Derek Costello, Alexander Alguezabal

Dummy Simulator User Manual

Dummy Simulator

Usage Guide.

Controls:

W - Forward Walking

A - Left Walking

D - Right Walking

S - Backwards Walking

Mouse - Controls FOV movement.

Left Click - Shoots a laser (1.0x force)

Right Click - Shoots a cannonball (2.0x force)

Overview:

You spawn inside a 512x512 terrain. The goal of the game is to walk around and shoot a variety of spawning dummies. Once shot, the dummies will fly around. Their bodies will contour with the placement of the associated bullet.

Dummies spawn within a 25 block radius of the player's character.

If not shot, dummies will respawn within 60 seconds of the original spawn time.

Walking is restrained to the bounding box of the terrain (512x512).

General game music plays casually in the background, while lazer and gun sounds are also present when shooting.

Discussion & Decisions:

Our project was built using a 3D Graphics software known as "Three.js". To simulate the physics of the game, we used a popular extension called "Physi.js". Physi.js allowed us to use things such as collisions, forces, and gravity with ease!

Why was this simulator a challenge to build?

One of the toughest parts of the simulator to build was the flexibility of the dummies. Allowing them to maintain their form as **rigidbodies** while also flying around the map was tough.

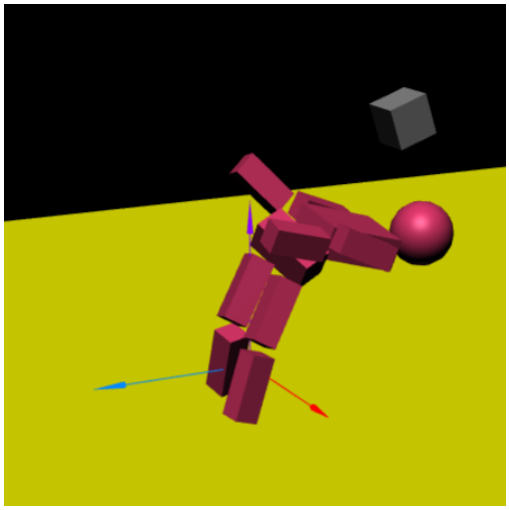
Originally the plan was to have the dummies use spring-like physics on each of its joints, but that became more difficult once we tried to do collision detection. It was noticeably easier to use **premade rigidbodies** from a physics library.

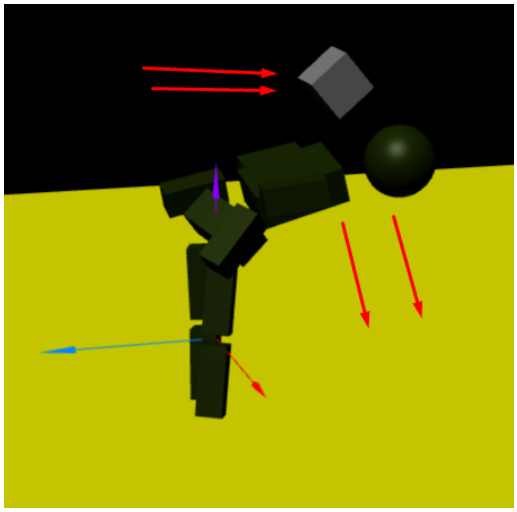
Rigidbodies are objects that cannot be overlapped by other rigidbody objects – this allows for seamless bouncing and colliding.

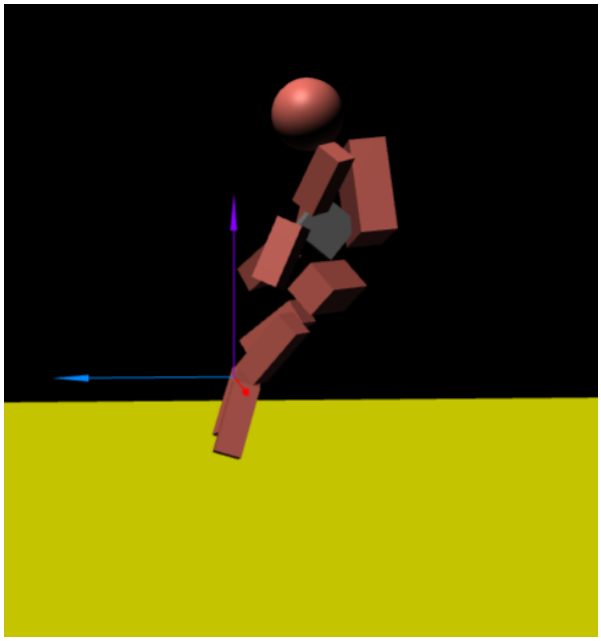
We ended up using a Physi.js to assist in the process of maintaining a rigid, flexible body for the dummies.

Here are some screenshots of our test environment for the dummies.

They are receiving an incoming force applied from a Physi.js BoxGeometry.







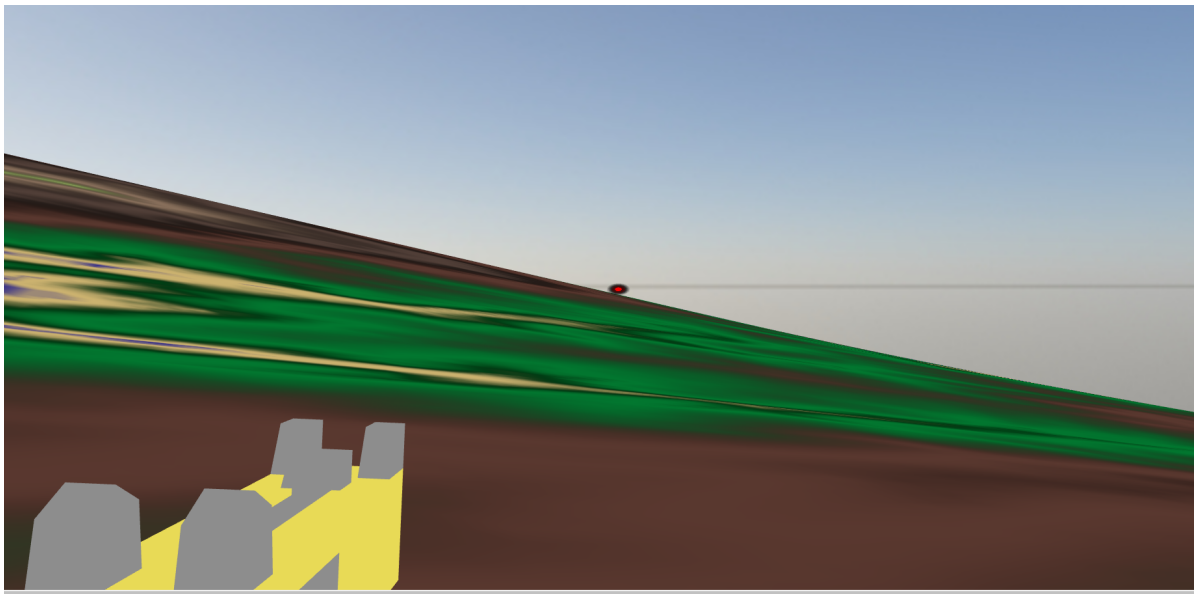
Additionally, getting the gun **model** and **texture** to load into the scene was a challenge.

Model = Shape of object we are loading into scene. This is usually designed outside of Three.js if we are trying to load it in.

Texture = Color and look of the object in Three.js. We can put grass on a ground model, felt on a poker table, you name it!

Aligning both to follow the camera was tough, as we wanted to have some animations for the movement of the gun alongside the camera.

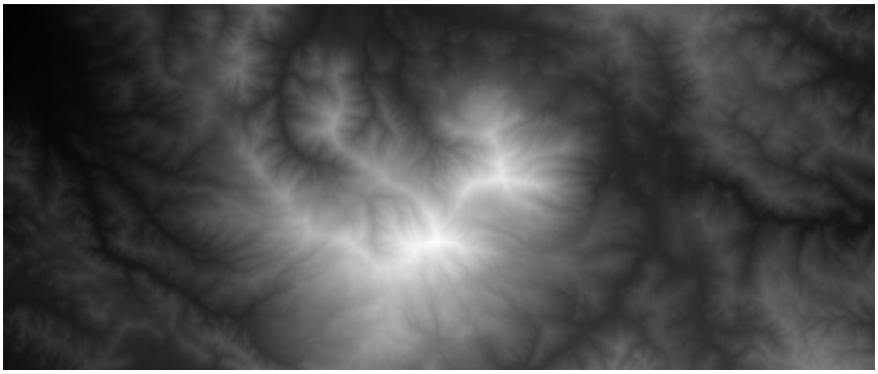
In the end, we decided to attach the gun model together with the camera as a group allowing the gun to stay in a consistent position.



Terrain generation was something we also played around with.

Originally, we used **vertex shaders** and **displacement maps** to load in a texture from a heightmap.

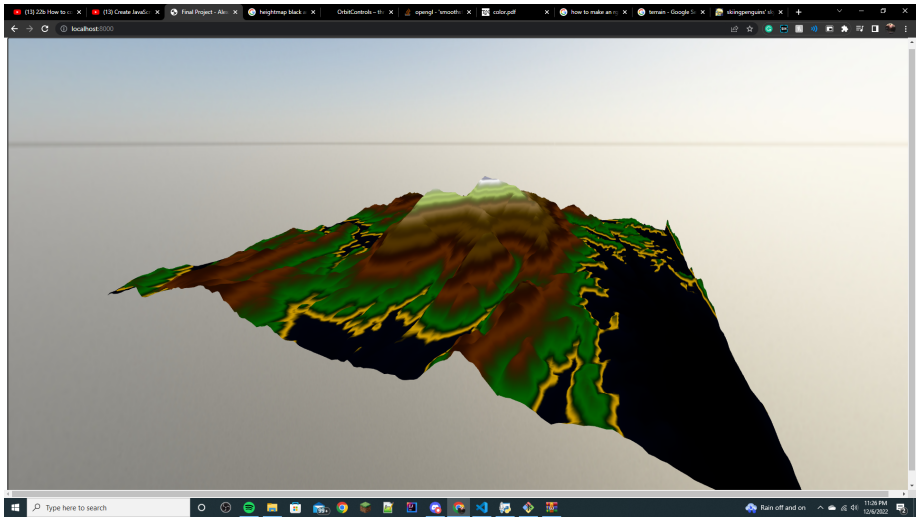
> **Vertex shaders** allow us to edit the vertices of an object (in our case, a plane used to represent the ground) quickly using our Graphics Card (which is much faster at those operations).
>
> **Displacement maps** are black and white images that tell us the height that each vertex should be placed at. Where there is white, the vertex will be higher -- darker areas have lower heights. Below is the height map we intended to use for our game.

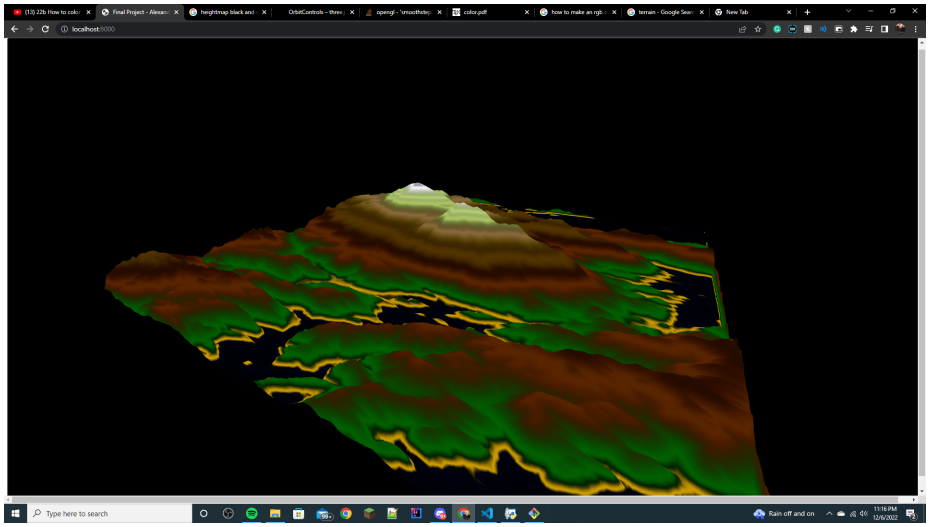


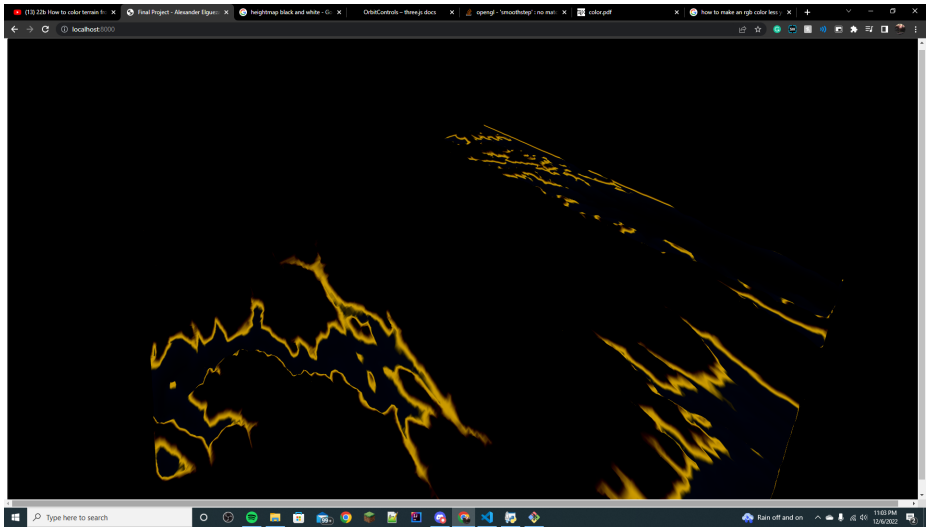
Though, we quickly realized that we could not **find the ground** effectively with this solution. We looked at solutions for this issue, but quickly realized that it would be too complicated to attempt to resolve.
Eventually, we simply edited the shape itself in Three.js so we could **find the ground**.

To '**find the ground**,' we used a method called "**Raycasting**", where we take the position of our point-of-view (**camera**) in our scene, and make an arrow (**ray**) directly downward. Then, we ask the "**Raycast**" which object was touched first by that **ray**. We assume the ray intersects the ground first, so we set the position of our **camera** to be just above the intersection (the point at which the ray crosses the object) we found.

The following is some images of the generated terrain in previous versions.







During this process we used **fragment shaders** which allow for displacement of each **fragment** to change the color of the terrain based on the heightmap provided. This in turn caused some issues.

Fragment shaders are shaders (AKA work with the computer's GPU) that work on the individual "**Fragments**" of an object in Three.js.

When an object is loaded in Three.js, it is split into a bunch of smaller units called "**Fragments**". It is safe to think of this as each area on the object that corresponds with a pixel (dot on the screen), but that is a bit oversimplifying.

These **fragments**, once in the **fragment shader**, are then adjusted based on what we want from the shader -- this is usually a visual change such as color.

In our case, we wanted to change the color of each **fragment** depending on its vertex's height.

We enjoyed the generated pattern and decided to keep it mapped to the displayed terrain.