ATTRACT

3D Real Time Puzzle-Platformer Game

Mark McKinney
California Polytechnic State University
Winter – Spring 2017
Advisor: Zoë Wood

Abstract

ATTRACT is a 3D video game developed in C++ which utilizes libraries such as OpenGL, GLSL, GLM, FMOD, and Bullet. The player traverses buildings by jumping and using a magnet gun to attract to and repel from magnetic objects. Aimed at being a graphically interesting and fun game, ATTRACT incorporates many advanced realtime graphics technologies and algorithms including normal mapping, an interactive level editor, and fog; these specific technologies will be analyzed in depth, but the game includes many others. Due to the limited timespan of the project, the game still has minor bugs and could use some graphical polish and a more interactive story, but the current version is fully playable, graphically appealing, and fun with the ability to add more levels easily.

Introduction

The video game industry has grown immensely since the Nintendo Entertainment System helped revitalize the nearly-destroyed industry in 1985. According to the Entertainment Software Association and the NPD Group, the U.S. video game industry generated more than $30.4 billion in revenue in 2016 [14]. As a result of its size, jobs in the video game market are typically in high demand by software developers. The popularity of the industry is beneficial for the field of computer science, as game design is a major contributor to the evolution of new and existing graphical and computational technologies. As such, building a video game is an excellent avenue for expanding and showcasing the programming and teamwork skills needed as a software developer. In particular, creating a puzzle game is an interesting challenge, as the developer must have an apt understanding of level design, the game mechanics, and the interaction between the two in order to create effective puzzles. It allows for practice in fine-tuning simple mechanics that can be used in a variety of situations, as opposed to introducing several shallow and/or unintuitive gameplay features. Thus, the team decided to develop a puzzle game, with a focus on platforming to add more opportunities for creative level design.

Project Overview

ATTRACT is a 3D real-time puzzle platformer game which incorporates magnets as one of its core gameplay elements. The player must traverse the tops of skyscrapers through platforming, frequently using a magnet gun to attract to and repel from magnetic objects. This game showcases a variety of computer graphics technologies which were built on top of the OpenGL framework using C++ and GLSL. It also made use of the FMOD, GLM, and Bullet Physics libraries for audio, mathematics, and physics simulation, respectively.

While there are other platformer games which highly utilize physics for solving puzzles, ATTRACT employs a unique approach with its magnet mechanics. By using magnetic forces, innovative puzzles can be created by combining the attracting and repelling fun of magnets with different buildings and landscapes. The possibilities are endless for creating new and thrilling levels with different types of jumps and launches to navigate through cities.

At the start of the ATTRACT, the player is shown a cutscene of their space ship malfunctioning and losing critical ship parts. The ship crashes; then, the player awakens on top of a building on an unknown planet with nothing but their magnet gun. The ship parts must be recovered in order to take off again, so the player makes their way through the city to collect all of the missing ship parts. Once the final level is completed and all of the ship parts have been collected, the player can take off in the spaceship and head back home.

ATTRACT utilizes a slight difficulty curve. The game starts off with simple levels, in order to familiarize the player with the game mechanics of launching off of and attracting to magnetic objects. The levels become progressively harder and grow in size over time, testing the player's skills and providing a satisfying challenge to overcome.

Having the environment of a city with skyscrapers, the choice was made to go with a mostly realistic art style. The city is meant to look real with the buildings extremely high in the sky, so much so that they're above the clouds. The magnet gun was crafted to look like a futuristic but possible technological advancement with beam effects to accentuate it. While the buildings are all rectangular, they vary in size and are crafted in multiple levels to have irregularly shaped buildings by combining multiple rectangular buildings together.

ATTRACT started as a CPE 476 team project alongside Cody Rhoads and Darryl Vo. Kyle Lonczak also assisted with creating a model for the game. After CPE 476 ended, Cody and I continued to work on the game, developing several new technologies.

Related Work

Portal

Portal was a significant influence on ATTRACT, and it is not difficult to see the resemblance: both are first-person puzzle platformers where the player solves puzzles using a specialized tool [11]. In both games, simple mechanics are combined with creative level design in order to steadily build increasingly difficult challenges (Figure 1). ATTRACT is also similar to Portal in that the individual levels are completely isolated. While all the levels are set in the same city, there are no connections between previous levels and current ones. The setting for ATTRACT was originally going to be a laboratory like Portal, though it was changed early on to allow for more interesting scenery.

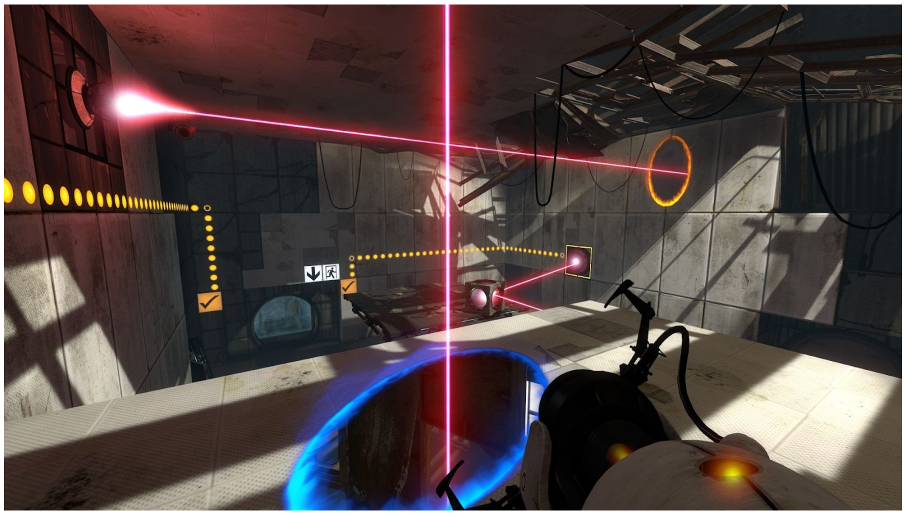


Figure 1 - An involved puzzle in Portal 2 that builds on simple mechanics like portals and lasers [6].

Legend of Zelda

The gameplay of ATTRACT was also heavily influenced by the Legend of Zelda (LoZ) series. Specifically, the idea of attracting to magnets in ATTRACT was based on LoZ's hookshot mechanic. In many LoZ games, such as Skyward Sword [7], players gain a hookshot item which allows them to "hook" to a designated wall and proceed to be launched towards the wall (Figure 2). The idea of attracting to magnets was conceived as it could replicate the thrill of flying through the air with the hookshot. It also had the benefit of being more flexible than the hookshot: instead of being limited to finding spots to attract to, players could also repel off of magnets, opening more options for puzzle solving and platforming challenges.

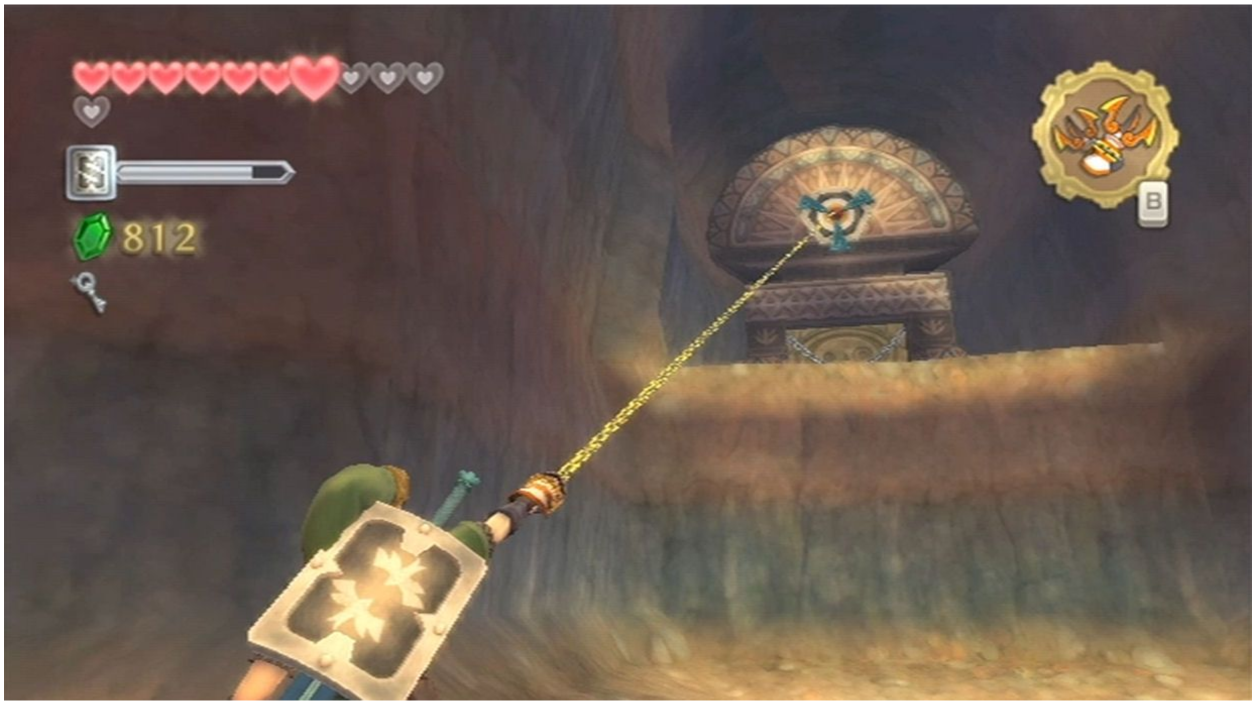


Figure 2 - Using the hookshot in Legend of Zelda Skyward Sword [13].

Mirror’s Edge

The environment was primarily influenced by the Mirror's Edge series [9]. In Mirror's Edge, players often find themselves free-running across the tops of skyscrapers (Figure 3). The idea of parkouring across skyscrapers lended itself well to platforming and could allow for interesting level design with different buildings. Mirror's Edge also assisted in game design decisions, specifically the color of the magnets. In Mirror's Edge, the environment is primarily white, but important objects that are interactive or mark the correct path are brightly colored to stand out. This helps the player make quick and informed choices. ATTRACT emulates this idea by having the magnets colored bright red, which stands out and helps prevent the player from getting confused on where to go.

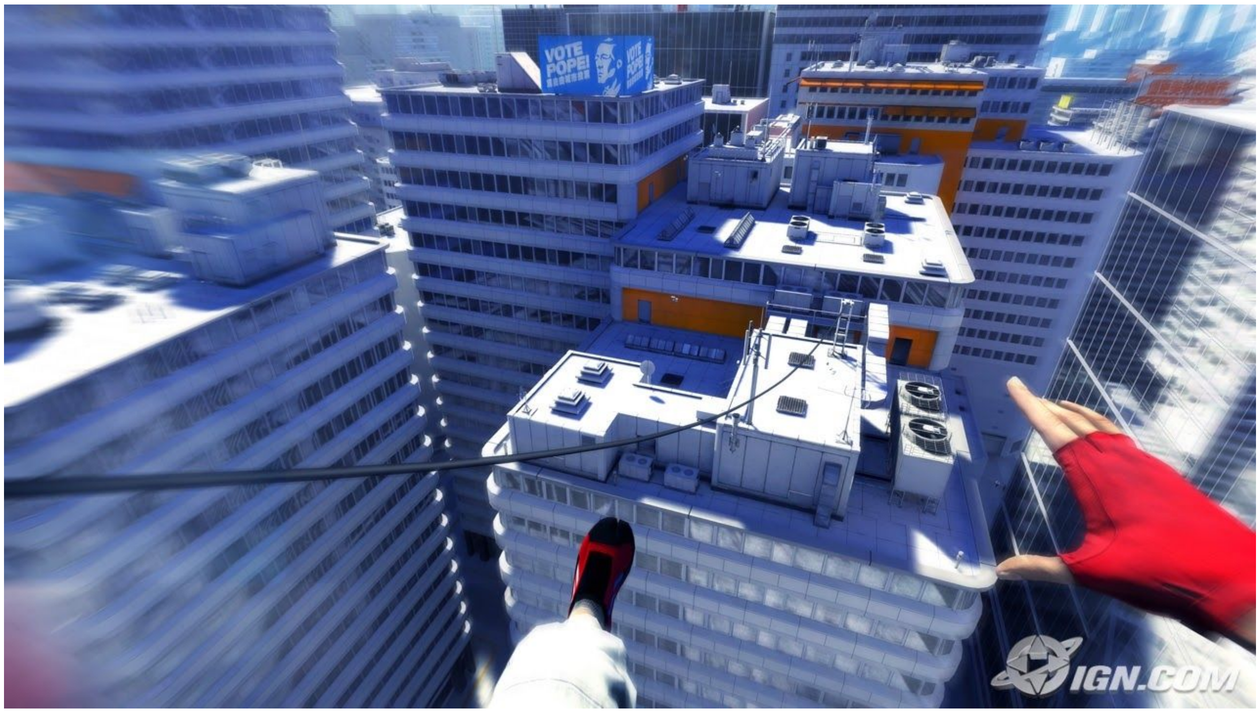


Figure 3 - Traversing the cityscape in Mirror’s Edge [4].

Technology List

The following is the list of major technologies added to ATTRACT over the course of two quarters.

Technology	**Author**
First-person camera	Mark McKinney
3D spatial data structure and collision detection	Cody Rhoads
Shadows	Cody Rhoads
View Frustum Culling	Darryl Vo
Normal Mapping	Mark McKinney
Bloom	Cody Rhoads
Level Editor	Mark McKinney
Particle Generation	Darryl Vo
Fog	Mark McKinney
Scripted fracturing	Cody Rhoads
Moving platforms	Mark McKinney
Audio	Darryl Vo
Magnet gun model	Kyle Lonczak

- Bullet Physics Engine was used for simulating physics within the game [1].
- GLM was used for vector and matrix mathematics [12].
- FMOD was used to manage in-game audio [5].

Algorithm Details

Normal Mapping

Almost every game applies 2D textures to objects in order to give them detail and hide the fact that all objects are in fact made of many polygons consisting of flat triangles. Much of the time, textures are applied to simple cubes like in ATTRACT. These flat faces can be given a little depth from putting a texture on them, but the lighting is not affected by whatever faked depth is in the texture. This is where normal mapping comes in.

Since the base of all geometry is a single flat triangle, the normal vector for the surface will be the same across the whole triangle and across the entire face of a cube. Instead of using the same normal for lighting computations which would be per-surface, a per-fragment normal is used which is different for each fragment. The result is an illusion of a complex surface even though the underlying surface is simpler, as shown in Figure 4 [3].

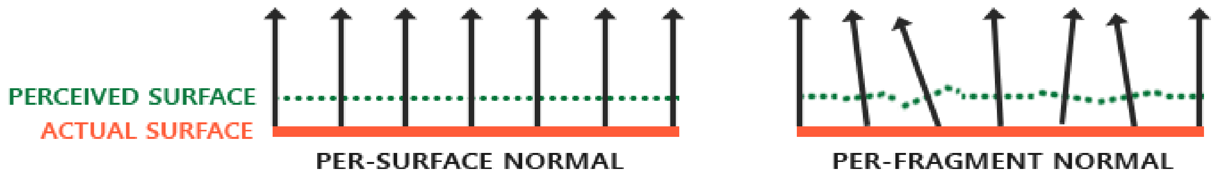


Figure 4 - Per-Surface vs Per-Fragment Normals [3]

Once the lighting uses the per-fragment normals, rendered scenes appear to have much more depth and detail without adding any expensive geometry. This means that, in order to save time rendering, higher polygon models can be converted into lower polygon versions and use normal mapping from from the normal data captured from the high polygon version.

In order to use normal mapping, the normal data is stored in a 2D texture. To achieve this, the r, g, and b components of the texture correspond to the normal vector's x, y, and z components. Since normal vectors range from [-1, 1], they are first mapped to [0, 1] in order to be in range for RGB color components. Figure 5 shows an example brick texture along with its normal map. Normal maps are typically mostly blue because normal vectors in normal maps are expressed in tangent space, so they're usually pointing in the positive z direction.

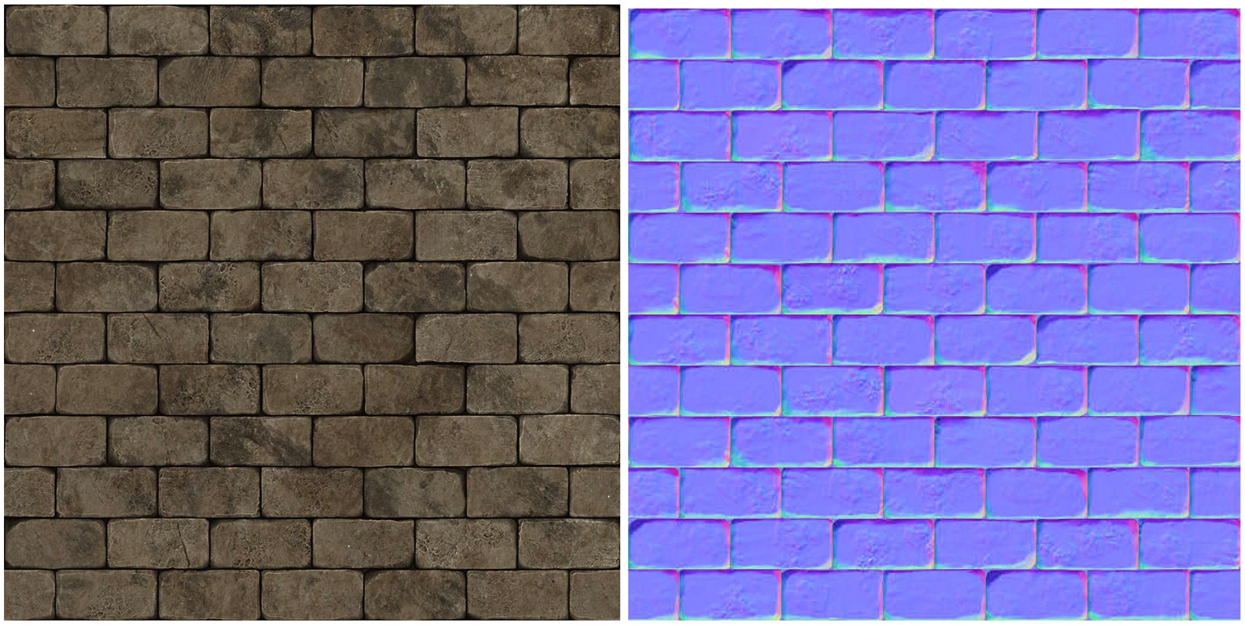


Figure 5 - Brick texture (left) and its normal texture (right) [3]

By converting the map back to [-1, 1] from [0, 5] normal mapping will work for a Z-facing surface, but once a surface is not parallel to the XY plane, the normals will not be correct since they are not transformed with the surface (Figure 6).

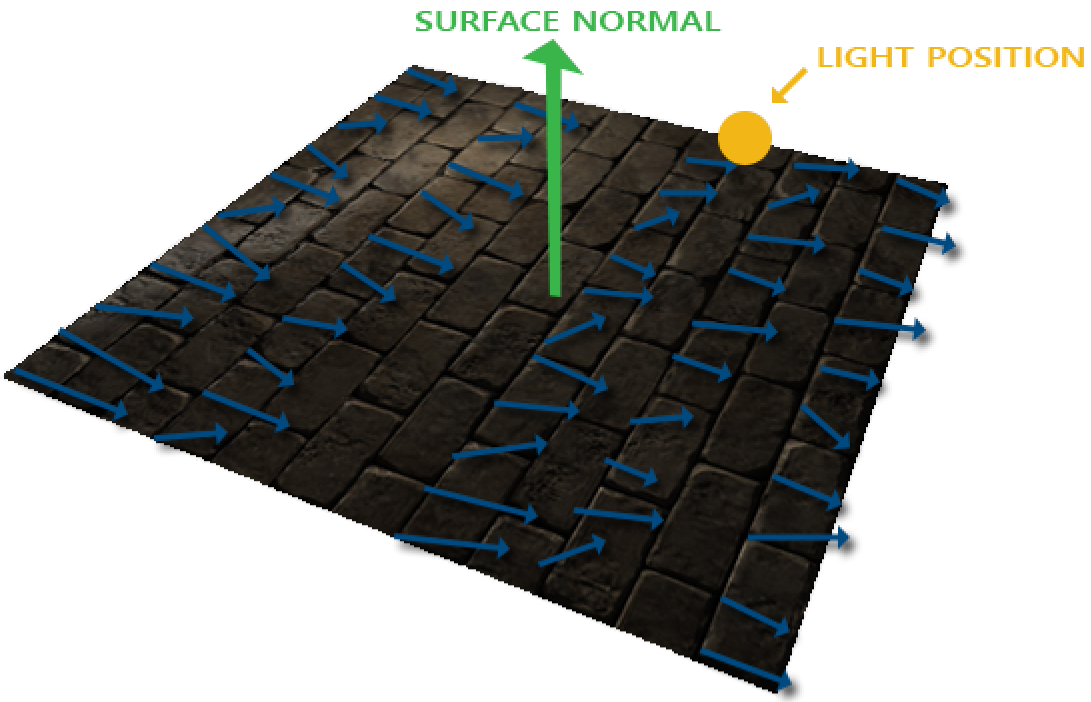


Figure 6 - Normals not being transformed [3]

In order to use the normals correctly, they must be transformed into the world space. To transform them, a TBN matrix is created from the tangent (T), bitangent (B), and normal (N) vectors of the surface, which are all perpendicular (Figure 7). The normal vector is trivial, but the tangent and bitangent vectors for each surface must be calculated.

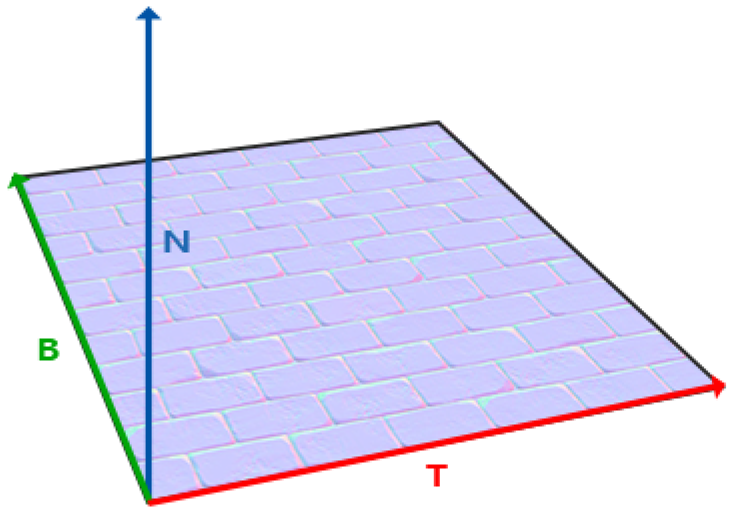


Figure 7 - TBN Vectors [3]

In order to calculate the tangent and bitangent, the texture coordinates are used. Figure 8 shows the basis of how to solve for T and N by solving for ΔU_2 and ΔV_2 since they are expressed in the same direction.

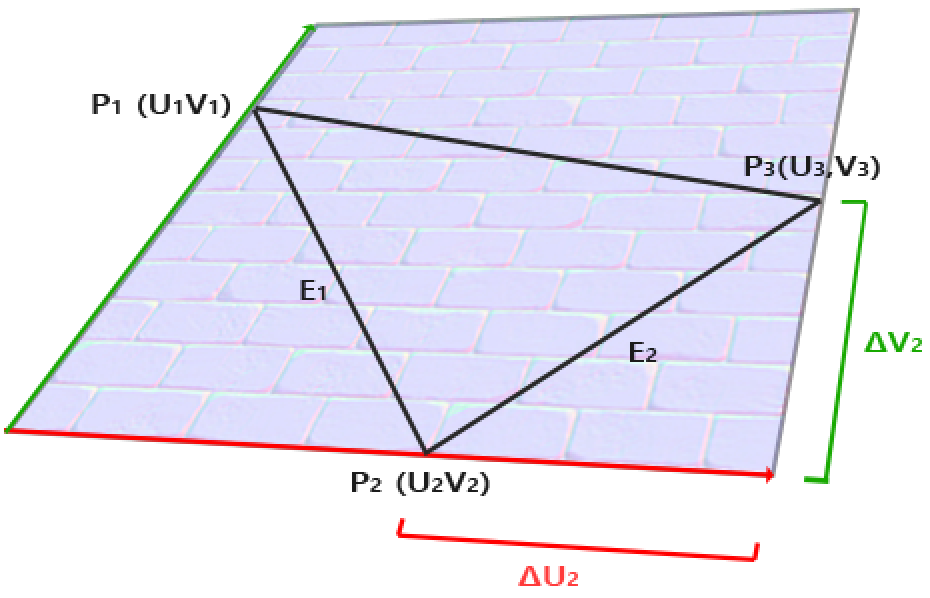


Figure 8 - Calculating T and B [3]

E_1 and E_2 can be described as the linear combination of T and B:

$$E_1 = \Delta U_1 T + \Delta V_1 B$$

$$E_2 = \Delta U_2 T + \Delta V_2 B$$

E can be calculated as the difference vector between ΔU and ΔV, which leaves T and B for solving by representing them as matrix multiplication:

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

By multiplying by the inverse of the ΔUΔV matrix, you can separate TB:

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

By calculating the inverse as 1 over the determinant multiplied by the adjugate matrix, T and B are solvable as follows:

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

In code, T and B are solved for each face upon importing an object file by modifying Tiny Obj Loader to calculate and store the values in another array which is then passed to the vertex shader. The code below shows the calculations where s is the current shape and i is the current face.

```
glm::vec3 v1 = glm::vec3(
    shapes[s].mesh.positions[shapes[s].mesh.indices[i * 3] * 3],
    shapes[s].mesh.positions[shapes[s].mesh.indices[i * 3] * 3 + 1],
    shapes[s].mesh.positions[shapes[s].mesh.indices[i * 3] * 3 + 2]);
glm::vec3 v2 = glm::vec3(...);
glm::vec3 v3 = glm::vec3(...);

glm::vec2 uv1 = glm::vec2(
    shapes[s].mesh.texcoords[shapes[s].mesh.indices[i * 3] * 2],
    shapes[s].mesh.texcoords[shapes[s].mesh.indices[i * 3] * 2 + 1]);
glm::vec2 uv2 = glm::vec2(...);
glm::vec2 uv3 = glm::vec2(...);

glm::vec3 e1 = v2 - v1;
glm::vec3 e2 = v3 - v1;

glm::vec2 dUV1 = uv2 - uv1;
glm::vec2 dUV2 = uv3 - uv1;
```

```
        float f = 1.0f / (dUV1.x * dUV2.y - dUV2.x * dUV1.y);

        glm::vec3 tangent = (e1 * dUV2.y - e2 * dUV1.y) * f;
        glm::vec3 bitangent = (e2 * dUV1.x - e1 * dUV2.x) * f;
```

Now that any imported object will have tangents and bitangents created and passed along with normals, all an object needs is a normal map texture and the shaders will do the rest.

```
uniform mat4 P;
uniform mat4 V;
uniform mat4 M;

layout(location = 1) in vec3 aNor;
layout(location = 3) in vec3 aTangent;
layout(location = 4) in vec3 aBitangent;

out mat3 TBN;

void main() {
    ...
    vec3 T = normalize(vec3(V * M * vec4(aTangent, 0.0)));
    vec3 B = normalize(vec3(V * M * vec4(aBitangent, 0.0)));
    vec3 N = normalize(vec3(V * M * vec4(aNor, 0.0)));
    TBN = mat3(T, B, N);
}
```

The vertex shader above creates the TBN matrix and passes it to the fragment shader.

```
uniform sampler2D normalTex;

in vec2 vTex;
in mat3 TBN;

void main() {
    ...
    vec3 normal = texture(normalTex, vTex).rgb;
    normal = normalize(normal * 2.0 - 1.0);
    normal = normalize(TBN * normal);
    ...
}
```

The fragment shader converts the normal back to [-1, 1] from [0, 1] and uses the TBN matrix to transform the normal into world space. Once the normal is transformed, it can be used in lighting computations correctly. The final result of normal mapping in ATTRACT can be seen in the comparison screenshots in Figure 9 below. The final effect made the buildings seem much less flat and plain with not a lot of extra processing.

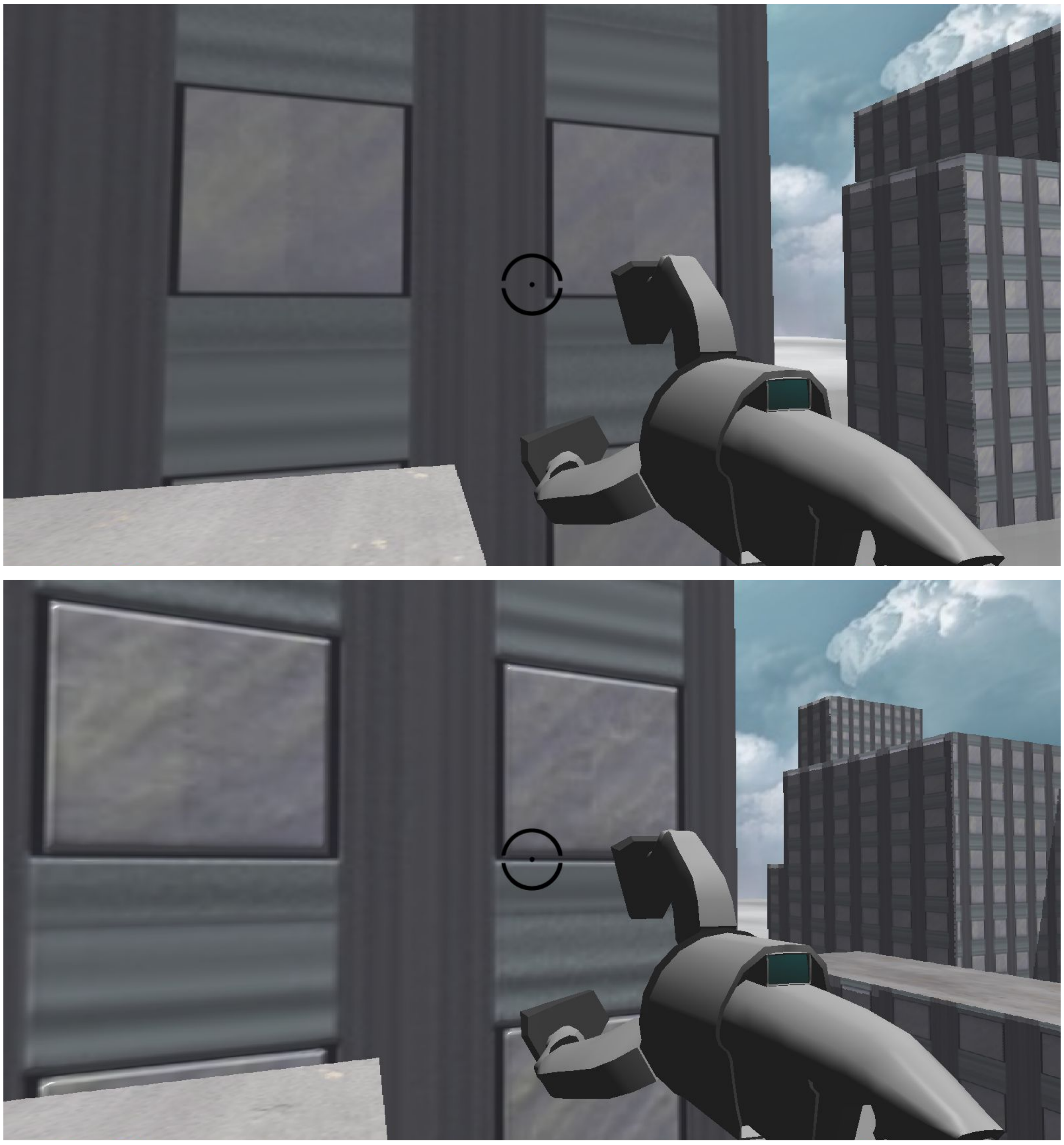


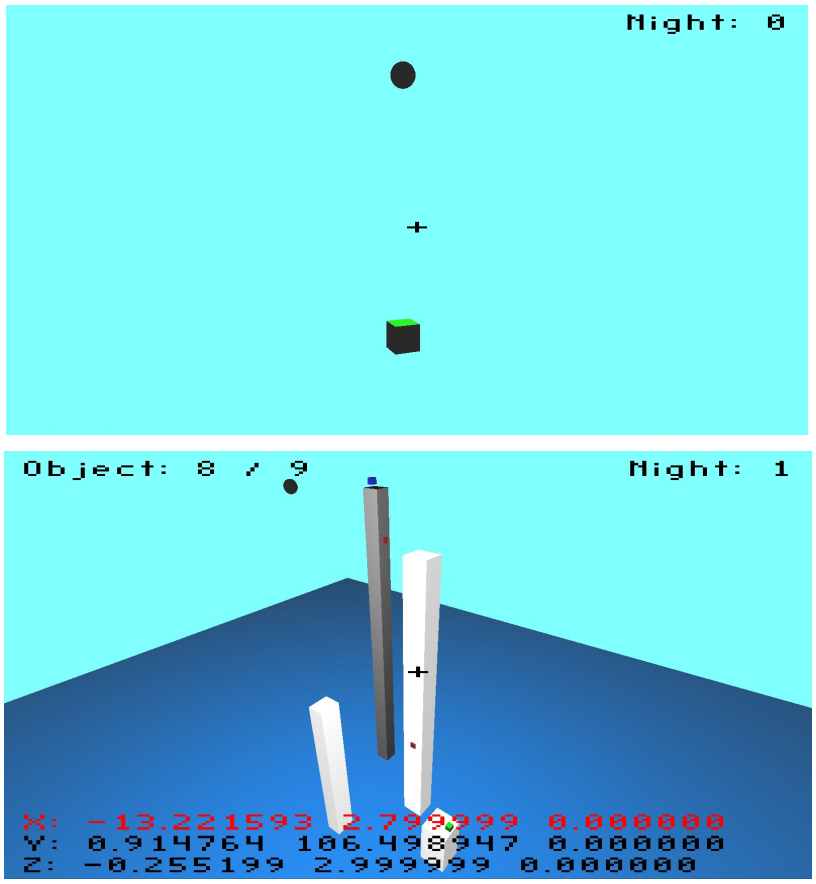
Figure 9 - Without Normal Mapping (top) vs with (bottom)

While there are sources to find textures with normal maps online, the texture for the skyscrapers did not come with one. The texture had already been chosen and fit nicely in ATTRACT, so instead of finding a new one, a normal map was created for it using Normal Map Online, a free normal map generator which is all client-based [10].

Level Editor

Most games need many different levels for progression and creating different challenges to overcome. In order to speed up the process of creating levels, a level editor was created to visually create levels in 3D; without it, levels would have taken painstakingly long to create by hardcoding levels into the game and constantly having to recompile and test to see if the numbers entered worked whatsoever. The level editor allows for creating new levels, saving levels in a format which is then read by the game, and loading previously saved levels to edit again. Levels are saved as plain text files with objects on each line; each object consists of comma separated values denoting their types, locations, and properties. The level editor is limited to creating city-like levels due to the fact that it only supports placing and scaling primitive cube objects in the world, but even with this limitation, interesting and challenging levels can still be created.

The level editor has free flying camera movement through the world with the WASD keys and mouse. Since it is an editor, there is no collision detection so that the user has no movement restrictions. To avoid having to create an extensive graphical user interface (GUI) with mouse interactive buttons, the GUI was kept to simple text for the current mode and almost all commands are given via keyboard commands.



```
Night: 0
```

```
Object: 8 / 9                    Night: 1

X: -13.221593 2.799999 0.000000
Y: 0.914764 106.498947 0.000000
Z: -0.255199 2.999999 0.000000
```

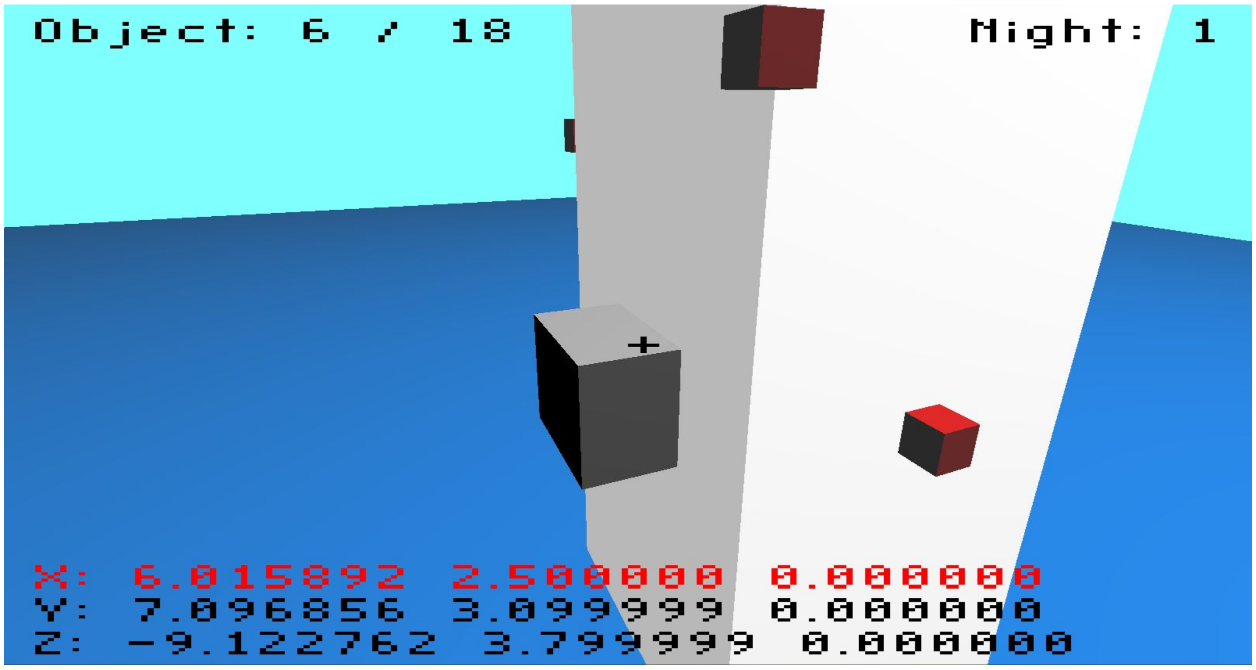
Figure 10 - Initial blank level (top) vs completed level (bottom)

Figure 10 shows a new level without any objects along with a completed level. To interact with the world, the following controls are used:

Action	Binding
Movement	WASD + Mouse
Toggle Mouse	M
Speed/Precision Modifier (move faster, change values slower)	Shift
Toggle Placement Mode	P
Previous Object	J
Next Object	K
Select Current Axis	X/Y/Z
Increase Scale	E
Decrease Scale	Q
Increase Rotation	Right
Decrease Rotation	Left
Increase Translation	Up
Decrease Translation	Down
Set Player Spawn	1
Set Spaceship Part	2
Set Light	3
Toggle Night Mode	6
Toggle Magnetic	4
Toggle Deadly	5
Toggle Door	7
Toggle Moving	8
Decrease Object Speed	9
Increase Object Speed	0

Move a moving object's final position instead of initial	Left Control
Delete Currently Selected Object	Delete

With the level editor, levels can be created either at night or in the day and they can have any number and combination of buildings and magnets, whether moving back and forth from one location to another or static. Currently, every level has one spawn, one spaceship part which indicates the end of the level, and one main light source. Deadly objects are currently implemented as fog, so most levels only have one large death floor which covers the entire span of the play area, but it is possible to put multiple deadly objects in a level. When placing or editing already placed objects, the axis is chosen and the user can manipulate the scale, translation, or rotation of the currently selected object on the selected axis (Figure 11).

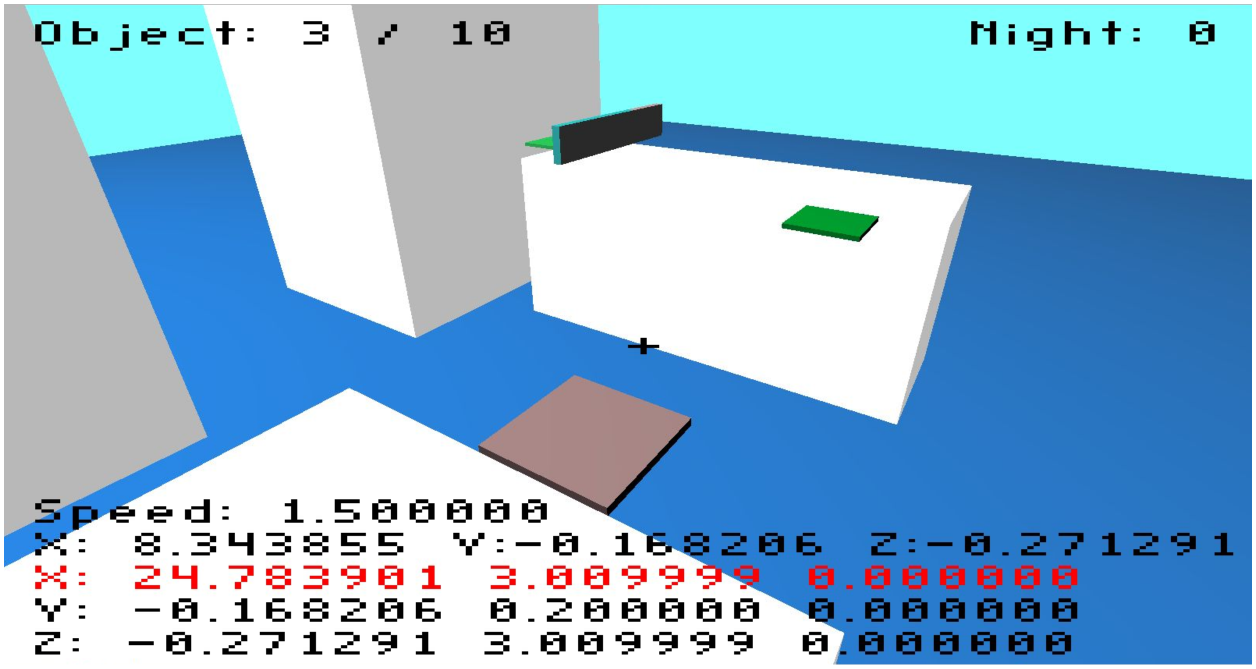


```
Object: 6 / 18                          Night: 1

X: 6.015892  2.500000  0.000000
Y: 7.096856  3.099999  0.000000
Z: -9.122762  3.799999  0.000000
```

Figure 11 - Editing an Object

As shown in Figure 12, moving objects are created by enabling movement, setting the movement speed, and moving the initial and final positions accordingly. The positions can be changed separately, but all scale and rotation changes will apply to the object as a whole. Moving objects are set to delay on each point for a couple seconds before continuing to move. Doors are set to move from one position to the other when the magnet gun is used on them (Figure 13).



```
Object: 3 / 10                          Night: 0
Speed: 1.500000
X: 8.343855  Y:-0.168206  Z:-0.271291
X: 24.783901 3.009999 0.000000
Y: -0.168206 0.200000 0.000000
Z: -0.271291 3.009999 0.000000
```

Figure 12 - Moving Objects

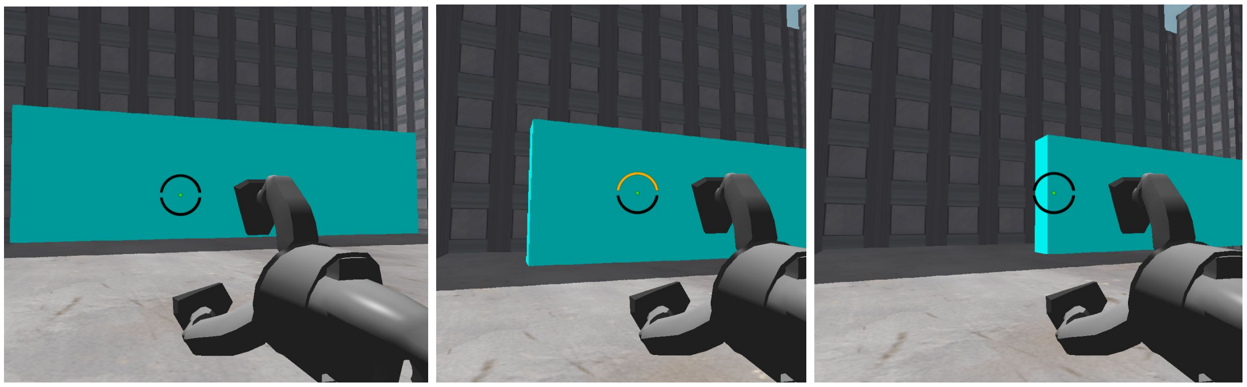


Figure 13 - Door activating (from left to right)

For importing and exporting, levels are stored as comma separated values with the first line indicating day or night, the second line indicating spawn, and the third line indicating finish. The subsequent lines define objects in the world including their position, scale, rotation, and other properties such as magnetic, deadly, and moving.

Overall, the level editor took the majority of development time available in the first quarter, but it saved large amounts of time by allowing for rapid level development and prototypes. In the second quarter, the level editor was not the primary focus, but it was expanded upon by adding new types of objects like moving objects and doors. It may not be fancy graphically, but it does everything it needs to, and the editor will not be seen by most players of the game.

Fog

At the start of the project, the skyscrapers in ATTRACT simply looked like they were magically floating in the air above nothingness. A ground plane was initially thought of as a solution, but ATTRACT was meant to feel like the player is very high up on tall skyscrapers, and a simple ground plane doesn't convey that well enough or show the player when they will die if they fall. Ground planes are also static and not very interesting. In order to convey the height of the buildings, cover up the floating buildings, and make the game more realistic and interesting, fog was implemented as part of the death floor. Exponential fog was implemented and is a simplified model of the real world phenomenon of light intensity reducing over distance due to absorption and scattering from air particles. For each particle light reaches, the intensity is absorbed by an attenuation factor. While the real world has a varying attenuation factor for each particle, this model is simplified by having a constant factor [2]. The resulting equation gives a value of the percent of light left:

$$f = e^{-d*b} = 1 / e^{d*b} \quad where\ d = distance,\ b = attenuation\,factor$$

To create the fog, the fragment shader calculates the distance from the camera to the fragment position and guarantees that there is always at least a small amount of fog:

```
float dist = abs(distance(fragPos, viewPos));
 // make sure there's always a little bit of fog
 if (dist < 5.0) {
     dist = 5.0;
 }
```

A texture of fog is then used for the color of the fragment, but the texture is also moved across the object over time. To get the final color of the fragment, f is solved for and used for the alpha. If f is below a threshold of 10% (alpha value over 90%), it will be set to 10% in order to always have slight opacity:

```
vec2 coords = vTex;
float t = time / 2000.0f;
coords.x += t;
coords.y += t;
vec3 fogColor = texture(diffuseTex, coords).rgb;
float f = 1 / exp(dist * 0.15);
// make sure there's always a little opacity
if (1 - f > 0.90) {
    f = 0.10;
}

FragColor = vec4(fogColor, 1 - f);
```

The fog was initially implemented without a texture, but it looked too flat and fake, so a texture was added and later made to move (Figure 14).

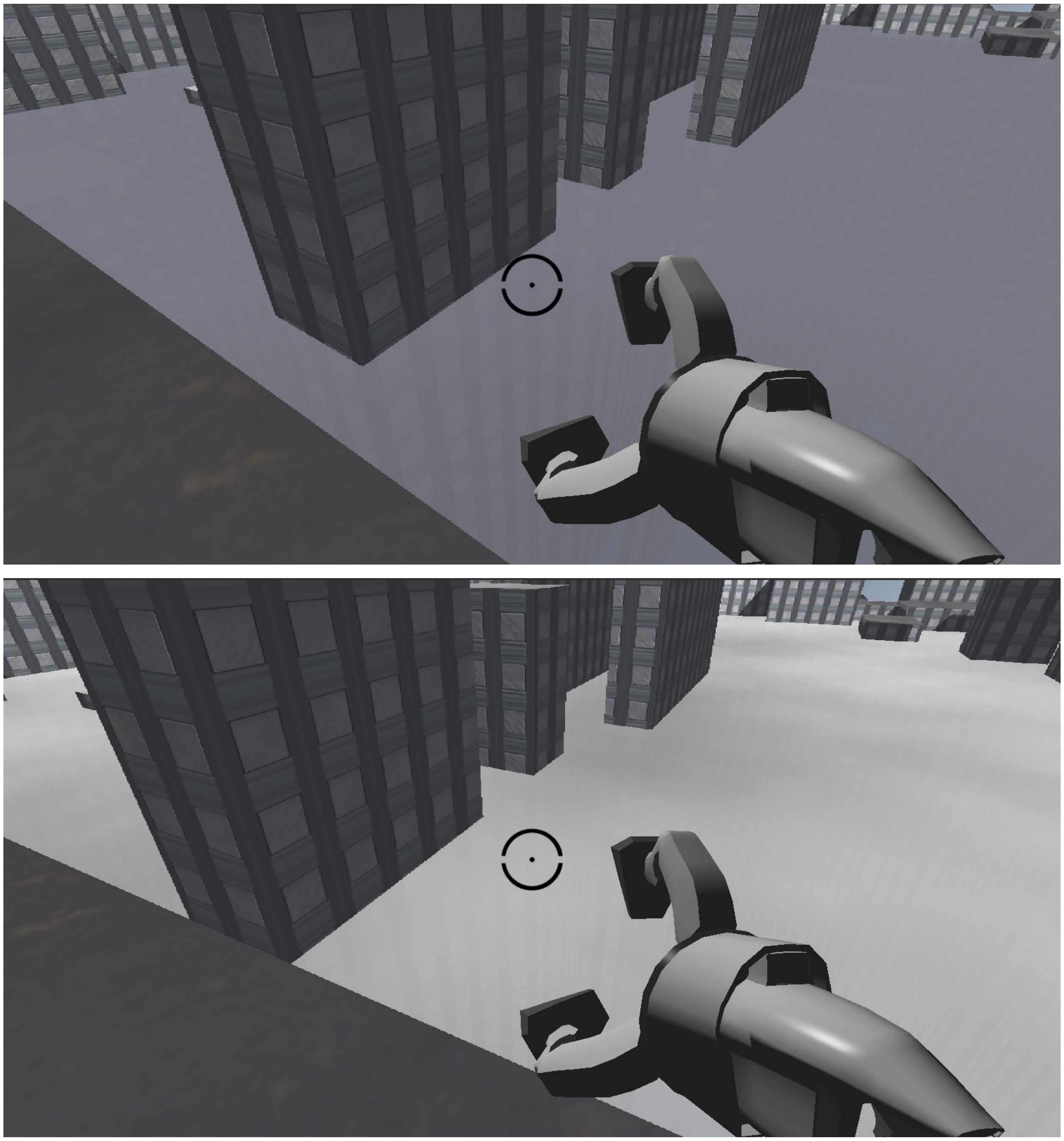


Figure 14 - Fog Without Texture (left) vs With Texture (right)

Results

The implementation of normal mapping, fog, and a level editor greatly improved the graphical quality, gameplay, and level design of ATTRACT. As shown in Figure 9, normal mapping allows for the skyscrapers to have more visual depth and react to light in more interesting ways. These more realistic walls are enhanced by the fog which is modeled off of physical fog and also serves as a warning for players to not fall. The overall effect is a graphically pleasing, realistic cityscape of levels which can easily be created and expanded using the level editor. Thanks to these technologies, varying stunning areas such as the ones in Figures 15-17 can be explored and conquered.

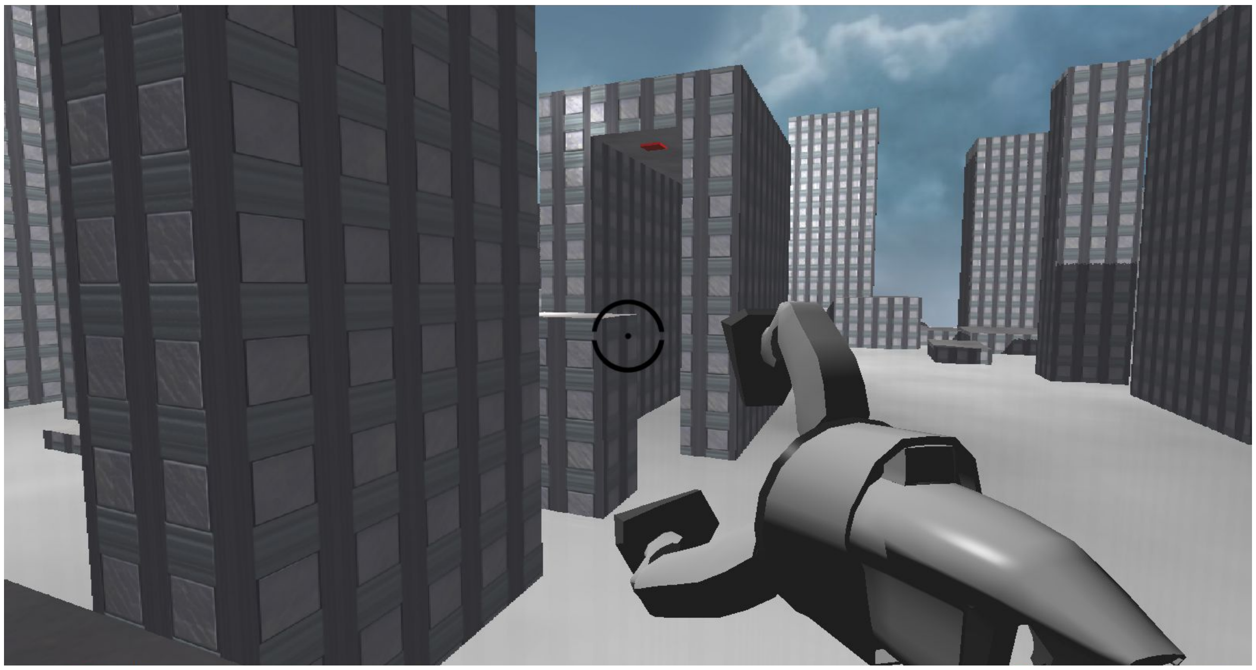


Figure 15 - Level 3

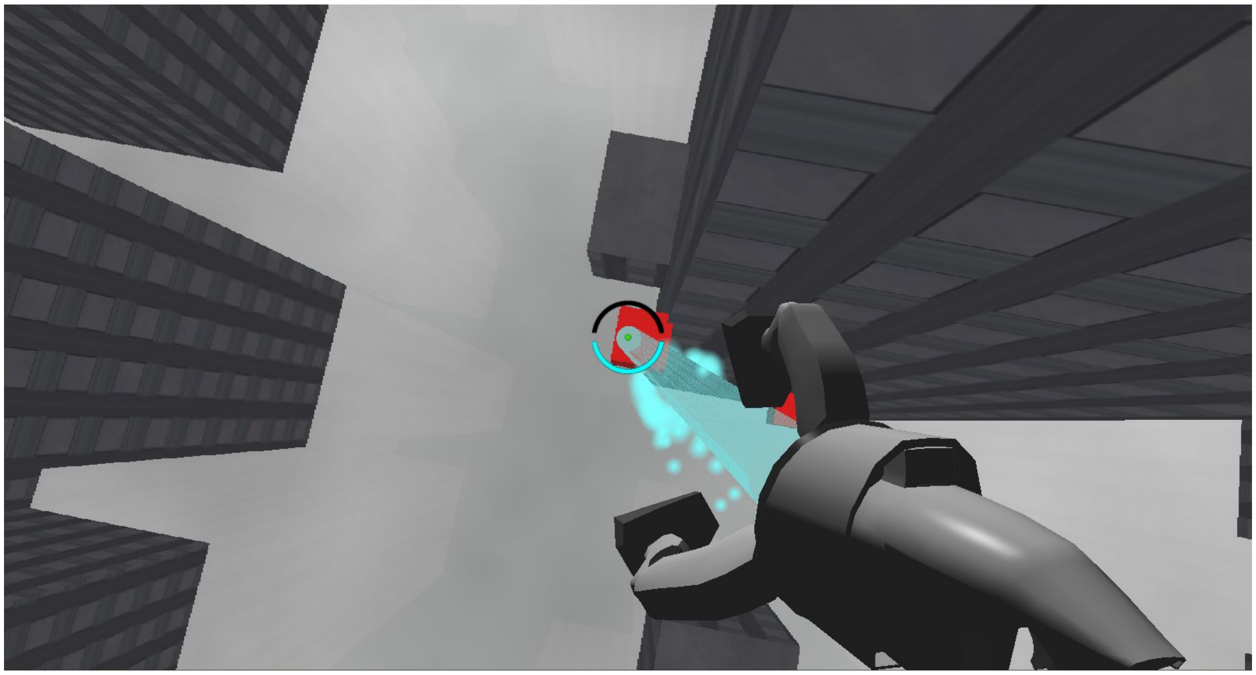


Figure 16 - Level 4

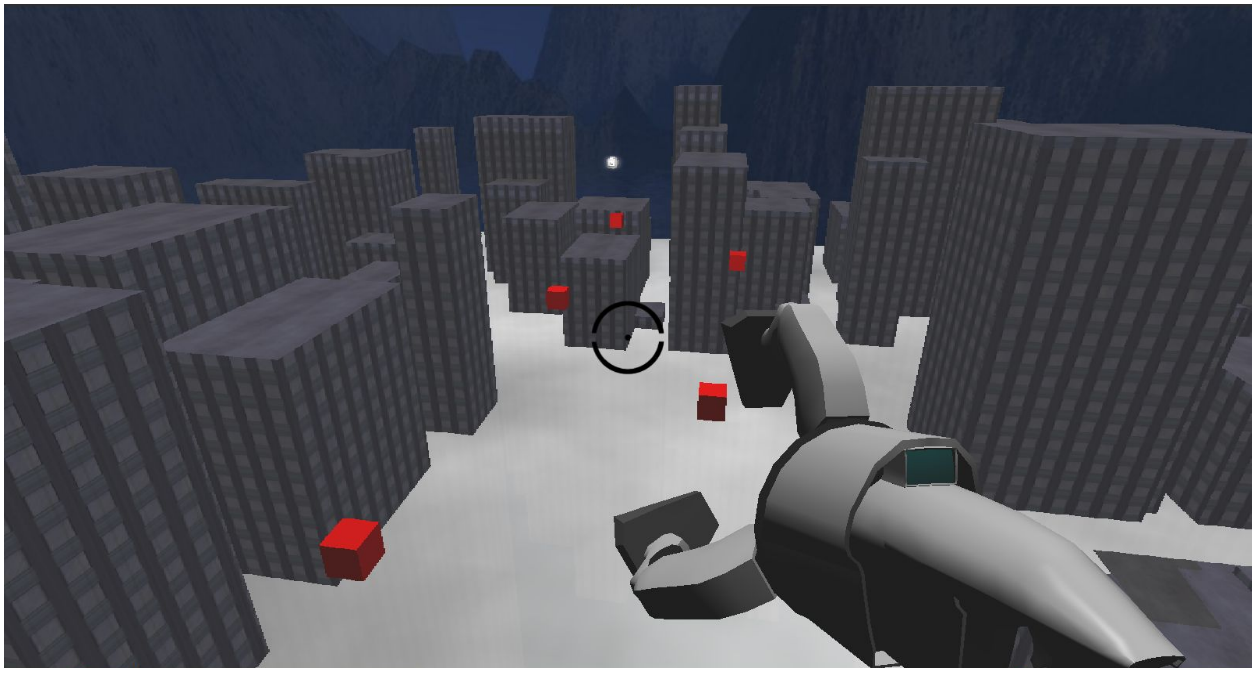


Figure 17 - Level 5

Reflections

Going into the first quarter, I had no idea what kind of game I was going to end up working on. I was glad to join onto the group for ATTRACT as it was a game that sounded very fun to me, which would make creating it that much more enjoyable. This is the biggest project I have worked on from scratch; a lot of time and effort went into it from me and my groupmates, and I believe we created a unique and fun game.

Having taken one real-time graphics course before starting this project and one during, my knowledge of the graphics pipeline has grown greatly. I have become comfortable with the OpenGL API, but I am still learning what it can truly be capable of. ATTRACT has given me a solid base knowledge of how to create a game from scratch along with bettering my overall programming skills.

Thanks to this project, C++ has become much more natural to use and my teamwork skills have improved. Due to the somewhat large codebase, communication was key for implementing new features. We had to work together and plan out overall architecture in order to create a working render cycle with multiple different graphics technologies from each of us. ATTRACT has strengthened all of these skills, but it has also given me a deeper respect than I already had for game developers and the massive amount of work that goes into even the smallest and sometimes overlooked aspects of games.

Future Work

If there were to be future work done for ATTRACT, the main additions would be polishing the graphics, as well as increasing the overall complexity of levels and objects within them. There is still work that needs to be done in order to give the levels a better look and feel. Cascaded shadow mapping allowed for better shadows than a typical shadow map, but they still need more work to make them look realistic. In addition, the game risks becoming visually stale, as there is a lack of visual variety within levels. Variations in level colors, building types, and textures would help prevent the player from becoming bored of the environment. The levels could also use more interesting geometry, as levels currently are primarily composed of cuboids. Adding objects such as fences or air vents on the roofs of skyscrapers could go a long way in making the world more believable.

Another feature which would help to make a more immersive game would be to add scripted scene elements to levels. These could trigger cutscenes or cause events to happen in the world, allowing for much more interactive, dynamic levels than what is currently implemented.

Over the course of the project, ATTRACT was only ever developed on and for Linux and Mac OSX. If the game were intended to be sold or put up for any kind of public download, a Windows version would most likely be needed. While the game in theory should be easily portable to the Windows environment, it would still take a decent amount of time and effort to make the proper environment changes to get a working version and then longer to verify that the game and all of its aspects work as intended. However, the time would be well spent if the goal was to pursue releasing the game.

References

1. *Bullet Physics Library*. Computer software. *Bulletphysics.org*. Vers. 2.86. N.p., 26 Jan. 2017. Web. 12 June 2017.
2. Craitoiu, Sergiu. "Create a Fog Shader." *In2GPU*. N.p., 20 Oct. 2014. Web. 12 June 2017. <http://in2gpu.com/2014/07/22/create-fog-shader/>.
3. De Vries, Joey. "Learn OpenGL, Extensive Tutorial Resource for Learning Modern OpenGL." *LearnOpenGL.com*. N.p., n.d. Web. 12 June 2017.
4. Felix. A screenshot of the player traversing the cityscape in Mirror's Edge. Digital image. *Felixmoviethoughts.wordpress.com*. N.p., 5 May 2016. Web. 12 June 2017.
5. *FMOD*. Computer software. *Fmod.com*. Vers. 1.09. Firelight Technologies, 15 Feb. 2017. Web. 12 June 2017.
6. Gardner, Ashley. *A Screenshot of a Player Solving a Puzzle with the Portal Gun and a Redirection Cube.* Digital image. *Thesenecatimes.wordpress.com*. N.p., 13 Dec. 2012. Web. 12 June 2017.
7. *Legend of Zelda Skyward Sword.* Nintendo of America. 2011. Video game.
8. Meiri, Etay. "Cascaded Shadow Mapping." *OGLdev.org*. N.p., n.d. Web. 12 June 2017.
9. *Mirror's Edge.* Electronic Arts. 2008. Video game.
10. Petry, Christian. *Normal Map Online*. N.p., 28 June 2014. Web. 12 June 2017. <http://cpetry.github.io/NormalMap-Online/>.
11. *Portal 2.* Valve Corporation. 2011. Video game.
12. Riccio, Christophe. *GLM*. Computer software. *Glm.g-truc.net*. Vers. 0.9.8. G-Truc Creation, 22 Jan. 2017. Web. 12 June 2017.
13. A screenshot of the player using the hookshot in Legend of Zelda Skyward Sword. Digital image. *Zeldadungeon.net*. N.p., n.d. Web. 12 June 2017.
14. "U.S. Video Game Industry Generates $30.4 Billion in Revenue for 2016." *Theesa.com*. Entertainment Software Association, 19 Jan. 2017. Web. 12 June 2017.