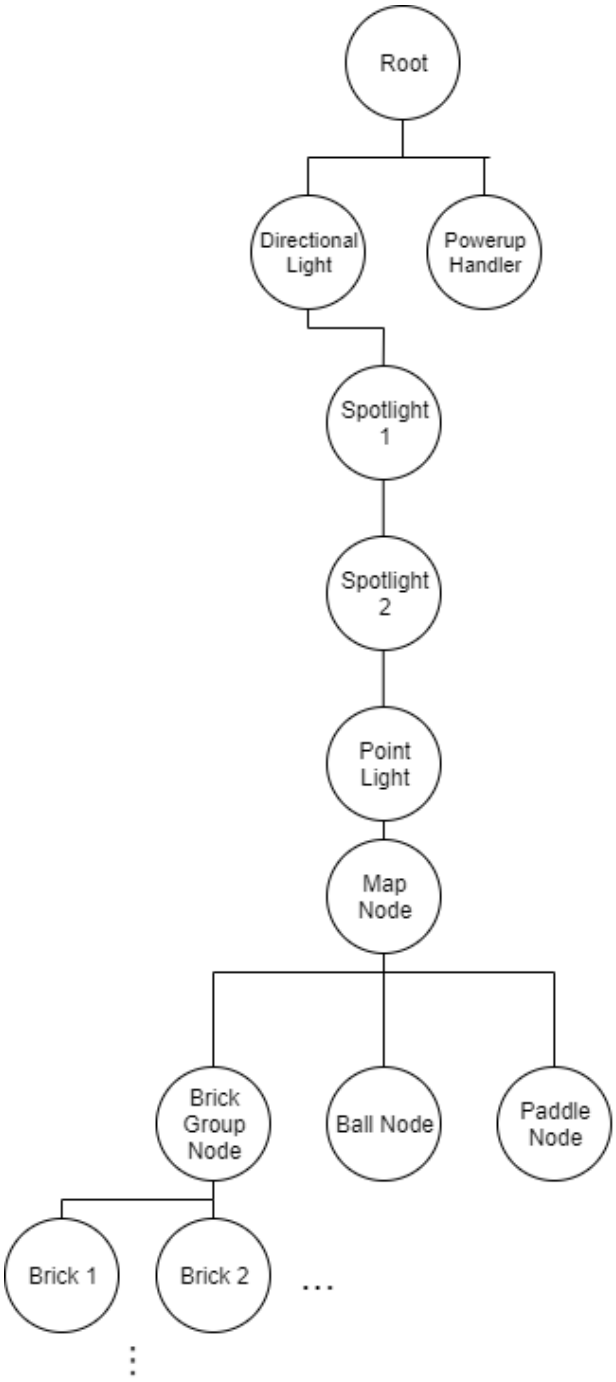

Figure 4.2: Scene graph for the game

Chapter 5

Task 5

This chapter goes over how the specified powerups were implemented.

5.1 Powerup Handler

In the *powerup.js* file, the powerupHandler object can be found. This object acts as an interface to the powerup system in the game. It is responsible for creating the powerups (through *powerupHandler.handlePowerup()*) as well as maintaining them for the duration they are active. When the ball intersects a block that has a powerup, the powerupHandler is called to create a drop with that powerup. If the drop intersects the paddle, it is called again, this time to actually activate the powerup or to reset it if it is already active.

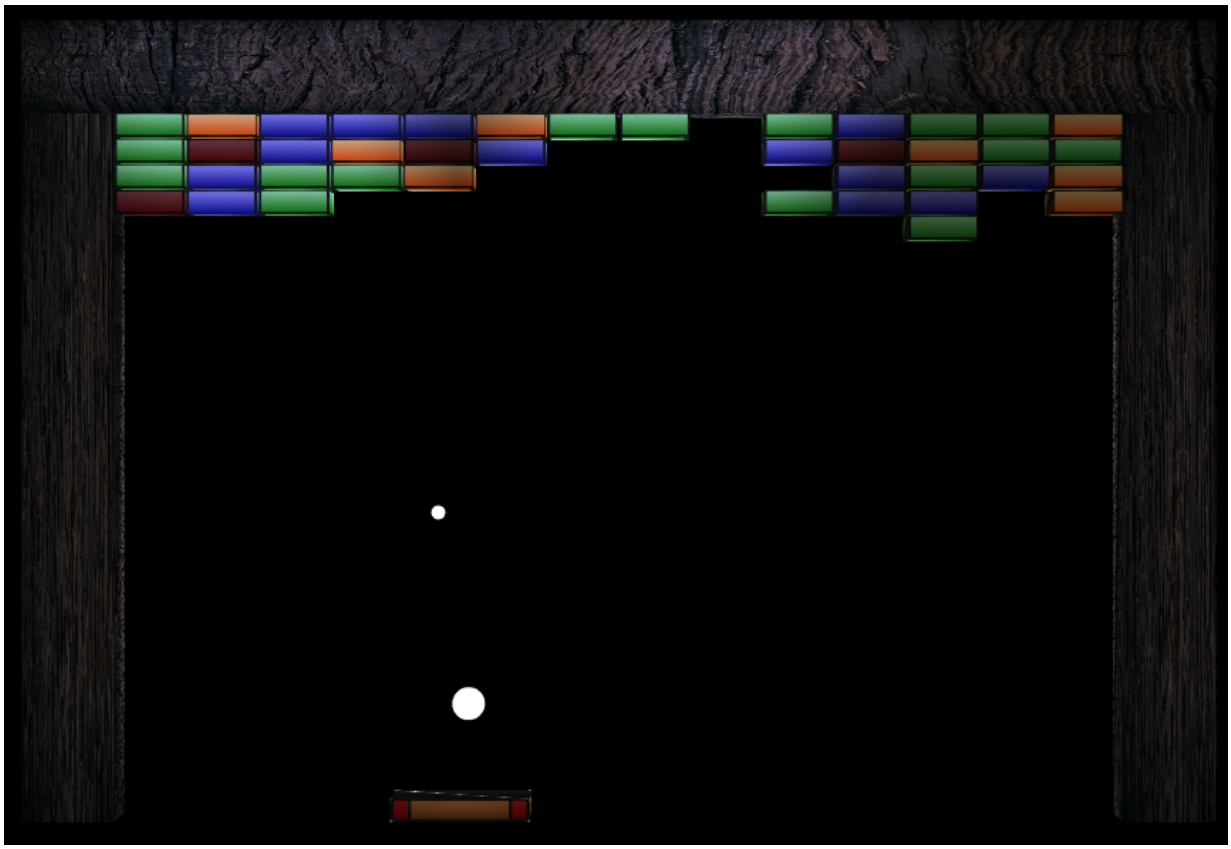

Figure 5.1: A powerup drop (the small ball)

5.2 Powerup Overview

5.2.1 Generating Powerups

When generating the bricks, each brick has a 10% chance of containing a powerup. If this is the case, its powerup will be chosen randomly from the available powerups.

5.2.2 Break Bricks

The ball node has a flag called *break*. When the powerupHandler activates the break bricks powerup, this flag is set to true, meaning that the ball will not calculate the reflection and will break all the blocks in its path unless it collides with a border block or the paddle. Lasts 5 seconds as it's a bit powerful.

5.2.3 Vaus Expansion

When this power is activated, the original paddle is placed somewhere arbitrary and a bigger paddle along with a new physics body are created that replace the nodeObject in the paddle node. After 10 seconds, the new paddle is deleted and replaced with the old one.

Figure 5.2: Expansion powerup

5.2.4 Splitting Ball

When this power is activated, the powerupHandler spawns a new ball next to the original ball. This ball is identical to the original ball except that it deletes itself after 10 seconds. Furthermore, if a ball goes out of the map it doesn't decrease any lives.

Figure 5.3: Split ball powerup

5.2.5 Half Speed

When this is activated, the powerupHandler updates the speed of the ball to half of it. It also stores the original speed. After 15 seconds, the ball node is updated with its original speed.

5.2.6 Sticky Ball

The ball model has a variable called *stick* associated with it. When this powerup is activated, the variable gets set to true for 10 seconds. During these 10 seconds, collision with the paddle stops the ball from reflecting and makes it stick to the paddle, as is the case in the beginning of the game. In order to shoot the ball, the **spacebar** must be pressed again. The longer it is held the faster the ball will go.

Figure 5.4: Sticky ball powerup (mid-game)

5.2.7 Laser Cannon

When this powerup is activated, the paddle emits up to a total of 5 lasers. These lasers keep going upwards until they intersect with a block. If it is not a border block, it deactivates the block and decrements the block count. This makes use of the physicsBody box-box intersection test.

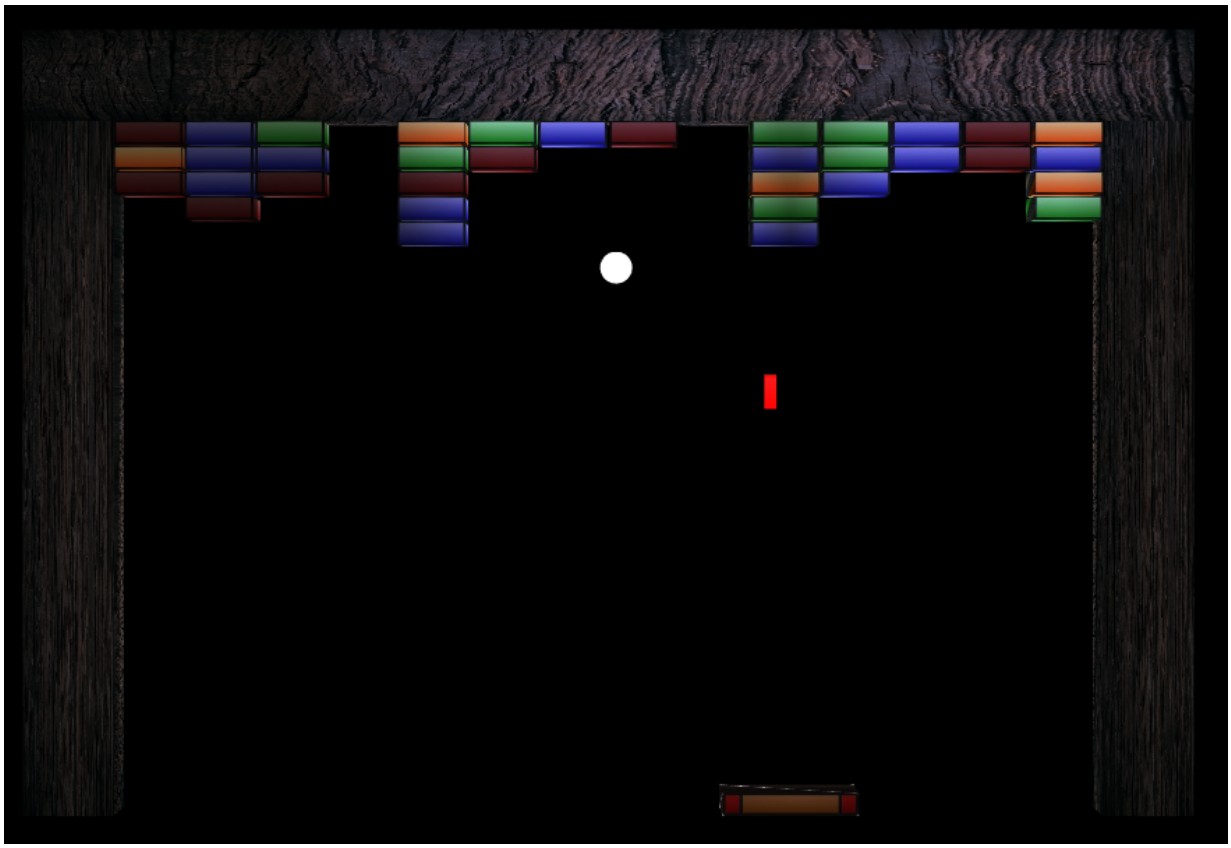

Figure 5.5: Laser Powerup

Chapter 6

Known Bugs

This section contains a list of known bugs.

- Sometimes lasers go through border blocks and brick blocks. Unsure why this happens.
- The paddle goes through the left and right border blocks. This could have been solved by either testing collision with the border blocks or simply limiting the x values to the length of the map. Due to time constraints and fear of breaking the code, it was not fixed.
- Split balls sometimes bounce back in the map.
- On Edge, the paddle sometimes starts to move in one direction with no input.
- The sticky ball powerup always resets the ball to the centre rather than where it hit.
- Sometimes the ball reflects weirdly off of blocks.
- Nothing happens upon breaking all the blocks.

Chapter 7

File Structure

The logic of the game is split into a number of files as follows:

- index.html: Contains the shader code.
- brick.js: Contains the code for generating a brick model.
- light.js: Contains the code for generating a light model.
- model.js: Contains the code for handling model objects.
- matrix.js: Contains the mathematical functions used in the game.
- material.js: Contains the code for the material object.
- powerup.js: Contains the code for handling the generating and maintaining powerups.
- texture.js: Has the textures encoded in base64 format.
- scene.js: The code for creating the scene graph as well as the node objects.
- PhysicsBody.js: The code responsible for the physics system in the game.
- gameMap.js: Responsible for generating the entire map, including loading textures and setting up lights.
- render.js: The main render loop. Also contains a number of important initializations as well as the logic for handling keyboard input.

Chapter 8

Conclusion

This assignment made use of many important concepts such as scene graphs in order to organise the scene and decrease complexity. Moreover, it showcased a custom built (but very limited) physics engine and how to make use of multiple lights in the scene. It isn't as optimised as the writer hoped, but due to time constraints and the small size of the game, the performance penalty was seen as acceptable.