Crispy Combat: A Two Player Battle Royale

The Writeup

Michelle Liu, Donna Wang, Lillian Ye

Abstract

Our project, Crispy Combat, is an interactive, 2-player game where each player's objective is to remain alive for a longer time than their opponent. The theme and environment of our game is inspired by the frequent chaos of Princeton's late meal dining. Thus, players in Crispy Combat must avoid moving tables and chairs by using their keyboard to move in 5 possible directions: left, right, forwards, backwards, and up (by jumping). In addition, players also have the opportunity to use either the space key or the "m" key to pelt chicken drumsticks at their opponent, who must then dodge the chicken in order to stay alive in the game. Player movements are restricted within the area of the floor. Players are allowed to bump into each other, but if they are hit by the opposing player's chicken drumsticks or if they bump into furniture, then they lose.

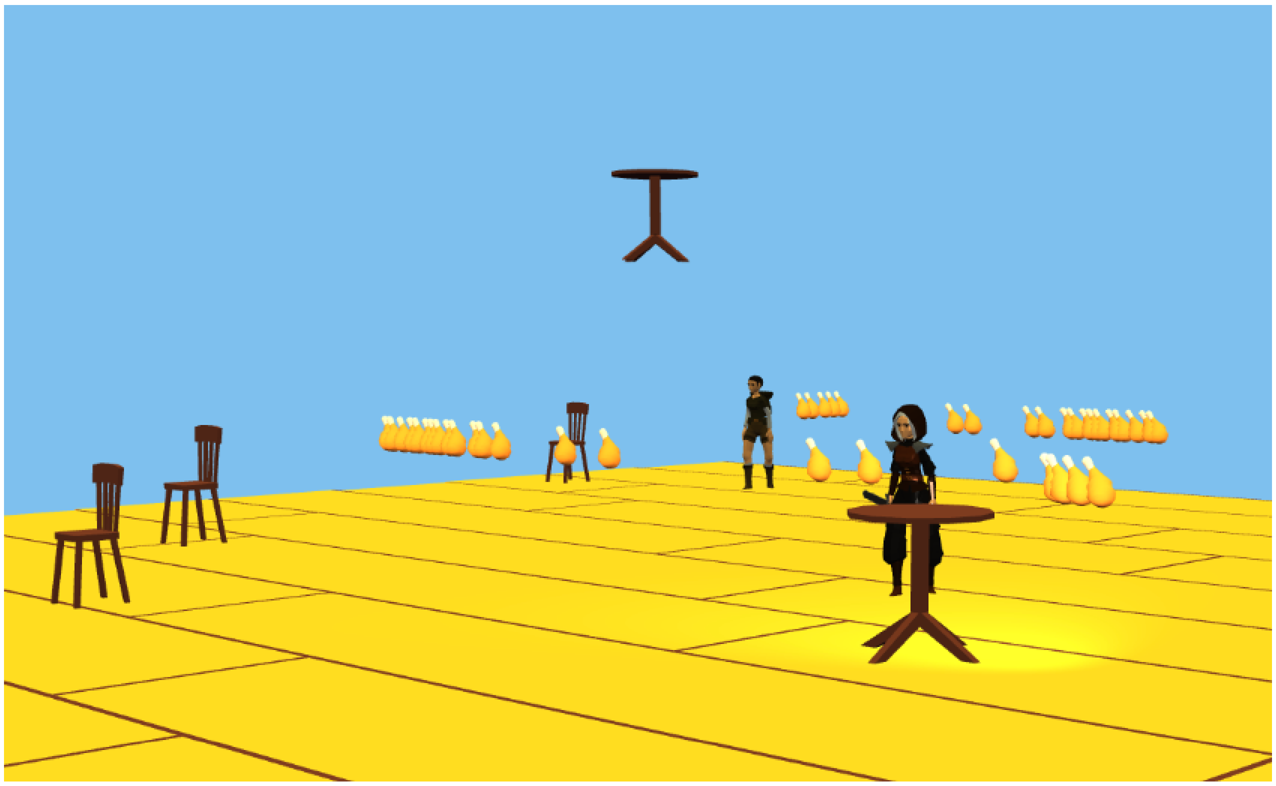
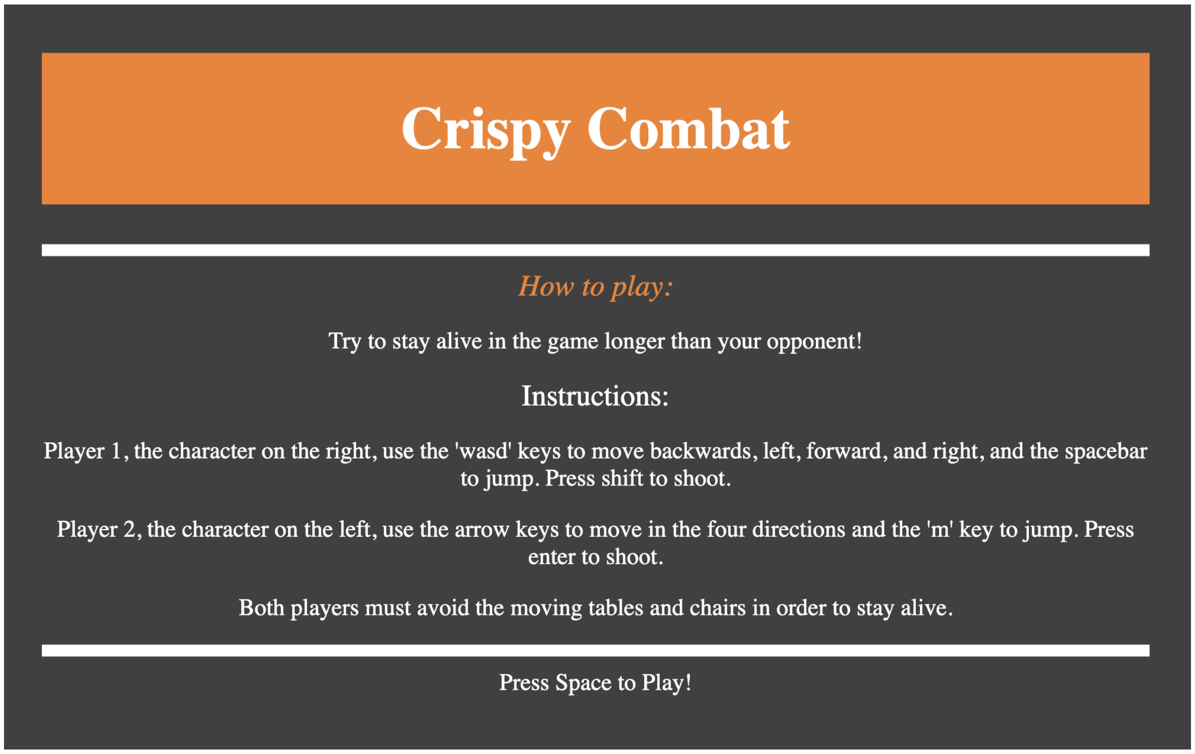


Fig: Intense mid-battle action shot of two players neck-and-neck in a cutthroat match of Crispy Combat.

Introduction

Goal

The goal of our project was to take the classic 2 player shooting game and introduce our own creative twist using a variety of unique obstacles, themes, and features. We wanted to incorporate aspects of our shared experiences at Princeton and apply them in a way that makes our game more engaging, interesting, and exciting. The three of us all agreed that an integral part of our freshman and sophomore years involved enjoying Frist late meals while forming memories with our friends and getting to know them better. Thus, we wanted to create a project where our users would not only experience an interactive and fun game, but also be reminded of a core, amusing component of their Princeton years. As it gets harder and harder for people to spend meaningful time together in this day and age, we wanted to use both our newfound knowledge of graphics and game design in order to build the gameplay mechanics of local multiplayer, which would achieve our goals of building genuine connections between people, similar to how late meal does in real life. As all undergraduate Princeton students can relate to spending their time as underclassmen at Frist late meal, Crispy Combat is a game that all students can enjoy and benefit from.


Crispy Combat

How to play:

Try to stay alive in the game longer than your opponent!

Instructions:

Player 1, the character on the right, use the 'wasd' keys to move backwards, left, forward, and right, and the spacebar to jump. Press shift to shoot.

Player 2, the character on the left, use the arrow keys to move in the four directions and the 'm' key to jump. Press enter to shoot.

Both players must avoid the moving tables and chairs in order to stay alive.

Press Space to Play!

Fig: Startup and instructions page.

Previous Work and Inspiration

It was 10:34 PM on a dark rainy night in Frist Campus Center, the night before the idea submission form was due. The three of us sat huddled on a dreary third floor couch, the steady pulse of the clock ticking down as we continued our soul-sucking search for a unique, fun, innovative game idea that would be a delicate synthesis of all the skills we had learned in COS426 this semester and our own undying passion for game design. We first floated ideas within the realm of single-player infinite travel games, similar to Temple Run — however, this felt awfully dry and overdone. We sat there forlorn, one of us shoving down another Frist Gallery chicken tender when it suddenly hit us all at the same time: Late Meal.

Our main problem with the single-player infinite running game was a lack of innovation and sense of community. With the new idea of a local multiplayer game, we realized we were able to build community through competition all at the same time, truly engaging players in real life through the platform of the game. Silly obstacles such as moving chairs and tables, and weapons such as fried chicken, would create a fun atmosphere reminiscent of our own experiences eating yummy unhealthy food together.

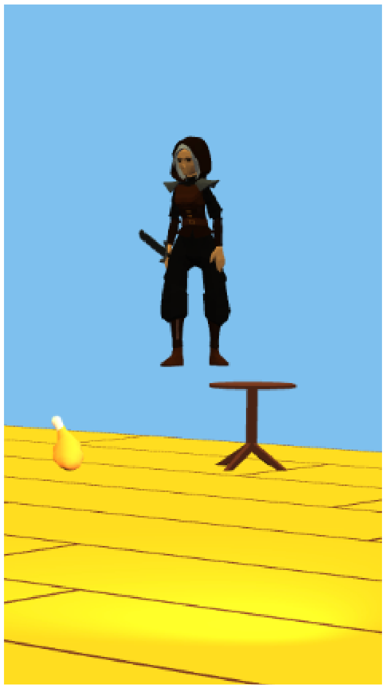


Fig: Player 2 jumps to avoid an incoming drumstick.

At first we came up with the idea of making the infinite travel game local multiplayer, either as a distance/time race or a last man standing knockout competition, but we soon realized that it would spark more connection to have both of the players in the same scene in a direct combat. In solidifying the game mechanics, we were inspired by the popular fighting video game, Super Smash Bros, where multiple players on a platform compete against each other by trying to take other players out first. However, unlike Super Smash Bros, which only allows players to move in two directions (in addition to jumping), our spin on the game gives players the capability to move in four different directions (on top of being able to jump). Further, rather than going with the 2D design of Super Smash Bros, we decided to implement the game in 3D space to further increase the degree of complexity within our game as players would have a wider range of motions. In addition, beyond watching out for the opposing player’s drumsticks, players must also watch out for moving obstacles (tables and chairs), which further adds a layer of difficulty and engagement while testing players’ coordination.

Approach

Our minimum viable product was to have a flat platform with 2 different players who can freely move in 4 directions and shoot chicken drumsticks at the opposing player in order to eliminate their opponent. We also wanted to add obstacles, which within the theme of Princeton Late Meal, were represented by tables and chairs.

For our stretch goals, we thought having the obstacles continuously move would both add complexity and excitement to the game and mimic the chaos of Late Meal. We also wanted users to be able to pause the game once they started playing, where the paused screen displayed the rules of the game (in case players needed a reminder) but also still showed the status of the game in the background. Lastly, we thought that allowing the players to jump onto a table or chair and be taken along the direction the object is moving would also make the game more fun for users.

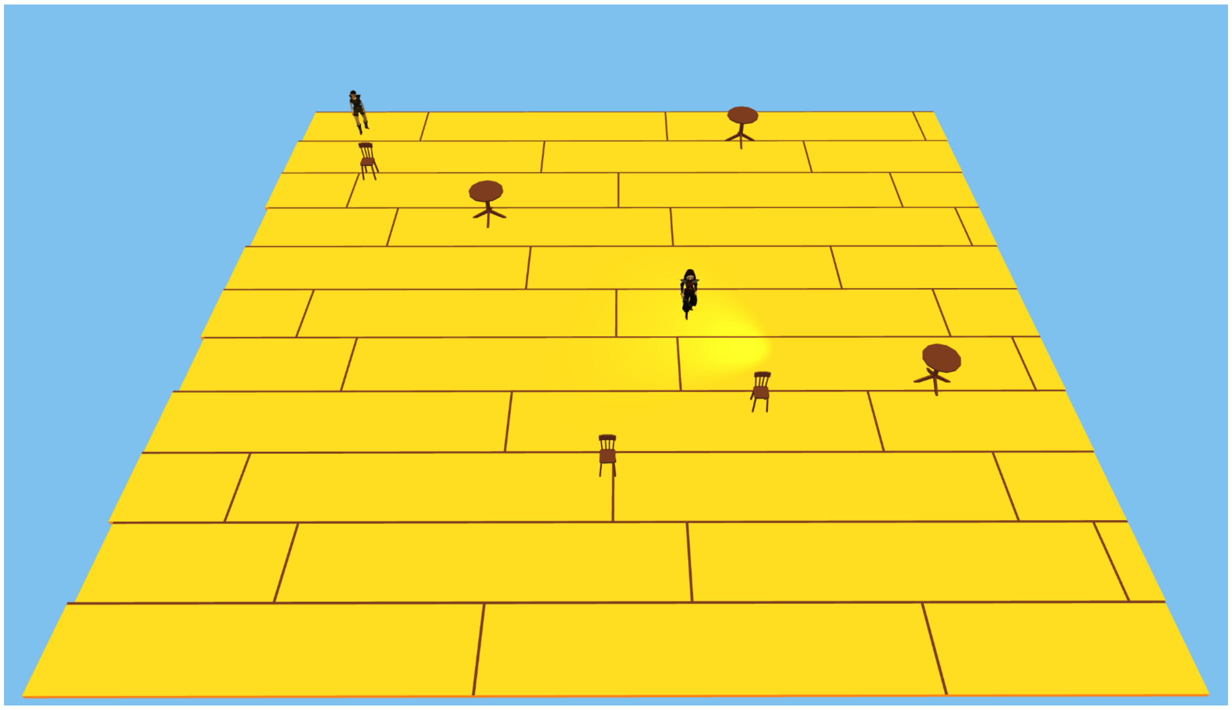


Fig: The game scene.

Methodology

User Controls

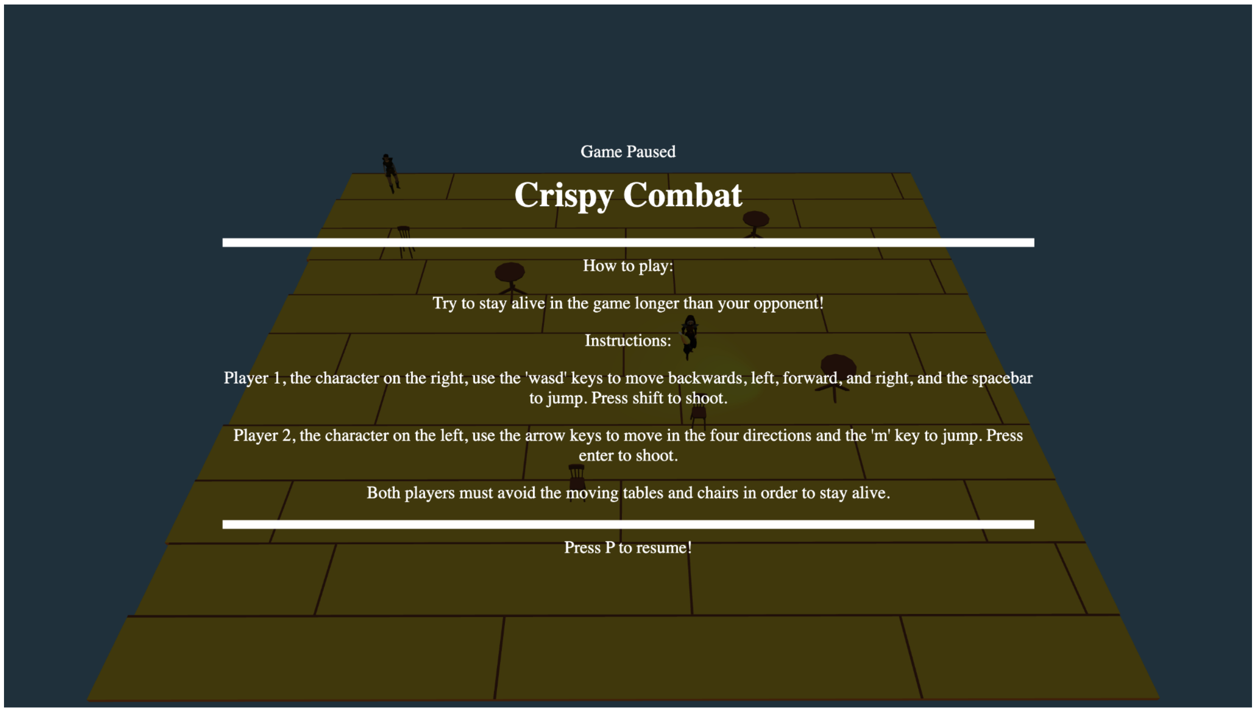
Our program involves two players acting at the same time, and each player has 6 possible actions: 4 of which consist of player movements on the land (i.e. moving forwards, backwards, left, and right), as well as jumping and shooting. Two players are able to simultaneously interact with the game's interface because each player is assigned a different set of keys as their controls.

Player A controls their character using the up, down, left, and right arrow keys, while Player B controls their character using the W, A, S, and D keys. They each correspond to movement into and out of the screen, and left and right of the screen. This is implemented using a handler that checks if the relevant keys have been pressed at the time stamp. If so, we first check if the player is within bounds and then move the player accordingly. To simulate smooth movements, we want the player to move for as long as the key is pressed and only stop when it has been released. We initially wanted the players to be able to turn so that they are facing the direction they are moving in. However, due to the way the player models are designed, we found that we could not simply rotate the player model because the player models themselves are not centrally aligned with the box they are in. Thus, rotating them actually moves them in a circle as well. This would involve further testing in order to find the correct axis, and since the focus of our game is more on the interactions between objects, we ultimately decided that this was not a very high priority.

Players are also allowed to jump a limited amount into the air (or in the z-direction, up and down the screen). Player A uses the M key to jump, while player B uses the Space key to jump. Unlike moving around the land, jumping is implemented using just a key down event handler since we are only interested in knowing whether or not a jump command has been given and not how long it has been pressed. The jumping mechanism is implemented using parabolic kinematics equations to simulate a more realistic jump. The handler for shooting is similar to how jumping is handled, in the way that we simply want to know if the shooting key has been pressed, so that we can initiate the shooting process.

We decided to go with the traditional keys for movements since that is usually the convention in web games. Since it is a two player game, we had to take into account that two people will be sharing the same keyboard, so we decided to let player 1 use the 'm' key to jump and the 'enter' key to shoot while player 2 uses the spacebar to jump and the shift key to shoot. This groups the controls for the players and separates them so that they would be able to comfortably play together. We utilize Javascript event handlers to detect keypress events and determine the key being pressed and respond accordingly.

To improve the quality of game experience, we also included controls that allow players to pause and unpause the game by pressing ‘p’. During the animation loop, we check if ‘p’ has been pressed; if it has not, then we proceed to update objects. Further, once a game has ended, we also added handlers so that players can easily restart the game by double clicking anywhere on the screen.



Game Paused

Crispy Combat

How to play:

Try to stay alive in the game longer than your opponent!

Instructions:

Player 1, the character on the right, use the 'wasd' keys to move backwards, left, forward, and right, and the spacebar to jump. Press shift to shoot.

Player 2, the character on the left, use the arrow keys to move in the four directions and the 'm' key to jump. Press enter to shoot.

Both players must avoid the moving tables and chairs in order to stay alive.

Press P to resume!

Fig: The pause page that appears when P is hit. In the background, the paused view of the game is visible, but faded dark in order to uphold accessibility standards.

Generating Objects

For generating the tables and chairs, we outsourced glb models for the objects. Using the provided Flower.js file as an example, we generated Table.js and Chair.js. For each, we created a constructor where we call the parent constructor, initialize the state of the object, and set the starting position of the object (constrained by the boundaries of the land). We then randomly picked either the x direction or the y direction for the object to move in and chose a speed between 0.05 to 0.1; we then had addTable() and addChair() functions where we loaded the model into the GLTFLoader. In order to have the tables and chairs continuously move, in the update functions in the classes, we updated the positions of the objects by adding the direction of the object, multiplied by their speed, to their position vectors. With the tables and chairs, since they are constantly moving, we did not want one of them hitting a player as soon as the game started, before the players even comprehended what was occurring in the game or realized how the game should be played. Thus, we tested different starting positions of the tables and chairs to ensure both that a player would not get immediately hit by an object and that the objects were still spread out enough to create meaningful obstacles for the players. Lastly, in SeedScene.js, we added 3 tables and 3 chairs by first calling the Table and Chair constructors and then calling the add() function for each item.

We initially wanted to randomly generate each object at the start of every game in order to provide a degree of randomness to the game each time. The disadvantage of this is that we would have to regenerate the positions each time there is an overlap among objects (or if the objects are too close to the players, which would result in a near instant game over). For the MVP, we decided to just predetermine the values of these positions, especially given that the difficulty of the game is already high enough without this extra randomness. Further, the random positions of the objects only adds a visual difference to the game and does not have a great impact on the game experience itself.

Handling Collisions

There are a couple elements that deal with collisions: Players, tables, chickens, and chairs. The basic framework for handling collisions is the same across these three models — it loops through every other object in the scene, and if the bounding box overlaps, then it is considered a collision. This part was quite difficult as first, as we did not understand how objects would be able to see each other in the scene, but eventually we were able to do a lot of debugging/learning about Three.JS in order to understand how to do computations on the objects in relation to each other.

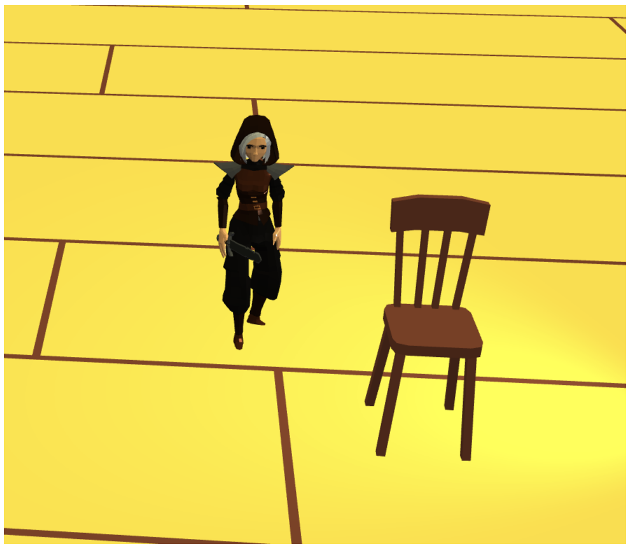


Fig: A player next to a moving chair.

From there, there is different logic for deciding what to do after it has been determined that a collision has occurred. For example, if a player hits any other object, including a chicken drumstick, the game over screen is triggered and a win screen is seen for the other player. If tables and chairs hit each other, they reflect back in the opposite direction from which they came.

Shooting the Chicken

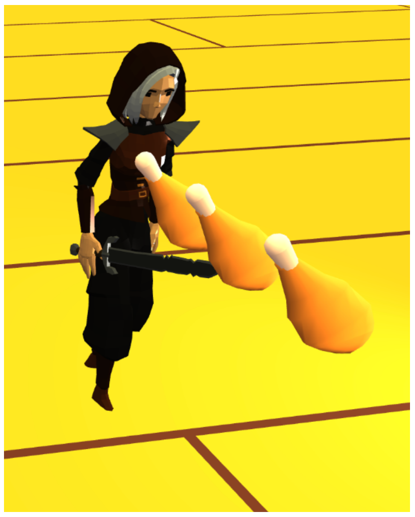


Fig: Player 2 after shooting three drumsticks into the scene.

The chicken shooting occurred when Player 1 pressed the Shift key or when Player 2 pressed the Enter key. In order to make shooting as intuitive as possible for the players, we decided that the drumsticks will be propelled from the player in the direction that they most recently moved in. We had JavaScript event handlers to indicate through their state variables that a player had shot a chicken. In the update() function of each player's class, if this.state.shoot was set to true, then we reset that value to false and created a chicken to be launched into the display. We then set the direction of the chicken to be the direction that the player most recently moved in, which was just the vector corresponding with this.state.direction because every time a player moved, we update that state variable. Lastly, we set the chicken's starting position to be the position that the player was currently in, before adding the chicken to the scene. We accomplished this by calling getWorldPosition() on the appropriate player object. After experimenting with different velocities for the chicken to move at, we settled on 0.1 as a speed that made it feasible to eliminate the opposing player with the chicken but also possible for the opponent to dodge the chicken.

We initially considered shooting the chicken in a parabolic path, but ultimately decided that it would be less intuitive for the players, as they would have to think more about where the chicken will land, whereas if the chicken was propelled in a linear direction, then it's much easier for the player to predict where the chicken will hit.

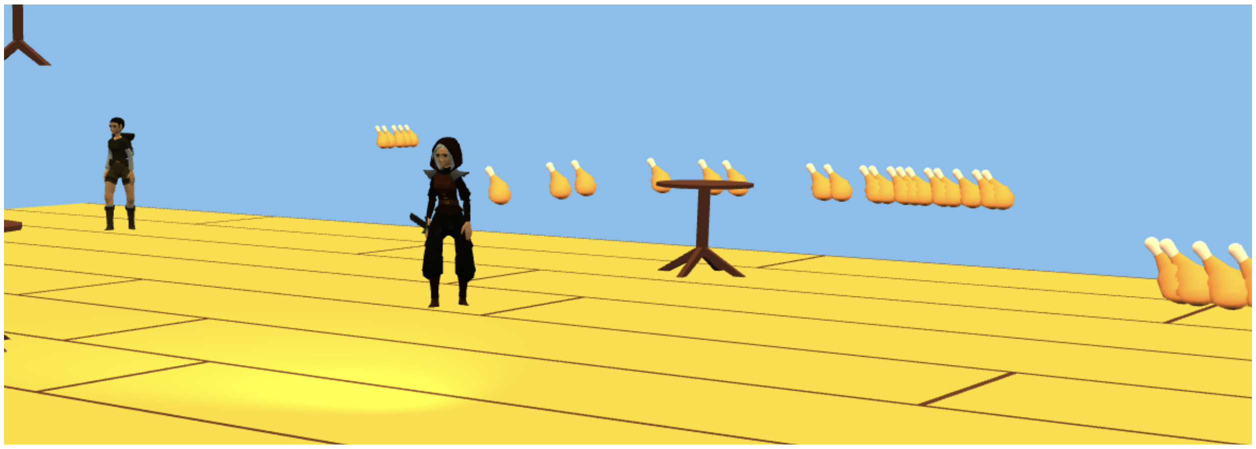


Fig: The straight-line path of the chickens shot by both players is demonstrated here.

Results

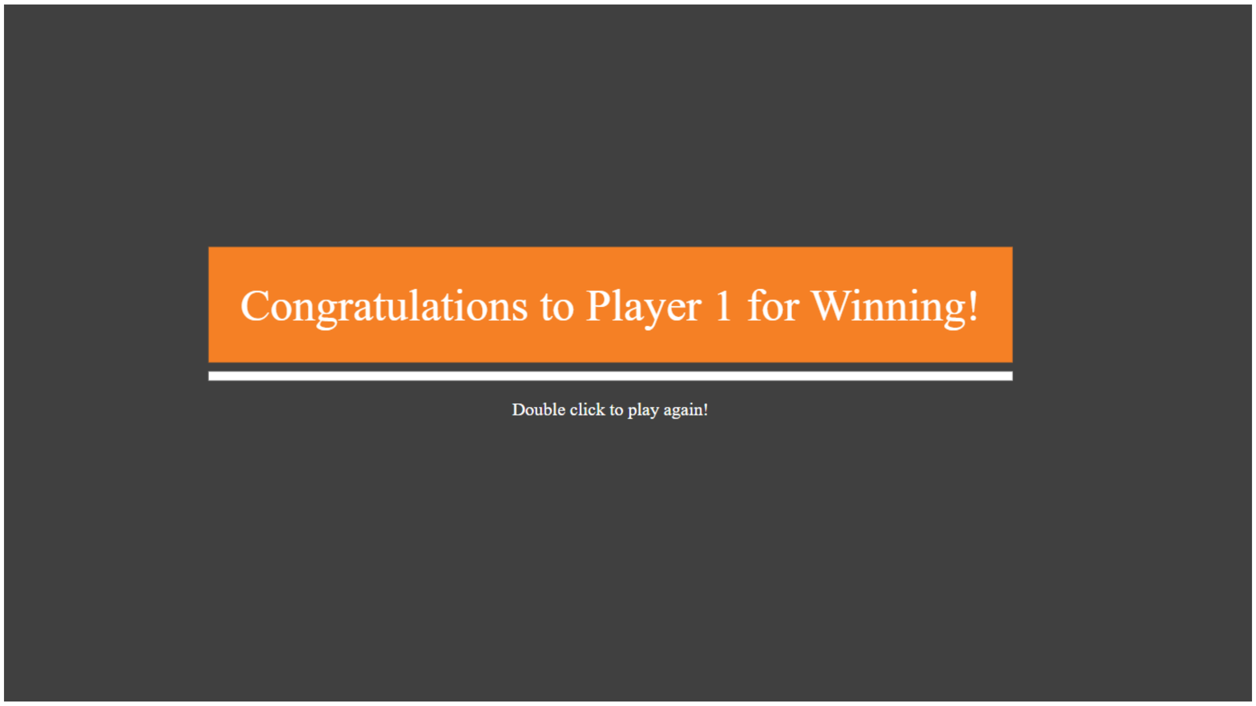
Through the methods described above, we were able to achieve our minimum viable product: a fully functioning interactive 2 player shooting game with easy-to-use user input controls and clear visuals. In addition, we were able to implement a couple of our stretch goals: this included moving the obstacles (the tables and chairs) at a constant rate and allowing users to pause the game, where they can see both the rules/instructions and the state of the game in the background.

While there were not many quantitative evaluation metrics of our project, we were able to measure our success qualitatively. We mostly made observations on whether the game was accessible to play in terms of the user controls, whether the game was an engaging and enticing experience for users, and whether there were any bugs that hindered the game's functionalities.

Testing

To ensure that our game was free of bugs, we wanted to thoroughly test our program. Since our game consisted of a random element with the movement direction of the chairs and tables, we extensively played our game to confirm that the collision handling worked correctly no matter which direction the tables and chairs moved in. During testing, one issue that we noticed was that after the game was over, objects in the game display were still moving (and were just hidden from the users). As a result, the game over page that congratulated the winning player would for example, sometimes have "Congratulations Player 1" as its message and then suddenly switch to "Congratulations Player 2". Thus, we fixed this issue by pausing every object and player on the game screen as soon as the game ended.

For the first two metrics, useability and engagement, we not only played the game many times ourselves, but we also tested the game amongst others and collected the feedback that they had on their experience. Between the users that we were able to test our project with, they appreciated that the game controls were simple to use: they liked how they didn't have to deal with both the mouse and the keyboard and how there weren't too many keys that they had to control. In addition, our test users appreciated that the game was 2 players with both shooting and obstacle-based elements because they found that they easily got addicted and competitive during the game. One critique that we received was that the instructions throughout the game could be larger and clearer. As a result, we adjusted both the description and the styling pertaining to the game rules that were displayed. Our test users also relayed that between a side view and a slanted top-down view for the game layout, they preferred the slanted top-down display because of the ability to visualize the depth of the game screen.


Congratulations to Player 1 for Winning!

Double click to play again!

Fig. Gameover page.

Discussion

While we were able to accomplish all of the goals and functionalities of our game, given more time, there are a couple of additional features that we would love to implement to enhance the user experience even more. For example, it would be exciting if players were able to jump on the moving tables and chairs and subsequently travel in the direction that the objects take them in. In addition, our current version of Crispy Combat only allows players to shoot in 4 directions, according to the direction that they last moved in. A future feature could allow players to choose their own direction to shoot drumsticks and the strength with which they use to project them.

One thing that would really improve our implementation would be creating our own models so that we would be able to better detect collisions rather than to guess the dimensions of models with over-generous bounding boxes. This would also allow us to animate the players. For instance, we would be able to better simulate turning the player based on the direction that they are moving in, which we could not do right now because of the way the models are designed. We would also be able to animate the “death” of a player by having them collapse on the floor when hit, which would make it more obvious that the game has ended.

Conclusion

Overall, we felt that we were able to learn a lot about the multi-faceted aspects of the real-world application of computer graphics throughout the project. Most importantly, we are proud that we were able to successfully build a 3D game that dynamically responds to user input and output with moving obstacles, randomized generation, and shooting mechanics, something none of us had any idea how to do coming into this class. We were able to play the final product both with each other and with friends who all enjoyed the game as well as the innovative late meal concept, fulfilling our goal of using a local multiplayer game mechanic to build an enjoyable graphics and gaming-centered social experience for all.

We were able to find a strong balance of workload distribution between members, collaborating to assign tasks and pair programming or debugging together as needed along the way. We also got experience with new technologies like importing Three.JS models, objects, and logic on our own without the scaffolding of the assignments. We also learned about using npm and packaging builds to deploy to GitHub pages. In terms of pure game design, we enjoyed the experience of figuring out the game mechanics and aesthetics to create an engaging game scene that would enthrall our players.

Overall, we are proud of the progress we were able to make over the span of our project and enjoyed the collaborative process throughout!

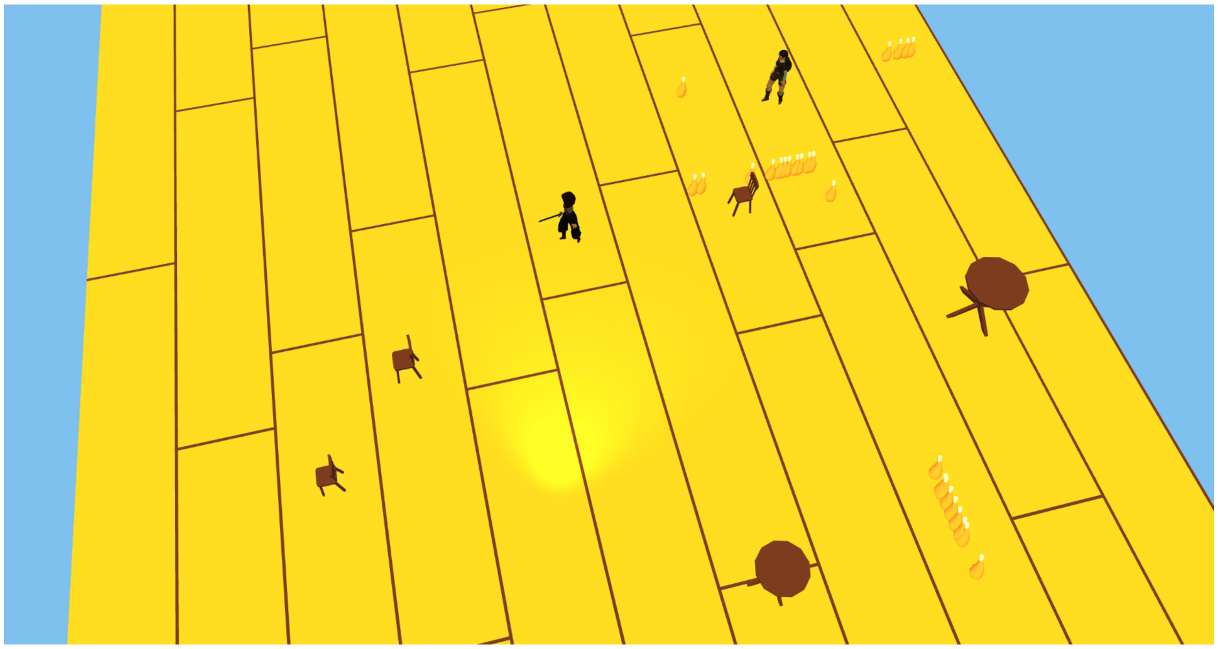


Fig: Bird's eye view of a tense, action-packed match of Crispy Combat.

Contributions

Lillian Ye

I initially worked with Donna on generating the objects used in the game, including the wooden floor, the tables, and the chairs. I tested different starting positions for the tables and chairs and wrote functions in order to continuously move the objects. In addition, I wrote logic for what happens when a player shoots a chicken: I initialized the starting position of the chicken to be the position that the player shooting it is currently in. In addition, I coded the function to have the chicken continuously travel in the direction that the player shooting the chicken most recently moved in. Lastly, I worked on some of the style elements for both the starting menu page and the game over page, as well as the controls that allow for starting and restarting the game.

Donna Wang

Initially, I worked on designing basic setups (i.e. coming up with relevant fields for each object's constructors and finding GLTF models for each object). I worked with Lillian to generate the chairs and tables on the land object. We also worked on writing the update functions to move each object along a randomly chosen axis (decided during the object's construction) while keeping the objects within bounds.

Later I focused mostly on player mechanics such as player movements around the map, jumping, and shooting. I designed a general framework in the player files and came up with some initial variables in the chicken object constructors to simulate shooting. Then, I worked on writing code for handling game state changes such as starting and pausing the game, which includes updating the screen to show relevant html files and messages.

Michelle Liu

In the beginning, I worked on importing values from land into the players and tables files so that we could figure out how to dynamically bound their movement. I also coded the collision logic for tables, chairs, players, and chicken tenders, making sure to trigger a bounce/reflection for solid objects and a death screen for players. I worked with Donna to debug the jumping mechanics of the player and the chicken shooting according to the gravity equations, and worked with Lillian to debug the movement of the chicken drumstick so that it would move across the scene at the correct speeds and directions.

I also worked on debugging the spawn point of the chicken drumsticks, as they were appearing in seemingly random places — to fix that, I worked on conversions between local and global coordinates between players and chickens in order to make sure that everything was spawning in the correct place relative to each other, which our program wasn't accounting for at all beforehand.

Works Cited

Inspiration

- https://harveyw24.github.io/Glider/

Models:

- Player 1: https://poly.pizza/m/y9KWOVG21R
- Player 2: https://poly.pizza/m/ZwF0K7WBmu
- Drumsticks: https://poly.pizza/m/7T8Ewz3VCom
- Table: https://poly.pizza/m/dfL6q_VTWVf
- Chair: https://poly.pizza/m/e9-mepLIq0X
- Land: https://poly.pizza/m/4qpvnIQNcl5