COS426 Final Project Spring 2015:
Escape: A First-Person Horror Game

Chen, Jessie jc29
Wong, Kenny kjwong

Introduction

<u>Goal</u>

We tried to make a 3D game in first person. The player spawns in a dark building, with only a flashlight with limited power at his disposal, and attempts to escape the building without getting caught by an enemy. The player can choose to toggle the flashlight off to save power, and turn it back on to give off a burst of light that rapidly drains power. We implemented 3D rendering, player camera movement, lighting and shadows, collision detection, enemy movement, texture mapping, particle movement, and non-graphics related parts such as sounds, timers, and options to pause and restart.

<u>Previous Work</u>

In the past, Jessie made a 2D shoot-em-up game in Java that utilized collision detection, particle movement, and general gameplay elements such as menus and pausing.

<u>Approach</u>

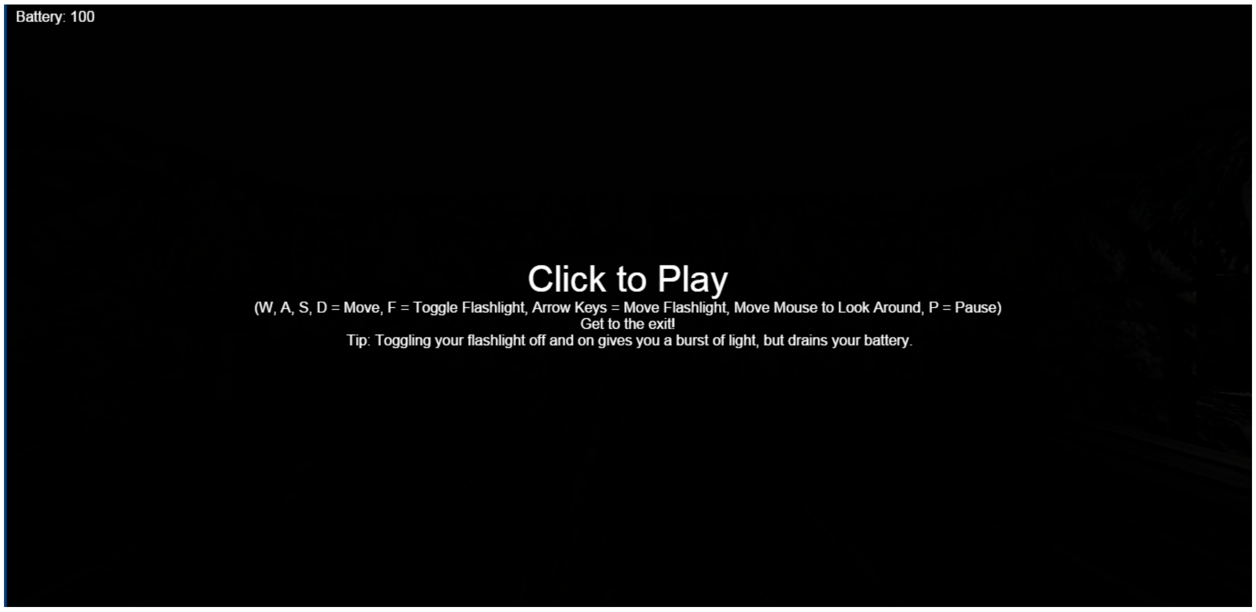
For our project, we decided to use Three.js. We were both familiar with it due to having taken this class, and from viewing the examples of Three.js projects on their website, it appeared reasonable to use it to make a 3D game that would run well.

Methodology

<u>Implemented Features</u>

-3D Camera Perspective
-3D Objects and Models
-Texture Mapping, Normal Mapping
-Lighting and Shadows
-User Input and Player Controls
-Enemy movement
-Collision Detection
-Particle System
-Time Restrictions
-Sound Effects
-Game Conditions, Pause and Restart

<u>Sample Gameplay Images</u>

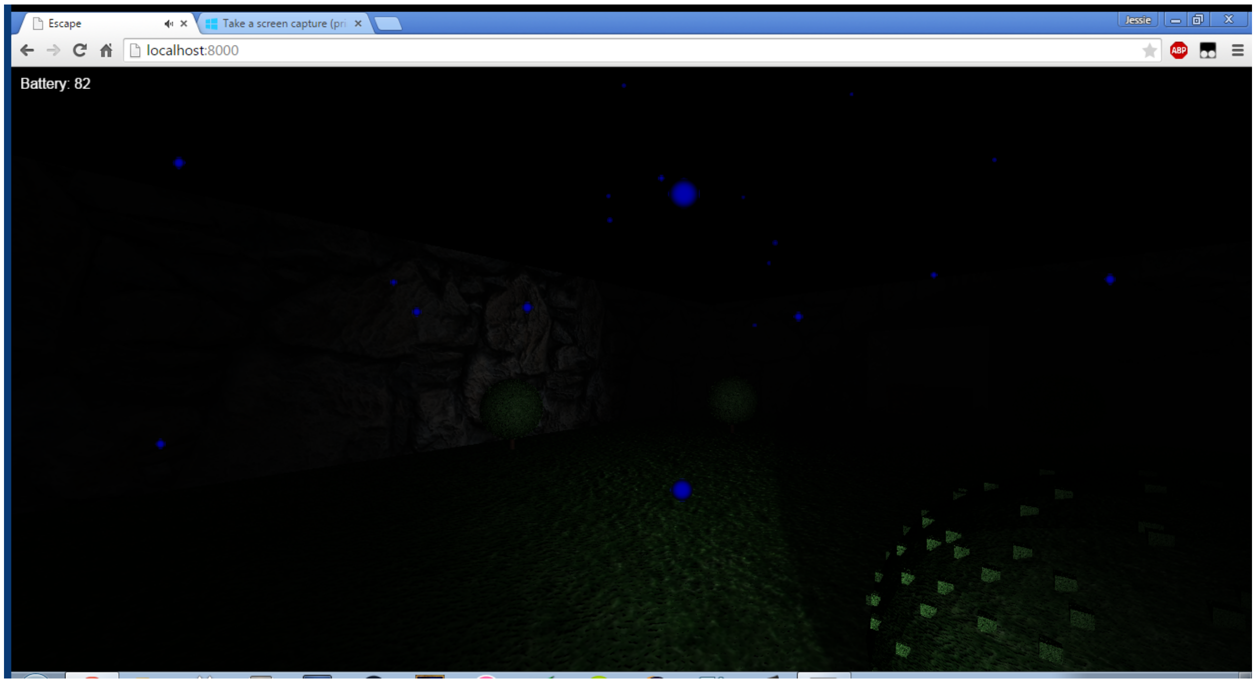

Battery: 100

Click to Play

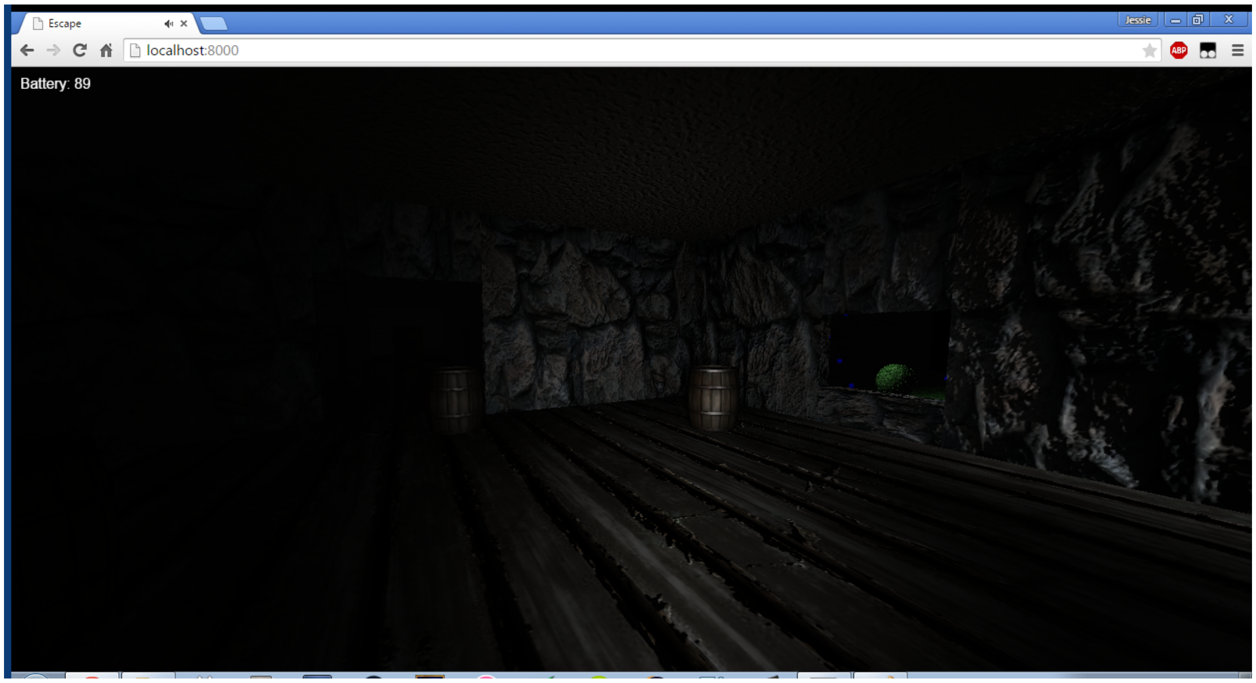
(W, A, S, D = Move, F = Toggle Flashlight, Arrow Keys = Move Flashlight, Move Mouse to Look Around, P = Pause)

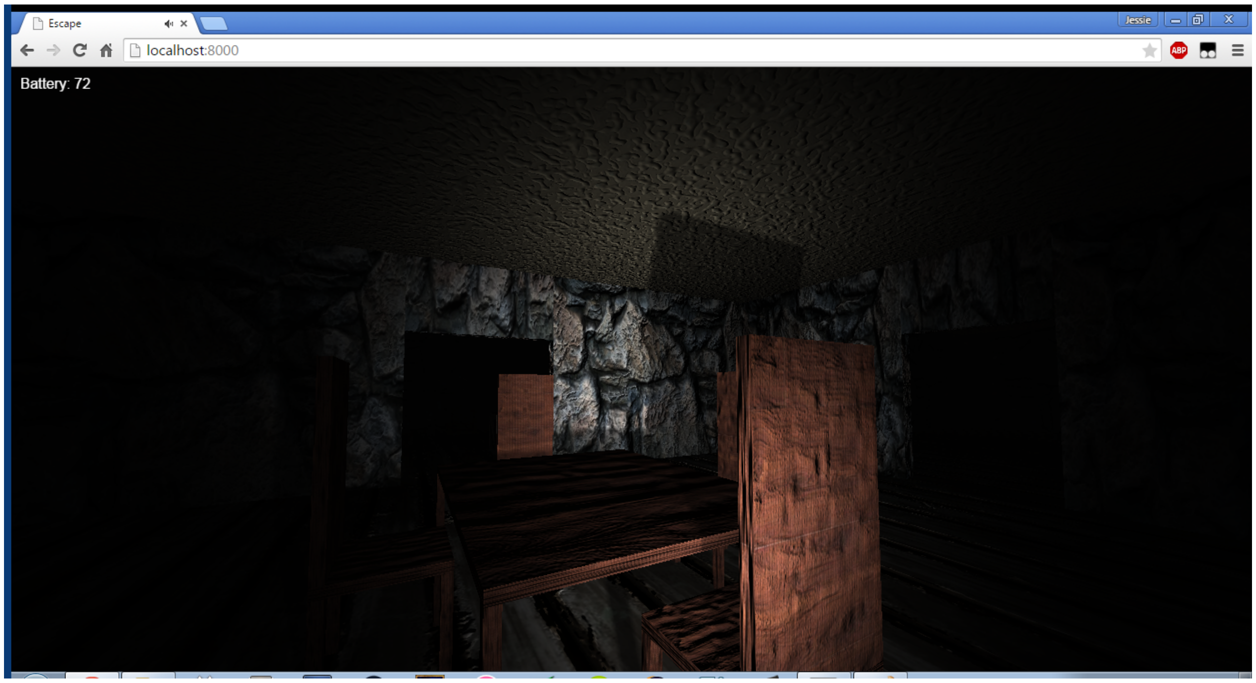
Get to the exit!

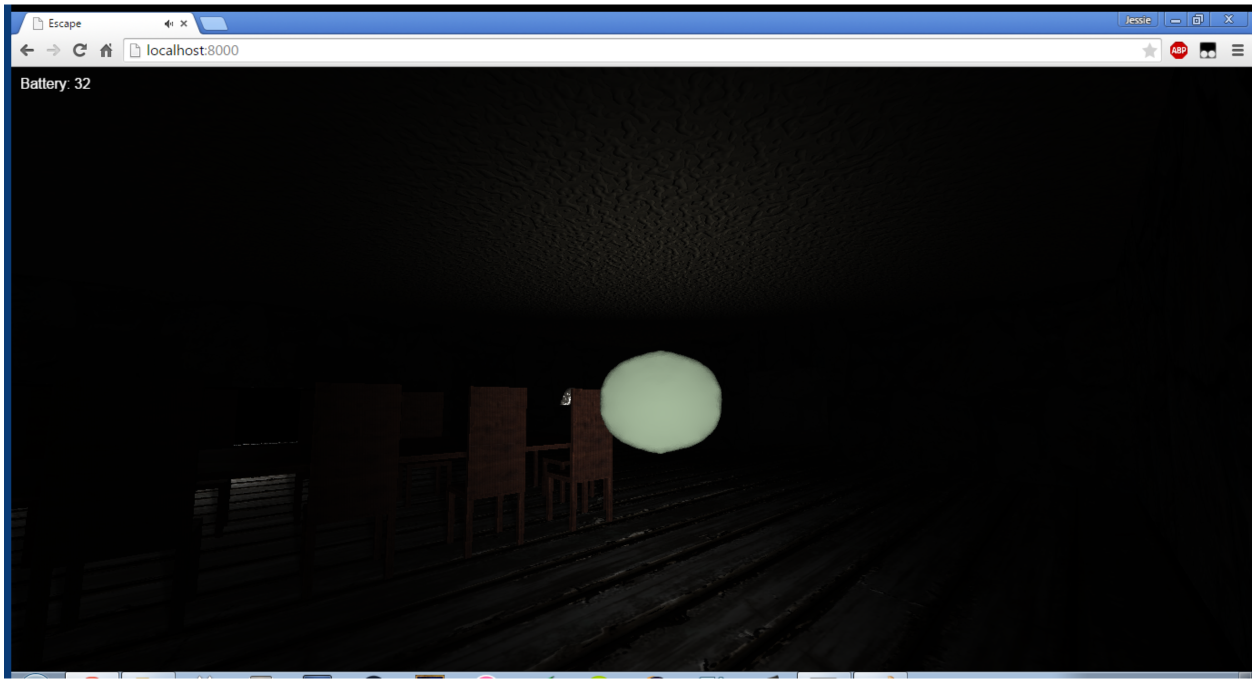
Tip: Toggling your flashlight off and on gives you a burst of light, but drains your battery.

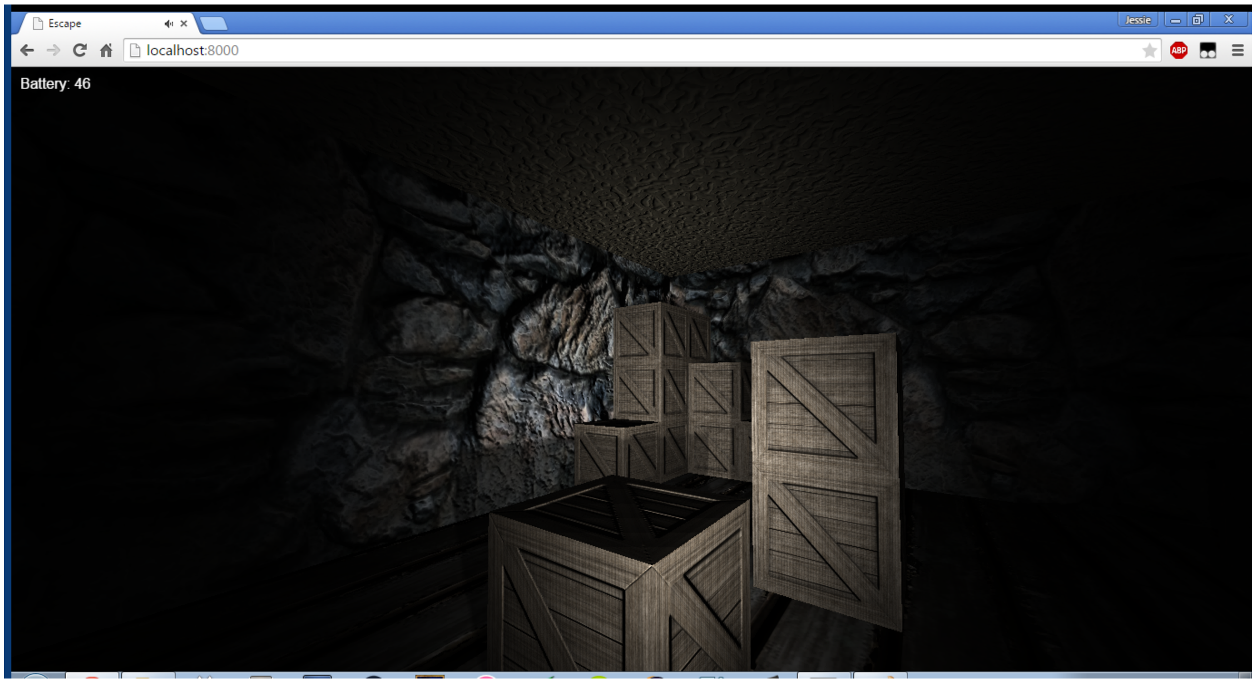


Battery: 82


Escape
localhost:8000
Jessie
Battery: 89


Escape
localhost:8000
Jessie
Battery: 72





3D Camera Perspective

We decided for our game to be in the first-person, where the camera sees what the player's eyes are supposed to see. We used THREE.PerspectiveCamera, which fits our needs perfectly. We set the field-of-view of the camera to 75, which is a typical FOV used in most first-person games.

3D Objects and Models

For the player environment, we needed to first design a basic level, which would be the building, and then add objects and models to the level. To make the level, we used THREE.PlaneBufferGeometry. We used PlaneBufferGeometry since it has a smaller memory footprint than PlaneGeometry, which we would need due to the large amount of total objects in our game (The game also ran more smoothly using BufferGeometry). The ceiling, floor, and simple walls consisted of single Planes, but for walls with windows and doors on them, we needed to add a thickness to the plane to make it look more realistic. This was done by breaking the wall into several Planes. Another option would be to use several objects with BoxGeometry to make the wall, but we felt that using Planes allowed more control over each section of the wall, for shadows and textures.

For the objects to make the level more interesting, we both made our own models and also imported free ones found online. The models that we made were chair, table, and bush. The chair and table was made using several THREE.BoxGeometry, and the bush was made using THREE.PolyhedronGeometry, with boxes placed randomly on several of the faces.
Models were imported using THREE.JSONLoader and THREE.ObjectLoader, depending on the format of the model we downloaded.

Model Sources:
Barrel: http://codepen.io/nickpettit/pen/ctpna
Ghost: https://clara.io/view/9381e504-f9e1-438e-aabe-857d5aec26a6
Skull: https://clara.io/view/663f6caa-74f5-4c01-8a60-54b01c26b283

Texture Mapping, Normal Mapping
In order to make the game appear realistic, we implemented textures on our objects. Textures were loaded using THREE.ImageUtils.loadTexture. In order to load the textures in-game, we had to run a server, using “python -m SimpleHTTPServer.” Due to different wall and object sizes, without adjustment, textures would be stretched or squashed. To fix this, we set the wrapS and wrapT parameters of each texture to THREE.RepeatMapping, to allow for repeating textures on a face, and set the number of times to repeat proportional to the width and height of a face. Since the walls with windows and doors are made of multiple separate planes, the textures do not line up perfectly, but we did not think it would be worth our time to manually adjust each texture perfectly.

THREE.MeshPhongMaterial allowed us to add textures to geometries using the “map” parameter. It also allowed us to implement normal mapping by specifying the “normalMap” and “normalScale” parameters. We adjusted the normalScale for different objects based on what we thought would be most visually appealing, increasing it for the rock walls and decreasing it for the ceiling, for example.

Texture sources:
Grass: http://www.rendertextures.com/grass-seamless-texture-20/
Rock Wall: http://www.rendertextures.com/seamless-rock-texture-18/

Wood Floor: https://www.filterforge.com/filters/11289.html
Ceiling: https://www.filterforge.com/filters/11092.html

Lighting and Shadows

The primary focus of our game is the dark lighting and shadows. The most important light is the player flashlight. We used THREE.SpotLight, which allowed us to control the intensity, the FOV, the distance, and the position. Three.js allows us to add child objects to other objects, so we attached the flashlight to the camera and placed the flashlight behind the camera pointing in the same direction as the camera. This means that the flashlight will always be in the same position relative to the camera and will always point in the same direction as the camera.
We also placed two more SpotLights, one at each window (there are two windows in our game).

Shadows were implemented using setting the lights to cast Shadows with .castShadow, and the objects to receive and cast shadows (.castShadow and .receiveShadow). We set in the renderer shadowMapEnabled to true and shadowMapType to THREE.PCFSoftShadowMap to have soft shadows.
The trickiest part about shadows is that in Three.js, they are not implemented in the way the real world works. If you shine light directly on a shadow created by another light, the shadow should disappear, but that is not the case. The area still remains dark. In areas with only a single light, usually the player flashlight, this is not a problem, but when the flashlight is shined on shadows generated from window lights, artifacts occur. This is because shadowmapping is used, which allows for faster real-time rendering. The workaround that we used was to not have the walls castShadows, and to strategically place objects and the lights to minimize how noticeable the artifact is.
One possible implementation would be to calculate the shadows ourselves, using our own shaders, similar to Assignment 3. However, this could slow down the game immensely. The Three.js documentation already states that their shadow casting is expensive, so this is a trade off that we decided to take.

User Input and Player Controls

We used THREE.PointerLockControls to allow the player to look around using the mouse. To save time and frustration, we used the code found here:
http://threejs.org/examples/misc_controls_pointerlock.html
as the basis for the controls. This provided us with camera direction and player movement (no collision detection). We removed gravity, since jumping is not necessary for our game, and simply set the player's y velocity (up and down) to be 0.

We also added other inputs that the player can perform. Using the arrow keys allows the player to move the flashlight around. This is done by moving the flashlight's position relative to the camera, and also moving its target, an object attached to the camera specified by flashlight.target, in the same direction. Pressing F toggles the flashlight on and off, done by changing the intensity of the light. Pressing P pauses or unpauses the game, and Space is used to restart the game after the player has won or lost (discussed in Game Conditions, Pause and Restart).

Enemy movement

We needed an AI for the enemy to give a difficulty level to the game. An AI can take a long time to make, depending on how complex it is. Since AI is not so related to graphics, we decided to implement a very simple one: Always move toward the player, ignoring collisions. This actually fit our goals very well, since a ghost, an enemy associated with horror, does not have to worry about collisions. We calculated the vector from the ghost's position to the players, and moved the ghost along that direction at constant speed. Since ghosts float, we set the ghost at the level of the player and simply moved it along the x and z direction.

If the ghost comes within 10 units of the player, by calculating the length of the vector, then the game ends. For a nice touch, we sink the player into the ground after he is killed. How the game ends is discussed in Pause and Restart.

Collision Detection

For collision detection, we attached 8 THREE.Object3D objects to the camera at different positions, one for each direction on the xz plane (front, frontleft, left, etc).We did not have to consider the y direction since we did not utilize vertical movement in our game. Because the objects are attached to the camera, they are always in the same position relative to the camera. To detect collisions, we created a THREE.Raycaster from the lower body of the camera (camera's y position minus a number) to each object, and used the raycaster.intersectObjects method to detect possible collisions. intersectObjects sorts by distance from the raycaster origin, with index 0 being the closest. We got the distance to the closest object, and if it was closer than a specified number, we halted movement in that direction. This works fine for large objects like walls and boxes, but does not work as well for chairs and tables, due to their skinny legs. To fix this, we put an invisible box (THREE.BoxGeometry) around each chair and table for the player to collide with instead.

Particle System

To make the environment surrounding the player even more interesting, we implemented a particle system that would serve as rain falling outside the building. This was done using THREE.Geometry. The vertices array in Geometry specify the particles, and we move the particles by updating the vertices at every step. We generated a number of particles with their initial positions chose randomly in a specified box volume. To move the rain, we decreased its y position at each step. To keep it raining without adding new particles at every step, we moved a particle to back up to the top if its position was below a certain value, creating a looping animation. To reduce the number of calculations, for the rain outside the window, instead of having rain over the entire outside plane, we only had it just outside the window, reducing the number of particles while achieving a similar effect.

Time Restrictions

We implemented a timer in the form of a battery. We do not actually set a strict time limit, so the player can still play after the battery runs out, but the game is intentionally nearly impossible then due to lack of light. The timer is implemented by creating a variable starting at 100, and slowly decrementing it by an amount at every render frame while the light is on. If the light is off, the battery very slowly regains power by incrementing the variable, so the player can wait until the flashlight gains power again if he runs out(though he may be caught by the enemy by then). Since turning on the flashlight gives it more power, we drain the battery faster by decrementing by a higher amount, this amount decreasing as the light remains on.

Sound Effects

Sounds can add a nice touch to the game without requiring extensive work. We used Howler.js to import the sounds and control their volume. To play the walking sound whenever the player walks, we set the sound to play on an endless loop at volume 0, and then set the volume to 1 whenever the player inputs a move command. For the ghost, we scaled the volume of the sound based on the ghost's distance from the player, with the volume increasing as the ghost draws nearer.

Sound sources:
Ghost: http://soundbible.com/1030-Zombie-Attack-Walk.html
Walking: https://www.freesound.org/people/Robinhood76/packs/4056/

Game Conditions, Pause and Restart

The player can either win or lose the game. To win, the player must reach the end of the level, indicated by using the player's z position, which is a door that leads to the open. To lose, the player must come into contact with the ghost. When a game ends, players should not be able to input move commands anymore. This is done by using a boolean variable that specifies if the player is in-game or not, and allowing movement only when in-game. Another variable is used in a similar manner to pause the game, which halts the player's and ghost's movements. When the game ends, a message appears in HTML text indicating whether the player has won or lost, done by setting a variable to true only if the player has achieved the win condition. The player can then replay the game by pressing Space, which sets the ghost and player positions back to their starting location and resets the battery life and win variable.

Results

Bugs

The ghost model does not load properly (it appears inside out). It is likely due to an error in the model, since the skull and barrel models that we found load into the game correctly.
If the game freezes or lags momentarily, the player can potentially glitch out of the building into a wall, and will be unable to reenter it.

Occasionally the rain particles will not spawn properly, resulting in them falling in a plane instead of at random positions.

Attempted Implementations and Features

An attempt at adding textured lighting was made, but it did not work properly. The attempted implementation was by adding a plane with a texture of the intended light effect in front of the flashlight, and writing a shader for the depth of the material which is used to indicate the shadows.

Particles were added to come from the ghost model, but this caused the framerate to drop immensely.

Discussion

Overall

We feel that we did a decent job with our project. We created a finished product before the deadline.

Future

In the future, we should look into improving the framerate, adding new enemies or levels, adding new graphics features such as textured lighting.