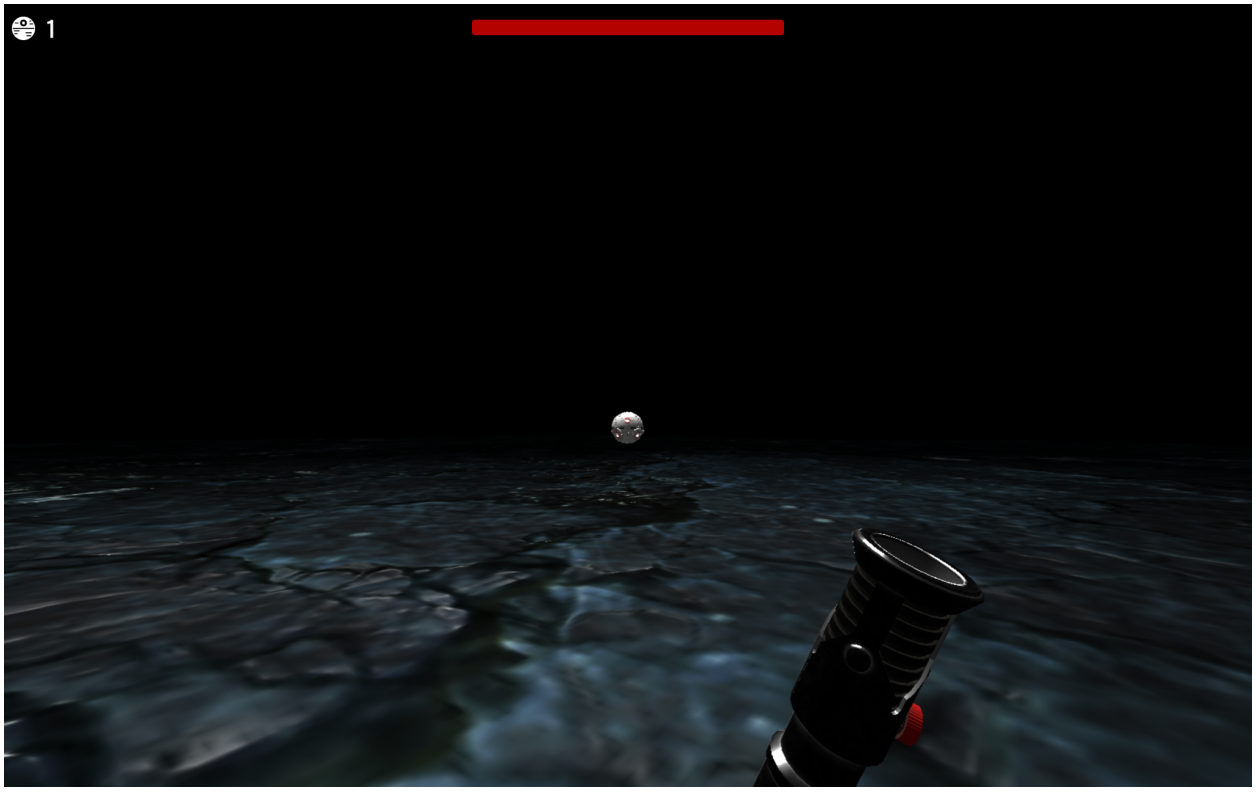
Lightsaber Dojo

By Arti Schmidt and Victor Chu



Abstract

Our project focuses on the construction of a Star Wars themed lightsaber-dojo. In this construction, we pay homage to the training droid that Luke faces in a New Hope (Luke Skywalker's Lightsaber Training - Star Wars: A New Hope 1977). The user learns to time their lightsaber strike in order to deflect the blaster fire in the right direction. The user, however, can only sustain a certain amount of hits before "dying". If the user successfully deflects enough bullets, the training bot will explode and spawn a new wave of robots that contains one more robot than the last wave. The user then tries to survive as many waves as possible.

Introduction/General Instructions

The main inspiration comes from the popular VR game "Vader Immortal". In the game, there is a training mode where users are able to wield a lightsaber and train against a variety of training droids. As we both like Star Wars and are huge fans of the Vader Immortal series, we wanted to emulate something similar using the things we have learned through the semester. While there is a very realistic VR game, we wanted to try and create a similarly immersive Star Wars Lightsaber game outside of VR. Other than "Vader Immortal", the largest Star Wars video game is the BattleFront series which focuses on 3rd person shooting.

Our approach is to use ThreeJs to generate a first person StarWars lightsaber game to create a fun and lighthearted twist to classic Star Wars Games. The success of this project largely depends on finding high quality assets for the assignment. Without visually appealing assets, we wouldn't be able to create a convincing feeling of Star Wars to our players. The gameplay mechanics are of reasonable complexity; therefore, the large bottleneck is the Assets. Luckily, the Star Wars community is very large, so there are many high quality fan made assets online. Therefore, finding the droid and lightsaber came pretty easily and at the very nice price of free.

Moving forward, our version consisted of 3 main components, the user, the droids, and the lightsaber. All of these components move around a simple scene which we create.

Environment and Frameworks

We used three.js because it is one of the most popular graphics frameworks for the web and because it provided a reasonable level of abstraction for this assignment. We also used the Bootstrap CSS framework to a limited extent for some of the 2D UI components which were implemented as HTML elements. We decided to structure the project ourselves instead of using the starter code because we had web development experience. We used the Vite development server and bundler which is very fast and pleasant to work with, and also used TypeScript to benefit from the static type checking and better code completion.

Player

For the player, we sourced a lot of the code from an online tutorial on First Person Controls (https://www.youtube.com/watch?v=oqKzxPMLWxo). In the tutorial, they discuss the usage of head bobbing to make it feel more realistic and a custom input controller to have a more flexible event handler. The source code can be found here: https://github.com/simondevyoutube/ThreeJS_Tutorial_FirstPersonCamera. With this code, we gained basic first person controls. When you move the mouse, the camera angle changes, and with WASD, the player can move through the scene.

From this starting point, we fine tuned a few of the parameters to fit our usage a bit better. For example, we reduced the head bop and added a few more useful fields (like a lightsaber object) in order to improve modularity. We then added a few more features to the Player. For the player, we create a sprint and a jump option to the movement. For the sprint, we check if the shift key is being held. If it is, we multiply the translation of the WASD keys by 3.5. Otherwise, we multiply by 1.5. This drastically increases the speed of the player, giving the feeling of sprinting.

For the jump, we implemented a similar Vetlet integration to that in Assignment 5. A jump happens when the user presses the spacebar. When the inputcontroller sees that the space bar has been pressed, there has been more than 0.25 seconds since the last jump, and the player has not

used more than two jumps before touching the ground, the player jumps on the screen. The jump is created by increasing the velocity by 1.5 (vtdt as in the last assignment) of the user which is stored as a global variable. Then, we had to implement gravity. This was done using the vtdt global variable, GRAVITY, DAMPING, and TIMESTEP constants similar to the last assignment but have been tuned to simulate a floatier "force jump". To calculate the translation, we multiply the vtdt with direction Vector3(0,1,0) with the 1-DAMPING and add it to the acceleration of gravity multiplied by TIMESTEP squared. Finally, we decrement vtdt by 9.8 * TIMESTEP divided by a scalar (which is mostly physics motivated although the force is not physics-y). The following image shows the user mid-air.

Droid (/src/Droid.ts)

For the droid, we load the droid assets and textures that we found online using a MaterialLoader and an OBJLoader. In the droid class, we have vec3 target where the droid will be moving towards, a waitTime before the droid moves again. The droid's position is initialized to a random position around the user 25 units away and with y-value above 4 and below 10 so that the droid doesn't spawn in a place the player can't easily see. This random position is done similar to the soft shadow we did in assignment 4. We randomly sample a vector in R3 and then normalize it (as seen in the end of this website: https://mathworld.wolfram.com/SpherePointPicking.html). We then scale the resulting vector by the radius of the sphere we want. Then we add it to the current user position and clamp the y-value in order to make sure it doesn't go into the floor and not too far above the player's head.

In the droid class we also have a moveDroid function which calculates the movement of the droid. If the droid is too far away from the user, it will immediately move towards the user by setting the target position of the droid to the user position and moving towards it until it reaches a certain threshold. If it is within the threshold, it will calculate a random waitDuration and accumulate waitTime until it reaches waitDuration. If the user moves significantly away from the droid, the waitTime will be set to zero. If waitTime reaches waitDuration, the target is then set to a new random position relative to the user and the droid then moves towards that target.

Lightsaber (/src/Lightsaber.ts)

For the lightsaber, we load the hilt asset that we found online using a GLTFLoader. We then overlap that lightsaber over a capsule geometry that acts as the "blade" of the lightsaber. We then create a group and add both the lightsaber hilt and blade to this group to become the complete lightsaber.

Then for the lightsaber update, we need to keep track of two things. Whether the light is on or in the process of turning off/on and whether there the lightsaber is swinging. For the lightsaber toggling, we have a method called toggleLightsaber that is called when the "f" key is pressed. If the toggling variable in the Lightsaber object is set to false, we set it to true otherwise we return. Then we toggle the on variable and play the respective sound (i.e. if the lightsaber was on, the off sound will play). In the update method, if the toggle variable is set, depending on the on state of the lightsaber, we move the capsule geometry up or down in the y-direction relative to the hilt and set the visibility accordingly. When the lightsaber is in one of its "threshold areas" (on position or off position) relative to the hilt, we set the toggle to false and end the toggling.

For the swing animation, we create a handleMouseDown() method for the lightsaber that will be called in the /src/main.ts as the mouseDown event handler. In the lightsaber handleMouseDown() method, if swinging is false, set swinging to true. Otherwise, we don't do anything. In the lightsaber update, if swinging is true, using the delta time given in the update function, we accumulate a "swingState" field which is incremented by delta/swingDuration. This ensures that the swing animation will take swingDuration time and be independent of the framerate. The swingState variable defines the point of the swing that the lightsaber is currently in between 0 and 1, and we use this to smoothly interpolate between 0 and 1 using the three.js smootherstep function. Using this value, we linearly interpolate the lightsaber's angle of rotation to produce a swing animation. When the swingState is at 1, we finish the swing and set swinging to false and swingstate to 0. We position the lightsaber relative to the camera position and set its initial rotation so that the swing animation looks like a natural human arm movement.

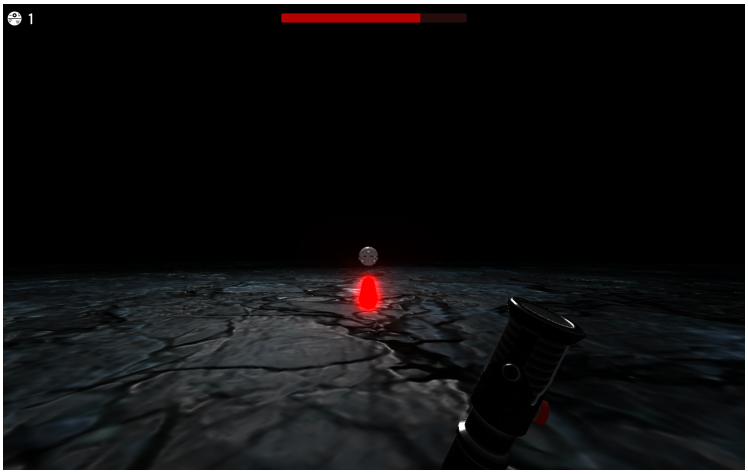
Scene (/src/TestScene.ts and /src/main.ts)

For the Scene, we created a simple dark scene that is inspired by the mirror cave in Ach-To:

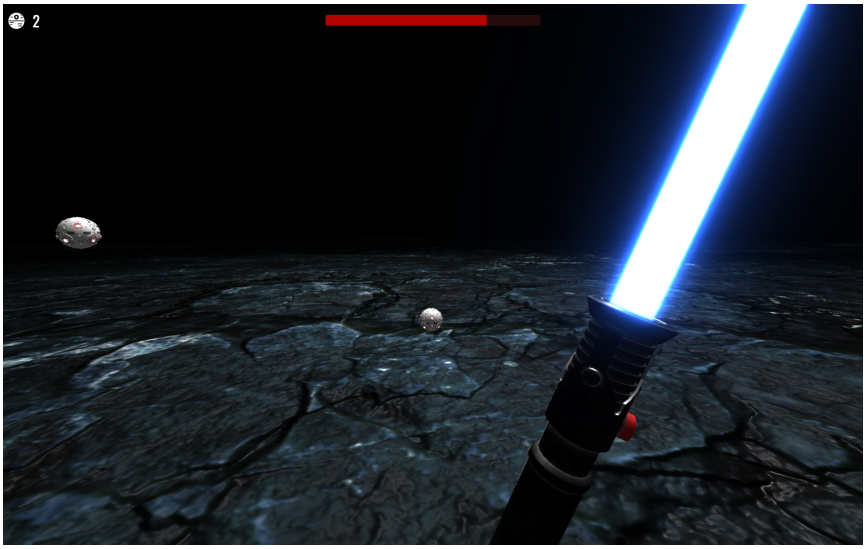
For the scene, we found a black rock texture that we liked and applied that to a plane positioned at y = -1. Then we added two lights, one ambient white light and another white spot light positioned above the player producing a cone of light below.

Next, we create a lightsaber object and a set of droid objects. We pass in the Lightsaber object into the player object so that the player can easily interact with the lightsaber. In the scene class, we also handle the lightsaber and blaster fire interactions as well as the bloom rendering. While the interactions may not be best suited for this section in terms of modularity, because of time constraints, this was a hack to cut a few corners. For blaster fire interactions, we store all the "bolts" which are the red blaster fire shots in the scene. For each droid, we set a random interval of shooting. Randomly between 2 and 8 seconds, the scene will create a bolt, set the position to the droid position, play the fire sound, and send it along the path to a little below the current camera position (which is the player's position in a first person system). It is set to a little bit below because this improves the depth perception of the bolt as it moves closer to the player.

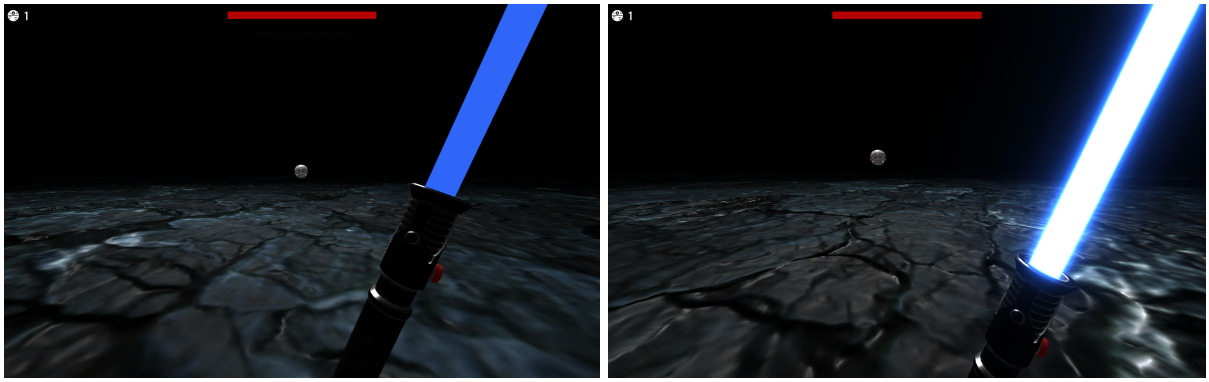
Once the bolt is in the air, we have to deal with the deflection mechanism which is how you defeat droids. If the bolt is in the air and gets within 2 units of the camera position and the user has clicked to wave the lightsaber, the bolt will get deflected 180 degrees. This is accomplished by simply rotating the bolt so that as it continues traveling along its positive z axis, it will now be directed toward the location where it was shot from. As long as the droid has not moved yet, it will hit the droid and destroy it. Initially, we considered making this more complex to make reflections more physically accurate; however, after playing with this mechanic, we found it too hard for the gameplay. In the wave based gameplay, it is hard enough to kill the droids and avoid the bolts as it is. If it gets to within 0.75 units of the player position, then the player takes damage. We create a spherical hitbox by calculating the bolt's distance from the camera position every frame. If it is within the hitbox, we decrement the damage.



If a droid gets hit by a deflected bolt, we “kill the droid” by adding it to a set of “dead droids”. The dead droids are now subject to a similar gravity function seen in the player class and falls to the ground. It no longer fires and gets removed from the scene after 5 seconds. Below, you can see the dead droid on the floor in front of the player as there are two new droids spawned beside the player.



The final component of the scene involves the rendering of the lightsaber. To do the glow effect for the lightsaber and the bolt-fire, we use the bloom effect. Bloom is a post-processing effect that only exists in the camera space. The logic for this effect largely follows the logic defined in the three js selective bloom example. The bloom effect is done by having a bloom pass calculated by threejs followed by a second render. Before the bloom render, we set everything but the bolts and lightsaber to a black texture and then calculate the bloom. The bloom render is then saved to a texture in the renderer. This texture is mixed with the normal renderer in a final effect composer called “finalComposer”. This resulting visual effect is the lightsaber. Below, we see the contrast between without bloom and with bloom on the lightsaber visuals.



UI elements

We created a simple title screen that is displayed to players before they start the game, containing the “Lightsaber Dojo” title and instructions for how to play. We also created an end screen that is displayed when the player dies, showing their score (the wave they made it to) and giving them an option to restart. While playing the game, an overlay is displayed showing the number of droids in the current wave in the upper left of the screen (with a droid icon found in an icon library), as well as a health bar centered at the top of the screen. It shows the amount of damage the player can take before dying, and is animated for visual appeal and to draw attention to it when damage is taken. It is implemented as a progress bar from the Bootstrap CSS framework. All of these UI elements are HTML elements displayed with fixed position relative to the three.js canvas and controlled via JavaScript.

Sound

We added sound effects and background music to the game for added immersion and entertainment. We created our own sound effects for events like turning the lightsaber on and off, ambient lightsaber sounds, swinging the lightsaber, deflecting bolts, firing bolts, hitting droids, taking damage, and moving. We load these sounds and play them at the appropriate times using three.js. We also play “Duel of the Fates” on loop at reduced volume as background music while playing the game.

Evaluation/Feedback:

As our evaluation, we asked many of our friends to play and give feedback on the game. We would show them the instructions and explain the game mechanics. Afterwards, we asked a few questions on the feel of the game and general feedback on what we should improve on. This process was incredibly rewarding not only in the interview portion, but also in seeing them play the game. They would initially get hit, not feeling the sweet spot at first, but they would slowly gain an intuition of the game. In addition, we saw many strategies for the game which later shaped some of our development choices. We saw spammers, campers, and evaders, all of which we ensured had their benefits and drawbacks. In particular, we made sure that spammers were

not completely overpowered by ensuring that there was some lag between lightsaber strokes, ensuring that the deflection window could not be always on.

In the interview section, users astoundingly enjoyed the sound created by our foley artist staff. Most of the feedback came from the lightsaber feel and the bolt speed. As the developers, we had gotten accustomed to how they felt with our default parameters so we didn't initially realize this problem. By listening to the feedback, we were able to retune the parameters and saw a great improvement in follow up playthroughs by other users, adjusting to the sweet-spot much faster than the previous testers.

As far as measuring success, we had a few goals for ourselves when starting the project. We more than accomplished the MVP and had lots of fun adding small easter eggs like making the sounds and playing with the lightsaber effects. In addition, most importantly, we really have fun playing our game–even though it still has a few minor bugs. Overall, with our deliverables and peer feedback, we feel like we have successfully achieved our goals.

Challenges

As mentioned in the feedback section, the hardest part of the game was tuning the deflection scheme to feel satisfying and realistic. For the deflection scheme, we had to carefully tune parameters in order for the deflection zone to feel large enough and not interfere with the hitbox. One problem we encountered was the character would take damage while also deflecting the bullet. Another problem we had was that it was hard to get a sense of the range of the "deflection zone". While we understand this is the main challenge of the game, we also felt necessary to make it feel as natural as possible. By tuning the parameters and the speed of the bolts/swing, we feel like we achieved a reasonable realism to the lightsaber/bolt mechanics.

Future work

In the future, we would like to have a larger variety of textures/environments to the game. In addition to the cave at Ach To, we would want other classic Star Wars environments like the Death Star, Hoth, or others. In addition, more character customizability (like changing the lightsaber customizability) would make the game more entertaining.

Other ways of making the game more engaging would be a better hit animation. While the combination of the health bar changing and a clear "hit" sound conveys the sensation of being hit, a hit animation where the screen flashes is a staple for many first person games that is not in our game. Another possible improvement would be to have even more complex deflection/hitbox schemes to make the game feel even more realistic and give the player even more freedom to destroy the training droids. This could, for example, allow players to kill droids with bullets that didn't come from the droid that you are targeting.

One other interesting addition would be to make more interactive objects in the scene. For example, we could have walls that sprout from the ground and move around, giving the player a dynamic environment that they could use to evade bullets. The bullets would not be able to go through these walls, allowing the user to get cover.

Another minor problem we never had time to resolve was that sometimes the droids overlap in the scene. We could fix this by making intersection checks and ensuring that they are far enough away from each other, but this would be another minor improvement that would greatly improve the realism.

Our project provides a very good foundation as a Star Wars lightsaber-combat game; however, there is still much more potential left to explore to make the game even more fun and engaging.

Contributions

Breakdown of each members' contribution

Arti:

Framework + UI

Droid and Lightsaber

Scene

Victor:

Player mechanics

Sounds

Assets+Textures

Bloom

Models:

Training Droid:
https://sketchfab.com/3d-models/jedi-training-droid-a9c9e3476eb54507978d3a6cf68e6ee2

Lightsaber Hilt:
https://sketchfab.com/3d-models/obi-wan-kenobi-lightsaber-50cb1f907d18485e9615b16708a3fbbb

Floor Texture:
https://ambientcg.com/view?id=Rock035

Code used for Inspiration:

https://github.com/simondevyoutube/ThreeJS_Tutorial_FirstPersonCamera

https://github.com/mrdoob/three.js/blob/master/examples/webgl_postprocessing_unreal_bloom_selective.html

https://github.com/efyang/portal-0.5/blob/main/src/audio.js

Background Audio

https://www.youtube.com/watch?v=snpS8Qa42nc

Foley Artists

Victor Chu, Arti Schmidt, and Edward Yang