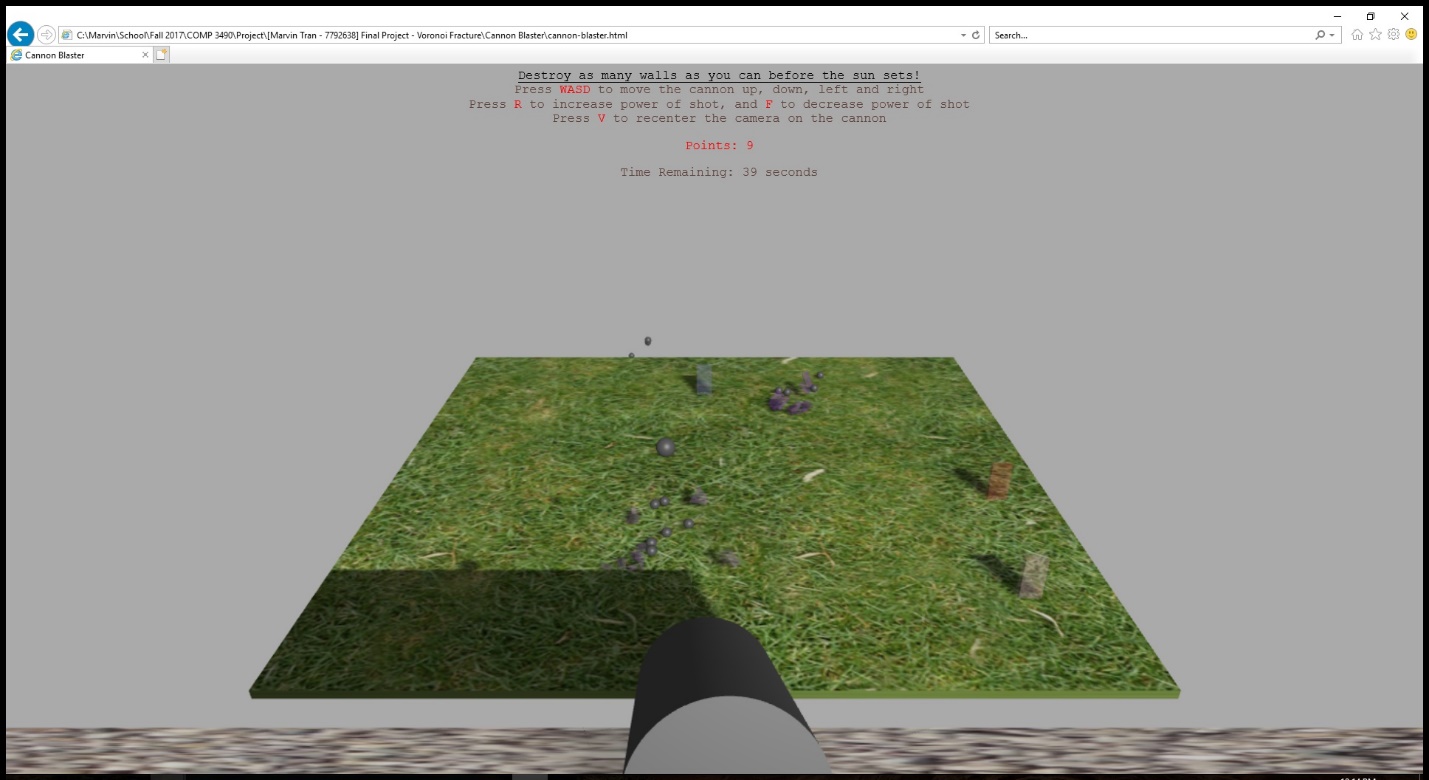
Cannon Blaster

*Voronoi Fracturing*

*COMP 3490: Computer Graphics 1*



Marvin Tran - 7792638

Fall 2017

***Introduction***

For my final project, I wanted to utilize three.js to make a fun mini-game. In Assignment 1 and Assignment 2, I became familiar with many of the basic requirements for setting up a scene such as positioning the camera, creating geometries, adding lighting and I also learned how to add some basic physics. For my final project, I wanted to work more with the physics side of things. Inspired by the example shown in class where balls were used to various objects in the scene [1], I wanted to create my own version of this, but add some more complexity.

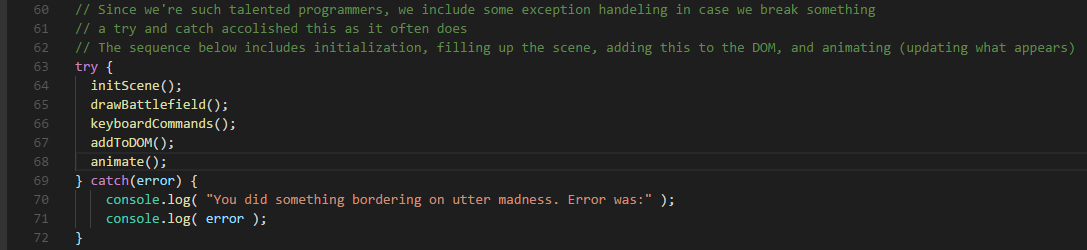
The idea is that a cannon sits on top of a cliff and shoots down into a valley at glass walls that will break by Voronoi fracture. The goal is to destroy as many glass walls as possible before the time runs out. There will always be a set number of glass walls that will spawn in the field, so the player has a few options to choose from. Extra scenery such as grass and trees will be in the valley and a sun will rise and set to show the remaining time.

I wasn’t sure how difficult implementing the Voronoi fracture will be, but I was also thinking of adding some volumetric light scattering. This would be implemented by making the breakable objects as tall glass walls where light could shine through, similar to stained glass windows and how the light shines through and creates long rays. As the sun rises and falls, the rays of light would hit the ground at different positions.

In the end, I was not able to add the lighting effect, but the tall glass walls breaking by Voronoi fracture was fully implemented. I opted not to create additional decorations which would be put into the scene, to help with the performance of the game. When a wall is broken into many fragments with each piece bumping around and casting shadows, the performance of the game does sometimes drop. As such, I made the game very simple and only included a ground which the glass walls spawned on and a cliff where the cannon sits on top.

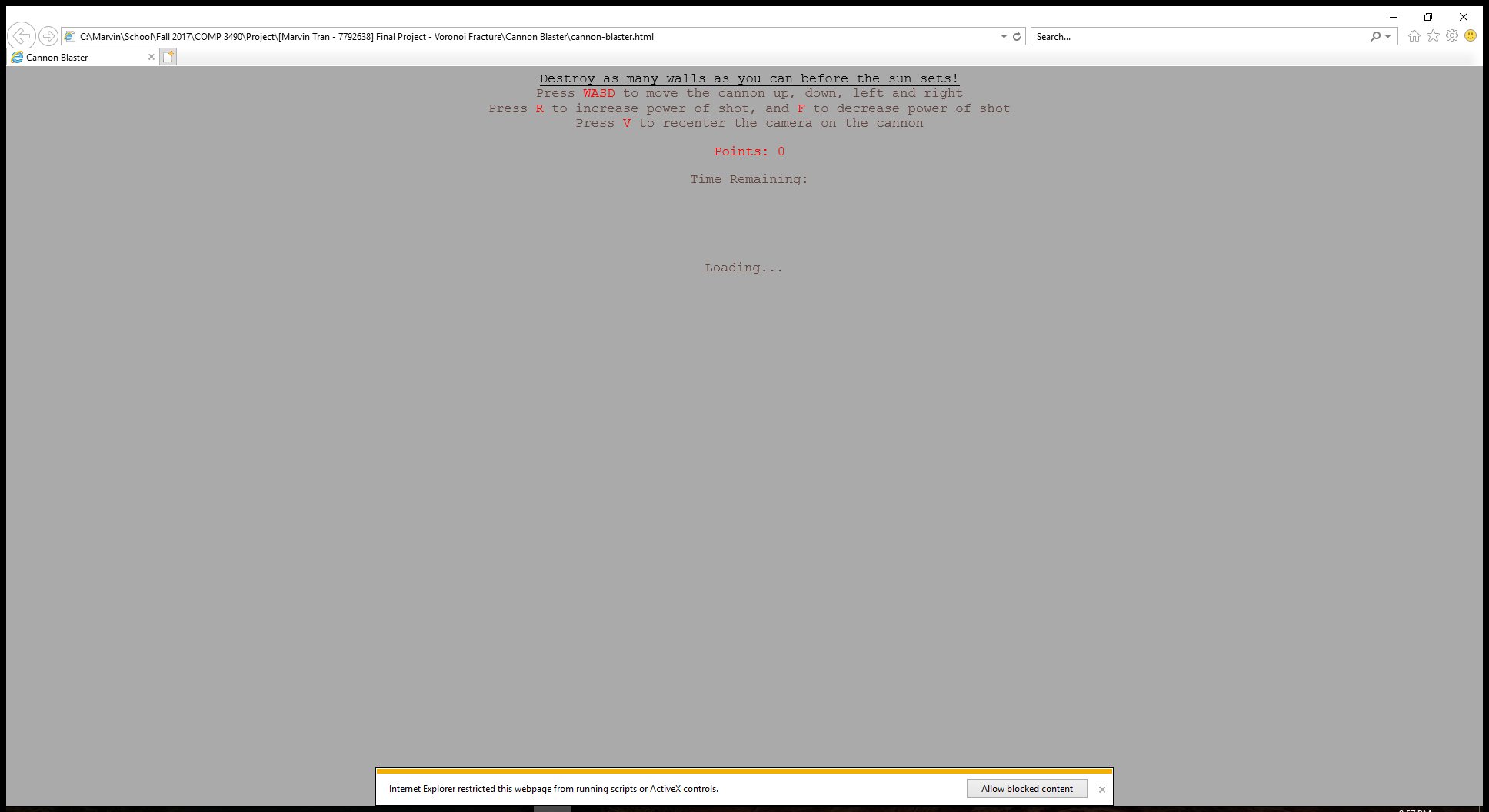
***Initial Setup***

Starting out, I decided to use the skeleton code provided with Assignment 1. This was because I was already familiar with the structure and organization after using it for Assignment 1 and Assignment 2. It would give me a nice way to quickly start development without having to start from scratch. This means that some of the structure of the code will be same. For example, the try/catch had a very similar structure, and functions such as addToDOM(); and animate(); were left largely left untouched.



Additional scripts were added to the skeleton to provide various functionalities to the game. A script called “Physijs” was used provide the physics such as the cannonball flying through the sky and the wall pieces falling down and interacting with each other. Physijs is built on top of another script called “ammo.js” which is a direct port of the “Bullet” physics engine. Physijs basically makes adding physics to a scene very easy, and so I decided to use this script.

One problem that occurred early on was Physijs’ use of WebWorkers. From my understanding, many browsers such as Chrome and Firefox block WebWorkers by default[2] and I could not figure out how to run Physijs on Chrome, my preferred browser. I tried using Internet Explorer and was met with this message at the bottom of the page.

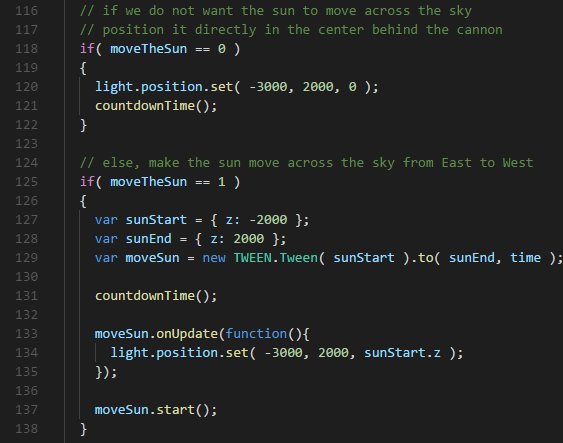


After clicking “Allow blocked content” the game loaded fine and I was able to use Physijs. This means I used Internet Explorer to test my game, but I used Visual Studio Code to develop and code the game.

Another script was tween.js which allowed for a sort of timer to update a variable from a start, to and end. This was used in functions such as moving the sun and deleting walls.

The last script that was added was a Javascript implementation of Steven J. Fortune's algorithm to efficiently compute Voronoi diagrams [3]. I quickly realized that implementing a Voronoi fracture algorithm by scratch was something that was a little bit too technical and out of reach with my current capabilities. In the interest of time, I decided that finding a script to do the calculations for me would be a better idea.

***Setting up the scene***

As seen in the try/catch block, the code starts out with the function initScene() which sets up basic requirements for a scene such as a renderer, a camera and some lights. We also have a Tween, which moves the sun from z: -2000 to z: 2000. This simulates the sun rising and setting from East to West. There is also a variable named moveTheSun which is one of the game setting that can be changed, located at the very top of the code. I added the ability to not move the sun, because I found that performance suffered a little bit when the shadows had to be updated constantly. The last part of the set is an event listener to allow resizing of the screen, to always make the cannon in the center of the screen.

***Geometries***

Moving on we have the function drawBattlefield() which creates the basic geometries. A valley is created, which is basically just a big square with grass textures. A cliff is created which is just a tall rectangle and it is textured with rocks. The textures for the valley and the cliff were taken from the examples given throughout the various examples that Physijs provided [4].

On top of the cliff is a cannon which is a simple cylinder. I had quickly dabbled with trying to import a mesh of a cannon, but gave up because of time constraints. The cannon is initially set with a default rotation which makes it point dead center on the screen. Moving the cannon is achieved through the “WASD” keys which moves the cannon up, down, left and right respectively. I ran into problems with rotation the cannon where it seemed like the cannon was moving, but it instantly snapped back to the rotation it was at before. I read that I need to set an objects \_\_dirtyRotation to “true” to rotate objects, but when I did that with the cannon, it still did not work. As a workaround, I decided to remove the cannon from the scene each time, and add it to the scene with the new rotation. This seemed to work perfectly.

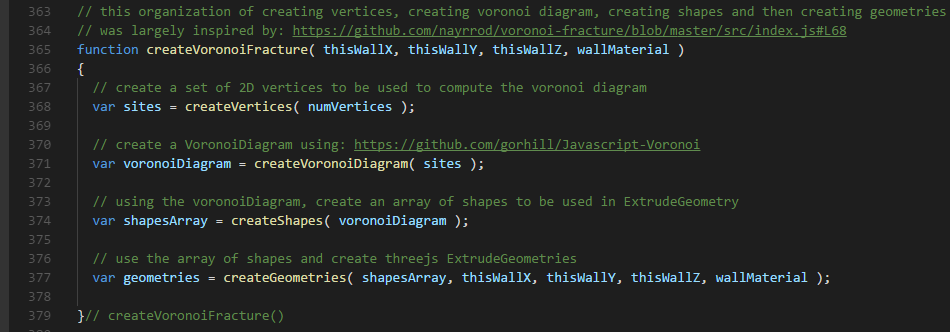
Physijs has special Physijs.createMaterial() which allows objects to have friction and restitution which is bounciness. I made the cannonballs have a high restitution to make them bounce a little bit and it seemed to look nice. The position of the cannonball moves as the cannon rotates. We set the direction of the cannonball shot with a method called setLinearVelocity().



If I had to be totally honest, I don’t really know the math behind what each xVelocity, yVelocity and zVelocity mean. I ended up setting each of these variables to random numbers and explored what each of the changes did. After exploring, I discovered the zVelocity shot the cannonball further left and right, the yVelocity shot the cannonball higher or lower and the xVelocity seemed to push the cannonball more forwards, which seemed like it gave it more power. I ended up hardcoding what I thought were comfortable maximum left/right values for zVelocity, and other maximums for yVelocity and xVelocity. In the end it looked alright, but I’m sure there was probably a much cleaner way of calculating each of those values.

***Voronoi Fracturing***

In the drawBattlefield() function, a function createWall() was also called to create the breakable glass walls. But we will take a look at how the Voronoi fracturing works. In this createWall() function, an event listener, unique to Physijs. Every geometry has a ‘collision’ event which passes values such as the object that collided with it in the “other\_object” variable [5]. Whenever I created a wall, this collision event would keep looking to see when anything collided with this wall, be it a cannonball, or pieces of another wall. Since the wall could collide with the ground, I made sure “other\_object != ground”. Then, I would remove this wall and call a function called createVoronoiFracture().



The structure is very clear. First I create a set of random x,y vertices. Then I use those vertices and create a Voronoi Diagram. This voronoi diagram was created by using the script I mentioned in the “Initial Setup” section. The script would use the vertices and create a voronoi diagram that contained all of the cells and their edges. For each cell I would draw trace the edge of the cell and make a shape. This shape was put into an array and used to make a geometry. This simple and clean structuring of code was largely inspired by another script who that also did something with this voronoi diagram script [6].

Each of these geometries was placed at the exact location that the wall I deleted earlier was, to give the illusion that the wall broke at that instant. Overall, it looked very realistic, but performance issues do occur with so many pieces being created.

***Conclusion***

Overall, I am very happy with how the project turned out and I really did learn a lot. Some changes I would make would be to make the fracturing at the exact location of where the ball hit and to implement the lighting, but I am happy with the end result.

***References***

[1] <https://threejs.org/examples/#webgl_physics_convex_break>

[2] <https://github.com/chandlerprall/Physijs/issues/141>

[3] <https://github.com/gorhill/Javascript-Voronoi>

[4] <https://github.com/chandlerprall/Physijs/tree/master/examples/images>

[5] <https://github.com/chandlerprall/Physijs/wiki/Collisions>

[6] <https://github.com/nayrrod/voronoi-fracture/blob/master/src/index.js#L69>