Catcher

Jake Kirkham, Zoya Shoaib, Yujin Yamahara, and Max Fu
COS426 Computer Graphics Final Project Report
Advised by: June Ho Park

Abstract

In this project, we create a game centered around catching falling objects from the sky. Inspired by the current COVID-19 pandemic, the game's player catches falling PPE and vaccine components while avoiding falling virus particles. In this report we detail how our game evolved from the starter code provided to the bare bones version presented as an MVP and finally to the final product demonstrated in the live demo. The report also details the multitude of features implemented by our game and the various obstacles that we encountered in developing them. Throughout this process, we were successfully able to create a game where the player receives points for catching falling physically-simulated objects while avoiding physically-simulated virus particles.

1. Introduction

When formulating ideas for our final project, we were inspired by the current COVID-19 pandemic in multiple ways. We were excited by the prospect of building an interactive web game because of the sheer number of people stuck at home during quarantine. Video and computer games have become a welcome distraction for many during these unprecedented times -- Verizon has observed an increase of gaming on its networks by 75% [1]. We wanted to contribute a game that people, especially those stuck at home with nothing else to do, could enjoy while stuck in quarantine. Catcher is directly inspired by COVID-19: the goal of the game is to successfully avoid falling COVID-19 particles while gathering key components of a vaccine. While the premise is simple, we hope that people stuck in quarantine find solace and distraction in our game.

Games based on catching falling objects from the top of the screen are immensely popular both to build and to play. There are many examples, such as iOS game Scoops where the player catches falling scoops of ice cream with their cone while avoiding other falling objects, that are similar to our game; we both take inspiration from and improve upon these existing games. Often, these simple catcher games are 2D, and lack the cutting-edge graphics employed by many games today. As such, the main point of novelty in our game is a 3D, physically simulated game environment. In addition to this, we improve upon existing games by adding more complex graphics through textures, lightning, and shadows to instill realism while also enhancing gameplay. We hope that these improvements will lend more depth to the gameplay experience and still stay true to the popular catching game formula.

We build off the provided three.js [2] framework with a variety of other frameworks and novel implementations. Our approach utilizes Javascript, HTML, and CSS to create exciting and fun visuals and objects. These languages both allow us to build off the concepts and implementations explored in this class and provide an environment that everyone has some experience and confidence in.

2. Methodology

Our game relies upon a variety of mostly independent underlying systems connected by a top-layer of gamification. Our overall vision to start was a physics-based, 3D game with interesting, fun visuals that tie into the gameplay. This required the development, implementation, and integration of both fundamental simulation systems -- lighting, physics, etc. -- and gameplay systems -- controls, scoring, content -- that collectively make sense as a cohesive game. We defined each subsystem as needed and divided the implementations into independent sections that could be distributed amongst the group.

FEATURES:

1. Physics

Physical simulation of a chaotic 3D environment is the backbone of the game as we originally envisioned it. Given the difficulty of implementing a 3D physics engine that is accurate and optimized sufficiently to be used by a game, we opted to use a publicly available physics library rather than rolling our own. As most physics libraries must be implemented into the basic definition of the scene, this decision was our first priority. This decision offered a variety of choices; initially, we elected to use Physijs due to its easy integration with Three.js, as well as its easy interface (compared to, for instance, cannon.js). However, the current version of Physijs is no longer compatible with the latest Three.js. After a brief struggle with cannon.js (which was essentially too complex for our needs), we settled on oimo.js, a fairly simple but sufficient library with enough example code to give us a starting point [4]. In retrospect, cannon.js may have been a better option -- the physics simulation of oimo is sub-par, and the documentation is hugely lacking. However, we charged on with Oimo. To start, we re-implemented an Oimo falling ball example in node; unfortunately, this took much more doing than expected, as the node support is lacking and we encountered lots of bizarre bugs, such as wonky sizing and invisible objects. After much effort, we finally got a basic scene -- a plane with physically simulated spheres bouncing onto it -- running. This physically simulated environment provided the foundation we needed to build our MVP.

2. **Player**

To make our physics simulation a game, we needed one thing -- a player. Adding a player created its own challenges: to be a catcher game, the player needed to be smoothly controllable, physically simulated, and be able to physically catch things. Initially, we tried to use Babylon.js to create a bucket; however, we encountered many problems integrating physics with the babylon object. We then switched over to work with the OIMO physics engine and worked backwards to create a player. We wanted the player character to be a scientist with a huge beaker to catch particles in; however, OIMO lacks any sort of collision modelling for arbitrary meshes, ruling out the idea of a modeled player character. Instead, we needed to define a player character using only 3D primitives -- cubes, spheres, and cylinders. As such, we chose to make the player a moving cube beaker, as it allowed us to still physically model the player character in a way that made sense thematically and gameplay-wise. We could have associated a different visual model with a simple physical implementation, but we were concerned that any mismatch between physics and visuals would hurt the player's ability to actually catch things.

Our scene now contains a plane, with a big bucket in the middle and falling spheres. We need to make our player controllable, while preserving the physical simulation that defines the game. This sounds easy, but is in reality tough because OIMO has very little methods to directly control or manipulate physics objects. As such, we had to come up with some very hack-y implementations based on the (extremely poorly documented, minified) OIMO source code. This eventually resulted in letting the player object be physically simulated, but resetting its position and rotation each tick to avoid weird control issues. We implemented simple key up/down event handlers for WASD, which add physics velocity in the appropriate direction to the player object. This allows us to implement smooth movement using our physics engine, and also let us fine tune acceleration and deceleration for gameplay purposes (we initially simply added to the position of the player, but this was jumpy and felt weird). Having good movement was vitally important, as proper movement determines if a player is good or bad. We fix a ~45 degree camera angle aimed at the player, and update the x,z coordinates of the camera to track the player object. Since we have smooth acceleration, the camera is fairly smooth and non-bothersome despite being hard locked to track the player. We elected not to implement a more complex, bounding box soft-tracking approach, as the current effect was sufficiently smooth for our purposes. Using this, we now have a physically simulated beaker that can be moved smoothly around the plane with WASD to catch falling balls.

3. **Scoring**

For scoring there were various options we could have implemented. These include creating a beaker that fills up, having the player's body fill up with liquid

correspondingly with the points, or creating a simple div to display the current score. We ended up choosing the third option because we decided that the other features–physics, player mesh, and functioning gameplay–were of higher priority than the score counter.
In the backend, the score is calculated by checking the material of the object that collided with the player. We created a blue material and a green material for our objects. The virus is modeled over the green material. Once the object collides with our player, we check if the object's material is blue or green. If the material is blue, we add a point to total winning points. If the object's material is green, we add a point to total losing points. In the end, the UI displays losing points subtracted from the winning points.

4. **Visuals**

The naive approach was to create a static 2D background as the background for our game. Given this was a Graphics class, the 2D background didn't meet our own expectations for the game. We wanted to create a 3D scene that would contribute to the theme of our game. We initially started with wanting to create a 3D lab scene as our background. In order to do so, we decided to use a CubeTextureLoader. We found six images from the same lab and tried to use that for six sides of our cube. However, since the images were not connected, our scene was disconnected and worked as a distraction away from our game. We then decided to use existing skyboxes as our background. We tried to find a skybox that would connect with the theme of our game; however, most existing skyboxes created scenes of skies, beaches, and space. We found an open source skybox generator (https://wwwtyro.github.io/space-3d/) and generated random seeds until we found the one that visually connected with the colors in our theme such as the virus. Once we had the images for our skybox, we called CubeTextureLoader to create our scene.

5. **UI**

For the user interface, our main goals were to create starting and end screens for the game. We created a start screen which provides the player with instructions on how to play the game. After the player reaches 10 points, the game ends, and the end screen shows up.

The two main ways of implementing the UI were to either inject divs with the stylesheet modifications through Javascript, or create HTML and CSS files and link that to the game using a HTML Webpack Plugin. The first approach would be somewhat time-consuming and more unwieldy than simply creating separate HTML and CSS files. I first attempted using the Webpack Plugin, but had issues integrating the index.html file's display with the canvas where our game was situated, and therefore switched to the first approach. It turned out that injecting HTML and CSS styling using Javascript was not too inconvenient, and I created the start and end screens using this approach.

The one main challenge I faced with creating the UI was the time component of our main Javascript code. Figuring out where I needed to put my code that creates new DOM elements within my Javascript code was more difficult than I expected, as I needed to have a strong understanding of when each part of the Javascript code executed.

6. **Lighting**

For the lighting of the scene, we used DirectionalLights and HemisphereLights to lighten up the screen. DirectionalLights are a three.js object that projects lights in a specified direction. There are multiple options for lights with three.js, such as SpotLight and PointLight, but we opted to use DirectionalLights over these counterparts as we can explicitly choose which DirectionalLight casts shadows and projects onto the ground of the scene. A DirectionalLight or SpotLight was preferred over a PointLight as casting shadows with a PointLight is much slower and would be the equivalent of casting 6 SpotLight shadows because a PointLight shines in all 6 directions. The DirectionalLight also had more flexibility in fine tuning the way shadows are cast over SpotLight in which most of the settings are automatically set. HemisphereLights were also used to simulate lighting from the sky and ground in the scene and works well in conjunction with another light in the scene. The HemisphereLights were important to highlight the texture and detail of the falling particles and in particular the virus' texture and its colors.

In terms of casting shadows, there were a multitude of ways to do this. The lighting choices above were made with shadows in mind so we decided to opt for casting shadows using the built in capabilities of the lights in three.js over projecting shadows based on our own calculations. While this offline method of projecting shadows onto the ground through calculations may prove to be computationally faster, we opted to use the built in capabilities of casting shadows for ease and avoidance of unexpected bugs. Another option was to cast "fake" shadows onto the ground by gray scaling the objects and drawing the approximated shadow texture onto the ground below the particles as they get closer to the ground. This was an interesting option, but we ended up going with builtin shadows in the hopes that it would be more realistic.

The shadows themselves were cast from one DirectionalLight to avoid multiple lights drawing shadows onto the scene and slowing down our game. Shadows were cast by assigning the value to be true for the light and renderer and choosing which meshes within the scene would cast or receive shadows.

7. **Gamification / content**

With an MVP implemented of physics, scoring, and a player character, we can expand our game to be more thoughtful with its gameplay elements. This entails a variety of mechanics: spawning protective elements (masks, etc.) that make the player immune to negative points, varied spawn rates, and more visual content in the form of both terrain and objects. These considerations are, for the most part minor: a naive implementation is surprisingly effective at convincing the user that the gameplay is tuned to give a fun experience; most of these are simply implemented by tweaking parameters and adding new models. We use models from Google's Poly, in particular for our virus model. There is a bit of nuance here in the interaction with the physics: while our model shows a complex virus, OIMO is only capable of simulating primitive objects. As such, while our virus model appears complex, its "hitbox" is really just a sphere. This compromise is quite effective at convincing the player that everything is actually simulated: even if it is fairly rudimentary, it is good enough. Our level design and content design, while still executed thoughtfully, has been kept fairly minimal to allow the fundamental gameplay elements -- physics, visuals, etc. -- to shine through.

3. Results

We measure success based on the implementation of required elements, the inclusion of reach goals, and gameplay testing. Our minimum implementation required 3 elements: a movable player character, physical simulation of both player and falling objects, and gamification through scoring of good / bad particles caught. While these requirements posed more difficulty than expected, we were able to achieve them to a reasonable degree of success fairly quickly.
Beyond this there were a variety of stretch goals: making the game "pretty," increasing our physics fidelity, adding more content, and making lighting elements that make sense with respect to gameplay. We fully achieved some of these goals: inclusion of UI elements that make the game pretty and more professional, backgrounds and lighting that add a degree of visual flair, and adjustable physics that have increased our fidelity. The remaining elements, such as shadows to make gameplay easier, are implemented in some limited regard, but could perhaps use more polish to fit more seamlessly.

We had some informal playtesting, but we felt that the concept was straightforward enough to not require a large playtest campaign. As we are building off an established game concept, the main issues are technical and visual; currently, our gameplay elements are far enough along that playtesting would not be worth the organizing effort to determine areas of relatively small improvement.

4. Discussion

Our approach has been fairly successful in reaching our basic goals. Our MVP, as described above, was achieved and actually works fairly well as a physics-based catcher. Our physics, while as discussed above are not ideal due to OIMO's limitations, are fairly realistic to the point where a player is reasonably capable of predicting the behavior of falling objects. As such, the game is in a state that is more complete and accurate than a wide variety of its peers. Furthermore, we were able to implement a 3D approach that actually does work in a fun way, while also adding some additional challenges for the player over similar 2D versions.

As discussed above, a different physics library (or a custom implementation) would have made this entire project much easier. Our choice of OIMO ended up providing lots of limitations that needed to be hacked around to provide the features that we had envisioned. Similarly, our use of Three.JS with OIMO gave less control of lighting than we had hoped for -- as a result, we needed to do much more work on small, seemingly non-essential tasks (such as shadows) that really should have been handled by Three itself. However, the built-in implementation hurt playability, requiring us to essentially retread ground that should have been covered already.

The result of the above issues is that we were less able to achieve a level of visual polish than we had originally hoped. We originally chose a fairly simple gameplay idea as it was going to give us plenty of time to focus on visual polish and gameplay specifics. However, the lacking library support meant that a lot of time was put towards basics, many of which are very important gameplay-wise but don't add much that is interesting from an outside perspective. As such, just getting the very foundation running hurt our ability to give the game the level of polish and artistic flair that made us excited about the project to begin with.

5. Conclusion

Despite the issues we faced, we believe that the game is in a fairly good state overall. We implemented a working catcher game that allows the player to move around a 3D environment and catch physically simulated objects -- that was our basic aim and we achieved it. Furthermore, the basic gameplay systems are solid enough that it is actually fairly fun to play -- movement is smooth, visuals are fairly interesting, and physics make sense.

More can certainly be done: as discussed above, most of this comes down to visual flair and content. Ideally, we will add more gameplay elements (such as masks) and add to the level environment in a fun way that adds some challenge to the game. Overall, we could use a level of polish that removes some odd artifacts, both visual and physical.

While the game isn't perfect, we are excited that we've made something that a player in quarantine can immediately understand and play. We hope to add that layer of polish soon, so that our players can have a fun, smooth, addicting experience, but until then we are overall happy with how our game currently plays and looks.

6. Contributions

Jake -- Physics, Player, Camera, Controls, Models / added content
Max - Lighting/Shadows, Other gamification (still in progress)
Yujin- Start Screen, End Screen, Scoring UI
Zoya- Initial Babylon Player (discarded), Scene, Score (backend), Timer

Works Cited

[1] M. Gault, “It's Fine to Play Lots of Video Games in Quarantine,” *Time*, April 23, 2020. Accessed May 12, 2020. https://time.com/5824415/video-games-quarantine/
[2] “three.jsr116,” *three.js – JavaScript 3D library*. Accessed May 12, 2020. https://threejs.org/
[3] Chandler Prall, “chandlerprall/Physijs,” *GitHub*, October 20, 2015. Accessed May 12, 2020. https://github.com/chandlerprall/Physijs
[4] lo-th, “lo-th/Oimo.js,” *Github*, January 21, 2019. Accessed May 12, 2020. https://github.com/lo-th/Oimo.js/