G Boots

A Real-Time 3D Puzzle Video Game

Graphics Senior Project

Zachary Glazer

Project Advisor: Zoë Wood

Spring 2015

Abstract

G Boots is a real-time 3D puzzle video game. It uses OpenGL, using GLSL for shaders, in order to implement cross platform support of advanced rendering and shading techniques. A level editing system was implemented so that anyone can make or edit levels that can then be played in the game. At the end of two quarters of development by a single developer, one polished proof of concept level was created using the level editing system in order to show the feasibility of the project.

Introduction

The video game industry is a multi-billion dollar industry. It used to be that only large companies could afford to make video games due to the high cost of making and distributing the games. But now, with the ability to distribute games over the internet with services like Steam and graphics technologies becoming more widely known, it has become possible for individual developers to make and distribute games.

G Boots was made with this in mind. It was made by one developer in only two quarters (6 months) while taking multiple other classes at California Polytechnic State University, San Luis Obispo. The purpose of this project was to build a fun, real-time 3D puzzle video game for desktop machines. It was built using OpenGL 2.1, GLSL 120, GLFW 3.0.4, FreeType 2.5.5, Bullet3 2.83, and SOIL (Simple OpenGL Image Library). The main mechanic of the game is the player’s ability to control how gravity affects them and other object. Due to the way a player uses this ability, the game was made with the intention of playing it using a DuelShock3 controller for user input and not a keyboard.

Related Work

G Boots was designed with a few previous video games in mind. The following titles influenced both the gameplay and design of G Boots.

Portal

Portal is a 3D puzzle video game made by Valve Corporation. G Boots’ design is heavily influence by the Portal series. Most notably the laboratory setting of game. Portal takes place in a series of test chambers where the player must solve a puzzle to advance to the next chamber. G Boots is designed in the same way.

Figure 1 – A Test Chamber in Portal

Little Big Planet

The level making system in G Boots was inspired by Little Big Planet. The player is able to create their own levels that they and others can then play.

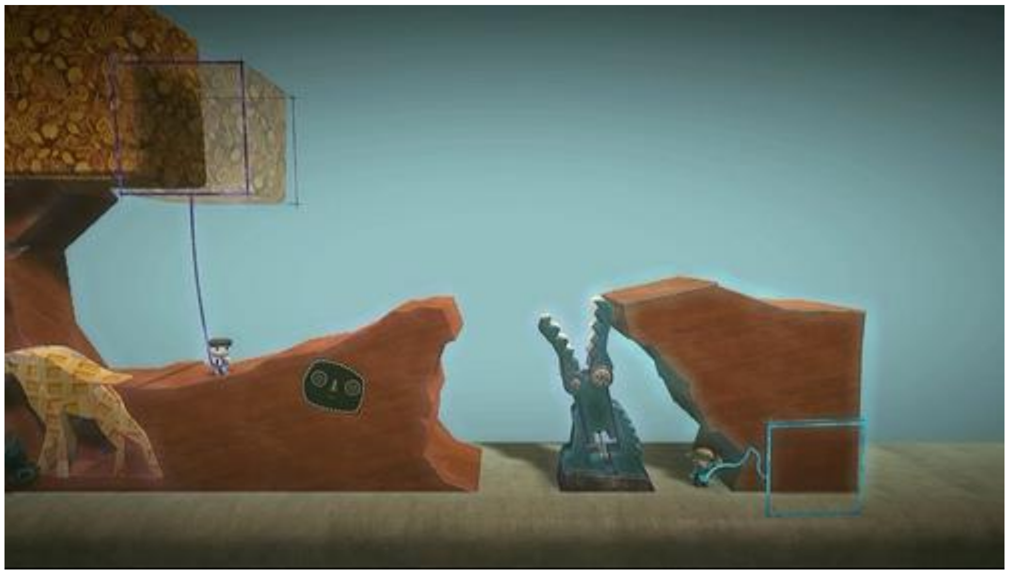


Figure 2 – Little Big Planet’s Level Making System

Project Overview

Story

The player has been hired by an unknown entity to test out a newly invented pair of boots that alter the wearer's molecular structure, giving him/her the ability to control gravity. As time progresses however, the player realized that the entity that hired them cannot be trusted, and must thwart its nefarious plans.

Genre and Setting

G Boots is a first person 3D puzzle platforming game. The game takes place in a laboratory at an unspecified place on earth. Being in a laboratory, the environment is very clean, with only the instruments necessary to complete the tests being present.

Game Mechanics

The core game mechanic of G Boots is its unique physics. The player has a pair of boots that allows them to change the direction of gravity affecting them. Using this core ability, the player must walk on the walls and fly through the air to solve various puzzles presented to them. The player also has the ability to change the direction of gravity affecting other objects. After picking an object up, the player can choose which direction they want gravity to affect that object.

Aside from this main mechanic, there are other smaller mechanics that are common to puzzle games. There are buttons that the player must press with their hands in order to activate other object like doors. There are also larger buttons on the floor and walls that the player must either stand on or put something on to activate another object. There are also lasers and spikes that, if touched, will harm the player. Lastly, there are areas of acid that will harm the player if they enter them.

Technical Specifics

The following table lists the technologies used in G Boots.

Technology	**Use**
OpenGL/GLSL	Multi-platform API for rendering 3D vector graphics.
GLM	OpenGL Mathematics library mainly used for calculating camera transforms
GLFW	Multi-platform library for creating windows with OpenGL contexts and receiving input and events.
FreeType2	Multi-platform library for rendering different fonts.
SOIL	A Simple OpenGL Image Library used to load image files directly into texture objects.
Bullet3	A physics library used for collision detection and realistic movement of objects.

Table 1 – Technologies Used

Project Technologies

The following table shows all technologies that were planned to be implemented in G Boots and what the current state of that technology is. Following the table, some of the more interesting technologies are explained.

Technology	**State**
3D collision detection	Implemented
Realistic 3D movement of objects	Implemented
Phong Lighting	Implemented
Texturing of Objects	Implemented
Shader Library	Implemented (currently has 8 shaders)
Texture Library	Implemented
Shadows	Implemented
Level Editing System	Implemented
Player Control/Movement with Unique physics	Implemented
View Frustum Culling	Not Implemented
Main Menu/Pause Menu	Implemented
Octree	Implemented

Table 2 – Technologies Implemented

Unique Physics

Making easy to use user controls that worked properly with the unique physics of the game was challenging. The first task was to make the player able to change the direction of gravity affecting them. The actual act of changing the gravity on an object was made easy thanks to Bullet Physics. Any Rigid Body, what Bullet Physics represents a physical object as, has a **setGravity** method that takes a three dimensional vector representing the pull of gravity in the x, y, and z direction and sets its gravity to that vector.

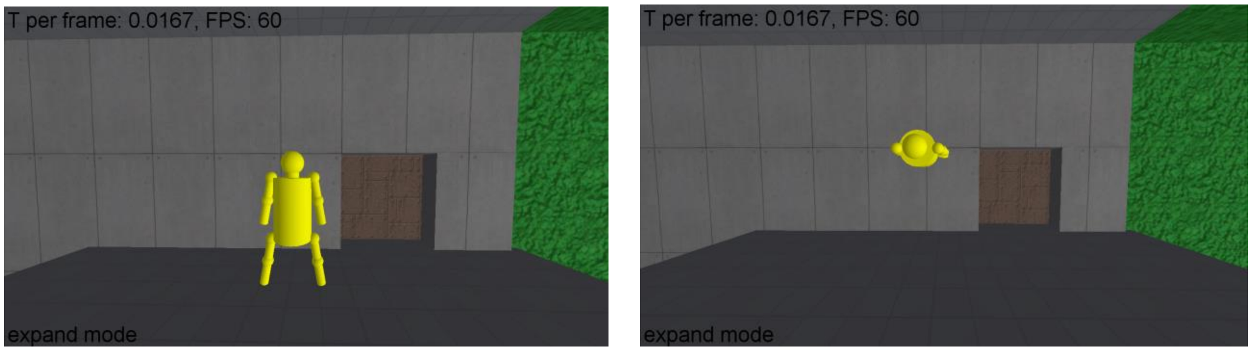


Figure 3 – Player model before changing gravity (left) and after changing gravity to be on wall (right)

However, making easy to use user controls for this action proved to be difficult. At first, the player was only given six buttons to change gravity in the 6 cardinal directions. This proved to be extremely difficult to use however. Due to the nature of the game, it was nearly impossible to correctly change the direction of gravity to the direction the user wanted. Because of this, special controls were made that allowed the player to change gravity based on their current orientation in space. These controls use the current direction of gravity and the current direction the player is facing to correctly calculate what direction gravity should be changed to. For example, assume the gravity on the player is currently in the negative X direction and the player is facing the negative Y direction. If the player were to tell the game to change gravity to be behind them, the controls would determine that this is the positive Y direction and change gravity accordingly. If the player wanted to change gravity to their right instead, the controls would change gravity to the positive Z direction.

There was still one problem however, the player isn't always facing directly at one of the axes. Therefore, it was necessary to calculate a relative forward direction based on the actual direction the player was facing. This was the hardest part of task one and was accomplished using trigonometry. The algorithm to do this uses the direction the player is looking and the current up vector of the player (explained below) to determine what cardinal direction the player is facing. The entire algorithm is shown in Code Snippet 1 bellow.

The second task was to change the camera's view depending on what direction gravity was affecting the player. This was accomplished by altering the view matrix that was passed to the shaders at render time. This was also made easy thanks to GLM's helpful **lookAt** method that calculates a view matrix given the camera position, the point where the camera is looking, and an up vector. Therefore, if the up vector passed to **lookAt** was changed to always be pointing straight above the player, the player's view would be fixed. To do this, the up vector was calculated by taking the normalized negative of the gravity acting on the player. For example, if the player changes gravity acting on them to be in the positive X direction, which is represented by a gravity vector of (9.8, 0, 0), then the up vector would be (-1, 0, 0). So by changing the cameras up vector in this way whenever the player changed gravity, the player's view was fixed.

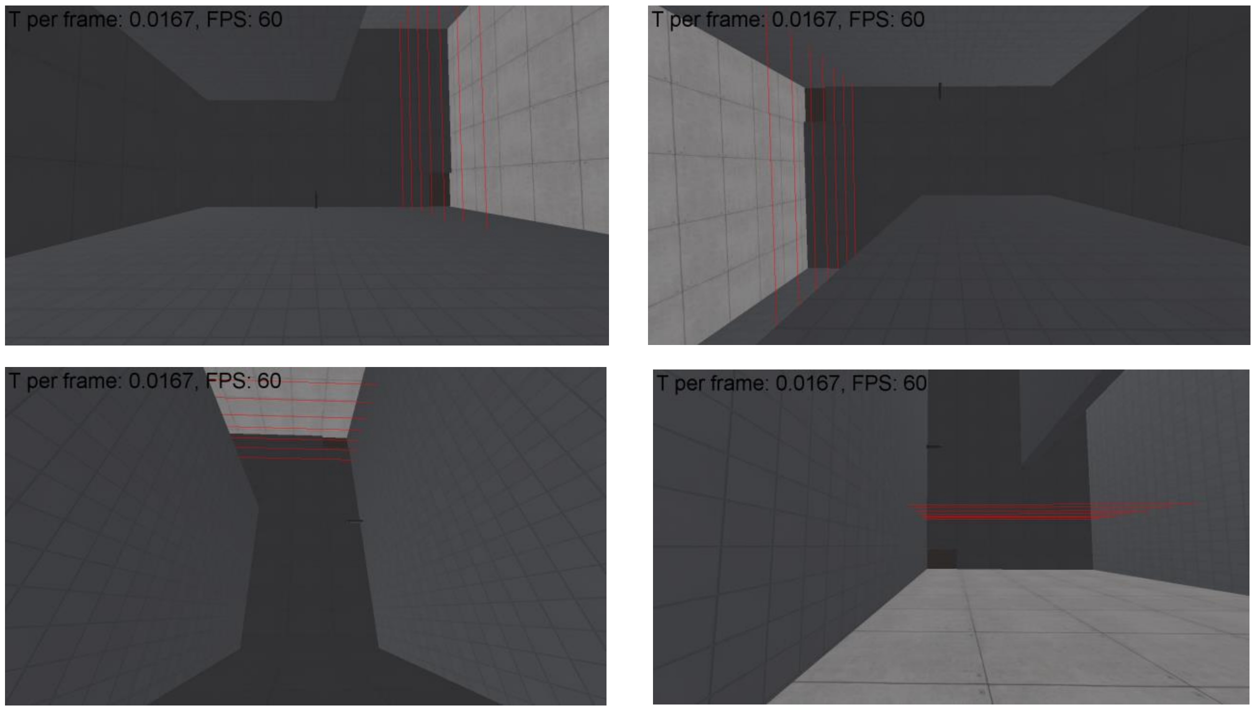

T per frame: 0.0167, FPS: 60
T per frame: 0.0167, FPS: 60
T per frame: 0.0167, FPS: 60
T per frame: 0.0167, FPS: 60

Figure 4 – The same view, but from different angles as the player changes gravity

```
glm::vec3 TestMap::getRelativeForwardDir() {
    glm::vec3 gaze = camera.getLookAt() - camera.getEye();
    //upVector is a global that has the current up vector of the player
    if (upVector.x != 0) {
        gaze.x = 0;
        gaze = glm::normalize(gaze);

        double cosTheta = gaze.z;
        double sinTheta = gaze.y;
        double theta = acos(cosTheta) * 180.0 / M_PI;
        if (sinTheta < 0) {
            theta = 360 - theta;
        }
        if (theta >= 45 && theta < 135) {
            return glm::vec3(0, 1, 0);
        } else if (theta >= 135 && theta < 225) {
            return glm::vec3(0, 0, -1);
        }  else if (theta >= 225 && theta < 315) {
            return glm::vec3(0, -1, 0);
        }  else {
            return glm::vec3(0, 0, 1);
        }
    } else if (upVector.y != 0) {
        gaze.y = 0;
        gaze = glm::normalize(gaze);
        double cosTheta = gaze.x;
        double sinTheta = -gaze.z;
        double theta = acos(cosTheta) * 180.0 / M_PI;
        if (sinTheta < 0) {
            theta = 360 - theta;
        }
        if (theta >= 45 && theta < 135) {
            return glm::vec3(0, 0, -1);
        } else if (theta >= 135 && theta < 225) {
            return glm::vec3(-1, 0, 0);
        }  else if (theta >= 225 && theta < 315) {
            return glm::vec3(0, 0, 1);
        }  else {
            return glm::vec3(1, 0, 0);
        }
    } else if (upVector.z != 0) {
        gaze.z = 0;
        gaze = glm::normalize(gaze);

        double cosTheta = gaze.x;
        double sinTheta = gaze.y;
        double theta = acos(cosTheta) * 180.0 / M_PI;
        if (sinTheta < 0) {
            theta = 360 - theta;
        }
        if (theta >= 45 && theta < 135) {
            return glm::vec3(0, 1, 0);
        } else if (theta >= 135 && theta < 225) {
            return glm::vec3(-1, 0, 0);
        }  else if (theta >= 225 && theta < 315) {
            return glm::vec3(0, -1, 0);
        }  else {
            return glm::vec3(1, 0, 0);
        }
    }
    return glm::vec3(0,0,0);
}
```

Code Snippet 1

The third and final task was to allow the player to actually move around on whatever they were standing on. There were two variable needed for this, the forward direction the player could move, and the sideways (strafing) direction the player could move. The forward direction was simply the direction the player was looking without the current up vector component. So, if the player was looking in the direction of (.21, .21. 21) and the current up vector was (0, 1, 0) then the forward direction would be (.21, 0, .21). The strafing direction is then the cross product of the forward direction and the up vector. Once these two values were calculated, if the player told the game to move in a certain direction, the game could easily move the character model in that direction.

Level Editing System

In order to create levels, a level editing system had to be made. This level editing system needed to be simple enough for anyone to use so that players could create their own levels, but also robust enough so that a sufficiently complex level could be made. To attain the goal of being simple enough for anyone to use, it was decided that a person should be able to create an entire level from scratch using only a PlayStation 3 controller. For a person to be able to make an entire level from scratch, three main editing functionalities were made for every object. A full list of all editing mode actions can be found at the bottom of the section in Table 3.

The first editing functionality was to simply move an object around in space. The actual game already had functionality to pick up certain objects, so this functionality was extended while in editing level mode so that the user could pick up any object, including walls and other object that could not be moved in the actual game. To do this with Bullet Physics however, every object had to be set with the activation state of DISABLE_DEACTIVATION so that the object would not become deactivated. Furthermore, all objects that would normally be affected by gravity, such as the player model or boxes, had to be given a mass of zero so that no forces would act on them. Simply moving an object proved to be difficult in editing mode however. Because of the speed the user moves, it was hard to make small adjustments to an objects position. To combat this, the user can hold the left bumper to slow down any action taken. On the other end of the spectrum, moving large distances was very time consuming, so the user can hold right bumper to speed up any action taken.

The second common editing functionality was the need to change the size of objects. Every object besides buttons and the player model are able to change size. User input for this functionality is simple. By holding a D-pad button and pushing either up or down on the right thumb stick, the game would be told to either expand or contract an object. For example, if while holding a wall the user holds the up D-pad button and pushes forward on the right thumb stick, the wall will should start to expand in the Z direction. While the user input for this action is simple, what the actual game needs to do is not.

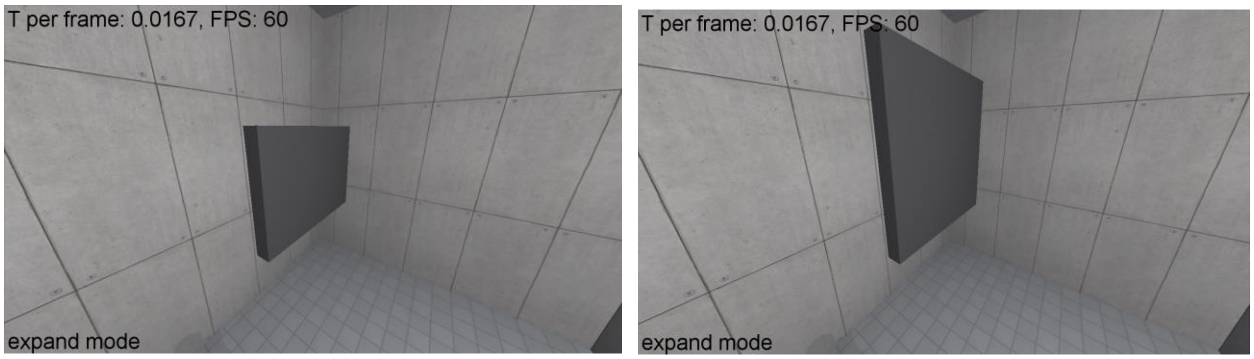
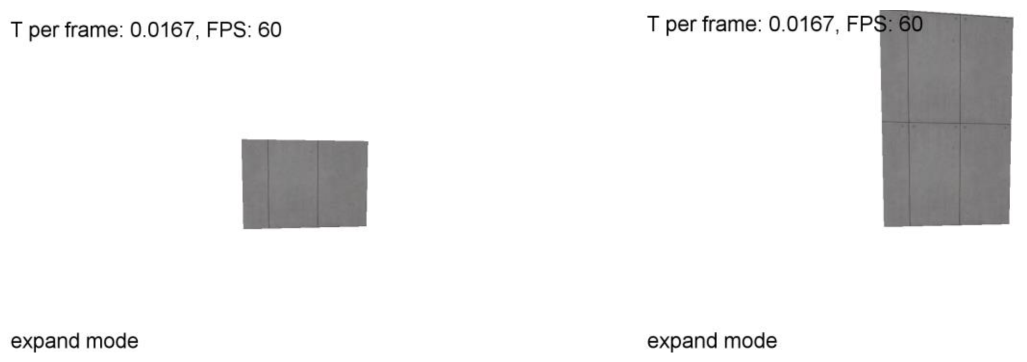


Figure 5 – An untextured wall before resizing (left) and after resizing (right)

In order to update an objects size on the screen and because an object wasn't simply being scaled, the object's mesh had to be changed. Because of this, it was decided to only use geometric primitives for expandable objects so that their new meshes could be computed easily. For example, walls and boxes are rectangular prisms, spikes are cones, and lasers are lines. This way, when creating new meshes, the primitives can be defined with only a few variables (e.g. width, height, depth for cubes).

Due to this, it became relatively simple to update the meshes of object. The steps taken to do this are to first calculate the new dimensions of the primitive object based on user input. Second, delete all the buffers associated with a mesh by calling **glDeleteBuffers** on them. Third, create new meshes using the new data. For this, the vertices, normals, and texture coordinates all had to be recomputed for the new shape. This was simple to do for objects like cubes that only have eight vertices, but significantly harder for objects like cones and spheres. To do this, a geometry creator made by Ian Dunn, a student at California Polytechnic State University, San Luis Obispo, was used. After this data is computed, new buffers are made with **glGenBuffers**, bound with **glBindBuffer**, and the data is written to them with **glBufferData**.

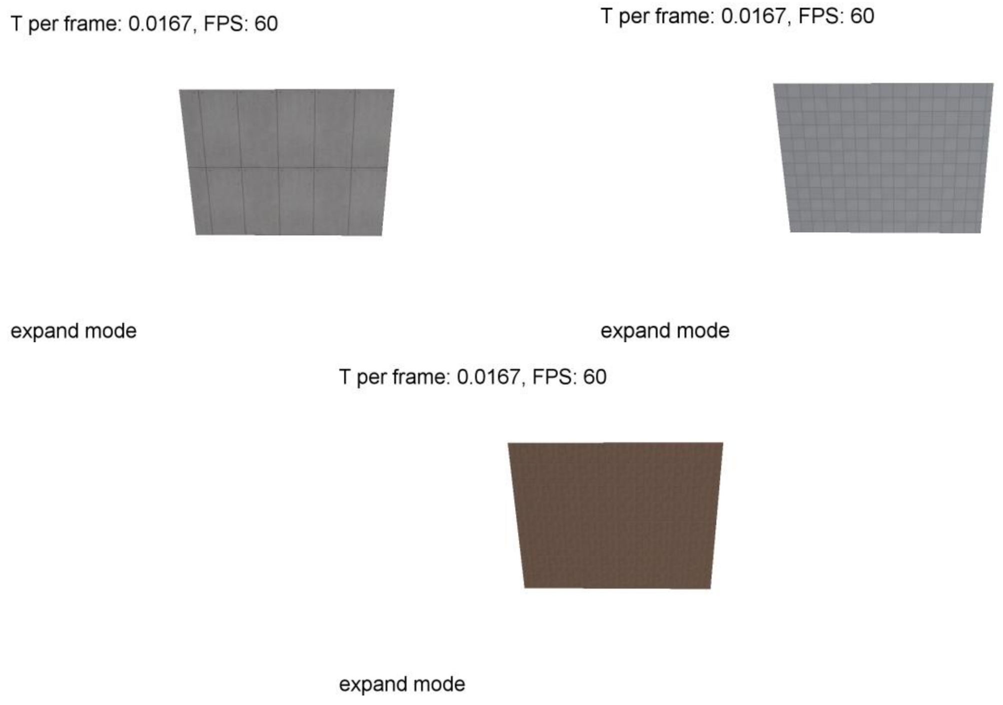
The third and last editing functionality is to change the texture on an object. This is done much the same way as changing the meshes above. The main difference is that vertices and normals do not need to be updated. In fact, some times the texture coordinates do not even need to be updated. For example, objects like boxes do not need to update their texture coordinates when the texture is switched because, with boxes, it is assumed that the entire texture is used on each side of the box without repeating, so the texture coordinates always stay the same. With walls however, the texture coordinates do need to be updated. This is because, with walls, it is assumed that the texture does not fit exactly one on the wall. Each texture has a width and a height which represents the world width and height that the texture would ideally have (this is "ideal" because boxes ignore this value). Because of this, the texture coordinates for walls are recomputed whenever the texture changes or the wall is expanded so that the texture properly repeats on the wall. The exact process for this is shown in Code Snippet 2. Because of this, the user can select any texture for any object that allows texturing and have the texture be scaled properly.



```
T per frame: 0.0167, FPS: 60
expand mode

T per frame: 0.0167, FPS: 60
expand mode
```

Figure 6 – Wall texture before resizing (left) and wall texture after resizing (right)



```
T per frame: 0.0167, FPS: 60
expand mode

T per frame: 0.0167, FPS: 60
expand mode

T per frame: 0.0167, FPS: 60
expand mode
```

Figure 7 – Same wall with three different textures

One final editing mode functionality that was essential to have was a way to save and load levels. In order to do this, a simple file reader and writer was made. The format of the level files is simple. Each line of the file is a different object. The first word on each line determines what object that line represents. The rest of the line tells the game specifics about that object. For example, the following would be a valid entry for a wall.

Wall: origin(3, 1, 2) extents(5, 7, .5) textureName(wallTex1) rotAngle(0) rotAxis(0, 0, 0) material(aColor(.1, .1, .1), dColor(.3, .3, .3), sColor(.03, .03, .03), shine(1))

This tells the program where the origin of the wall is, the width, depth and height of the wall, the name of the texture to apply to the wall, how much to rotate the wall (zero here) around what axis, and what material the wall is made out of.

```
std::vector<float> TextureCoords;
float x = width / curTexture->texWidth;
float y = height / curTexture->texHeight;
float lrx = depth / curTexture->texWidth;
float tby = depth / curTexture->texHeight;

float const CubeTextureCoordinates[] =
{
   x,  0, // back face verts [0-3]
   x,  y,
   0,  y,
   0,  0,

   0,  0, // front face verts [4-7]
   0,  y,
   x,  y,
   x,  0,

   lrx,  0, // left face verts [8-11]
   0,  0,
   0,  y,
   lrx,  y,

   0,  0, // right face verts [12-15]
   lrx,  0,
   lrx,  y,
   0,  y,

   0,  0, // top face verts [16-19]
   x,  0,
   x,  tby,
   0,  tby,

   0,  0, // bottom face verts [20-23]
   x,  0,
   x,  tby,
   0,  tby
};
TextureCoords = std::vector<float>(CubeTextureCoordinates, CubeTextureCoordinates + 24 * 2);

Meshes[0]->updateTextureCoords(TextureCoords);
```

Code Snippet 2

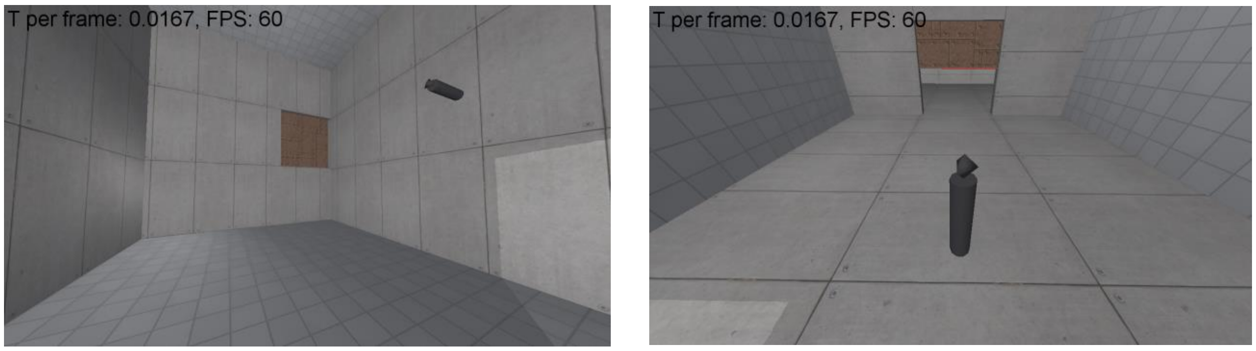
Functionality	**Action**
Select/Release object	Press circle while looking at object
Move object	Move left and right thumb sticks
Move object closer or farther away	Hold X and push or pull left thumb stick
Rotate object	Hold right on the D-pad and press down the right thumb stick
Switch modes	Press Square
Move Object along X Axis	While in “move mode” hold down left or right on the D-pad and push right thumb stick forward or backward
Move Object along Y Axis	While in “move mode” hold down up or down on the D-pad and push right thumb stick forward or backward
Move Object along Z Axis	While in “move mode” hold down up and left or down and right on the D-pad and push right thumb stick forward or backward
Expand Object in X direction	While in “expand mode” hold down left or right on the D-pad and push right thumb stick forward or backward
Expand Object in Y direction	While in “expand mode” hold down up or down on the D-pad and push right thumb stick forward or backward
Expand Object in Z direction	While in “expand mode” hold down up and left or down and right on the D-pad and push right thumb stick forward or backward

Change laser max length	While in “expand mode” hold down any D-pad button and push right thumb stick forward or backward
Delete Object	Press center PlayStation button
Add/Change Object	Press Select button
Change Texture	Hold down X button and press right D-pad button
Set saved notifier (currently only buttons)	With object selected, press Triangle
Set object to listen to saved notifier (currently only doors)	With object select, press triangle
Save Level	Press start
Set door open state to current position	Press left stick

Table 3 – Editing Mode Actions

Results

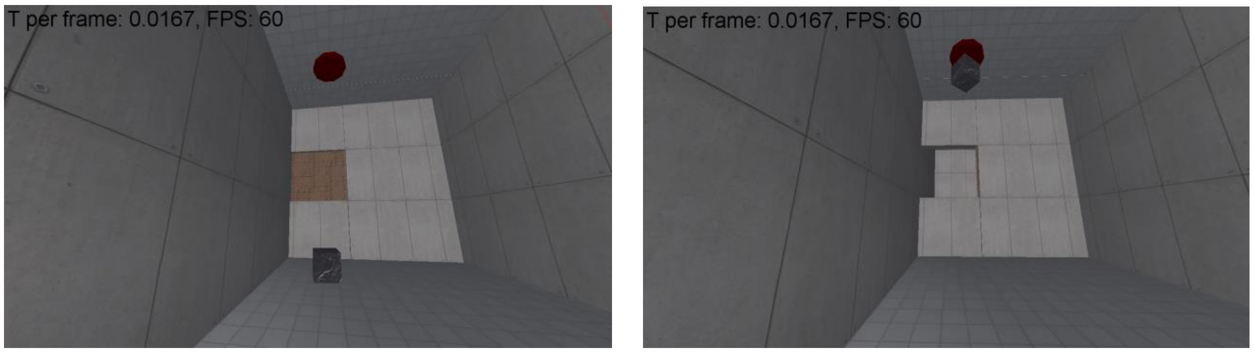
At the end of two quarters of work, G Boots has turned into a fully playable, and makeable, game/proof of concept. There is currently one fully complete test level that demonstrates all working functionality and gives examples of some puzzles that could be made. After receiving positive feedback from a non-formal play testers, it is safe to say that the goal of creating a fun, graphics intensive puzzle game has been achieved. Below are some screenshots which show the capabilities of the game.



```
T per frame: 0.0167, FPS: 60
```

```
T per frame: 0.0167, FPS: 60
```

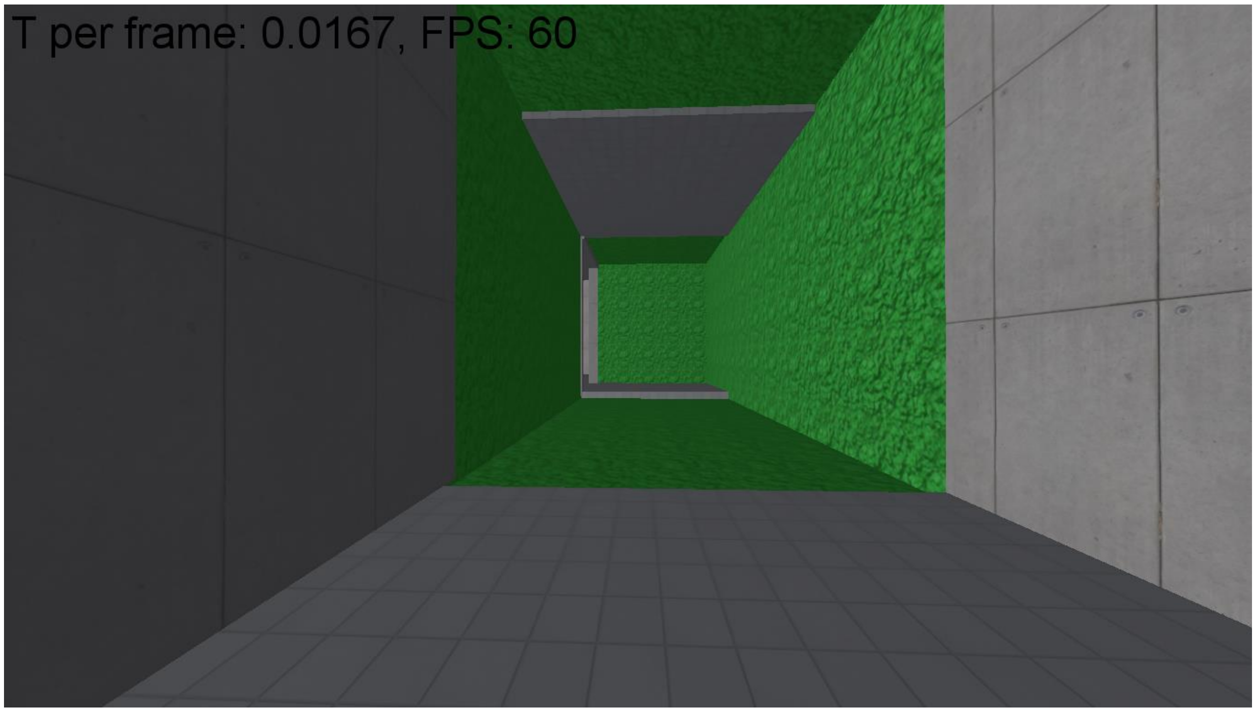
Figure 8 – Player must change gravity to reach button that opens door



```
T per frame: 0.0167, FPS: 60
```

```
T per frame: 0.0167, FPS: 60
```

Figure 9 – Player must change gravity affecting box to make it fall onto button



```
T per frame: 0.0167, FPS: 60
```

Figure 10 – Player must change gravity to avoid acid (green areas)

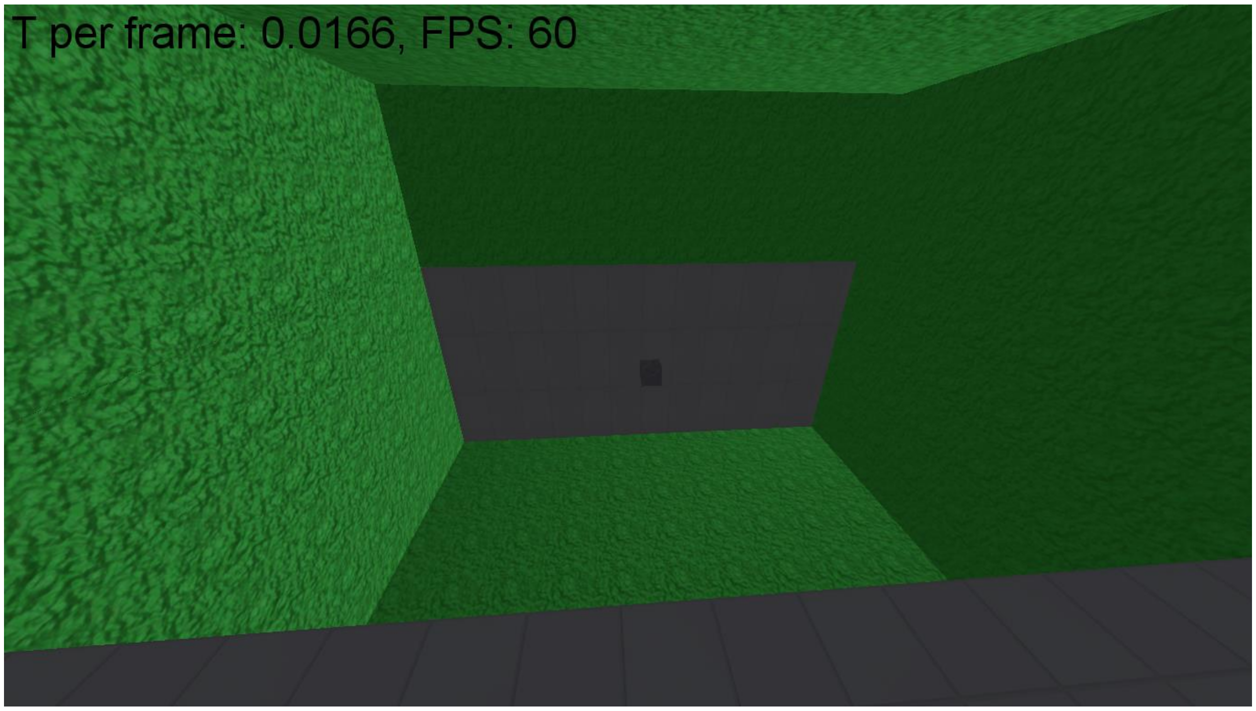


Figure 11 – Player must change gravity to reach box

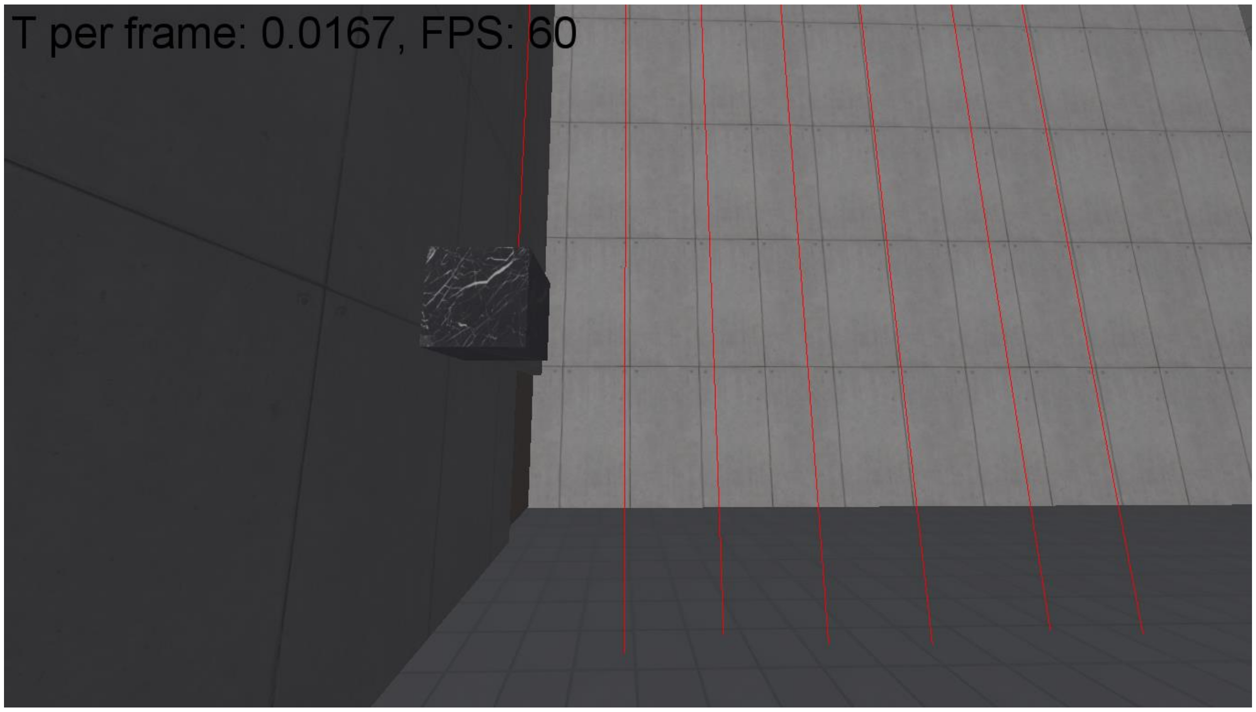


Figure 12 – Player can use boxes to block lasers

Future Work

While much was accomplished during the two quarter development of G Boots, there is still much that could be improved. One main future goal would be to add more levels for a story mode and add sound so that a story could be told. It would be nice to add View Frustum Culling to make general rendering faster. This should be relatively easy because of the implemented Octree.

Other smaller future goals would be to add smoothness to the change of gravity. Currently the view simply snaps to the new view direction. Having moving platforms would also be nice and adding other objects and obstacles for puzzles is always welcome.

Bibliography

Valve Corporation. (2007). [Portal video game]. Bellevue, Washington: Valve Corporation.

Media Molecule. (2008). [Little Big Planet video game]. Sony Computer Entertainment Europe.

" Tutorial 16 : Shadow mapping” opengl-tutorial. n.p., September 7, 2013, Web. March 5, 2015. < http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/>

thecplusplusguy. YouTube. YouTube, September 28, 2012, Web. January 17, 2015 < https://www.youtube.com/watch?v=wbu5MdsFYko&list=PLj4I-yBchUboXrXmPh6iepqw2lbAlq7gt/>

Bullet3. Bullet Collision Detection & Physics Library. Doxygen, n.d. Web. January 10, 2015 < http://bulletphysics.org/Bullet/BulletFull/index.html/>