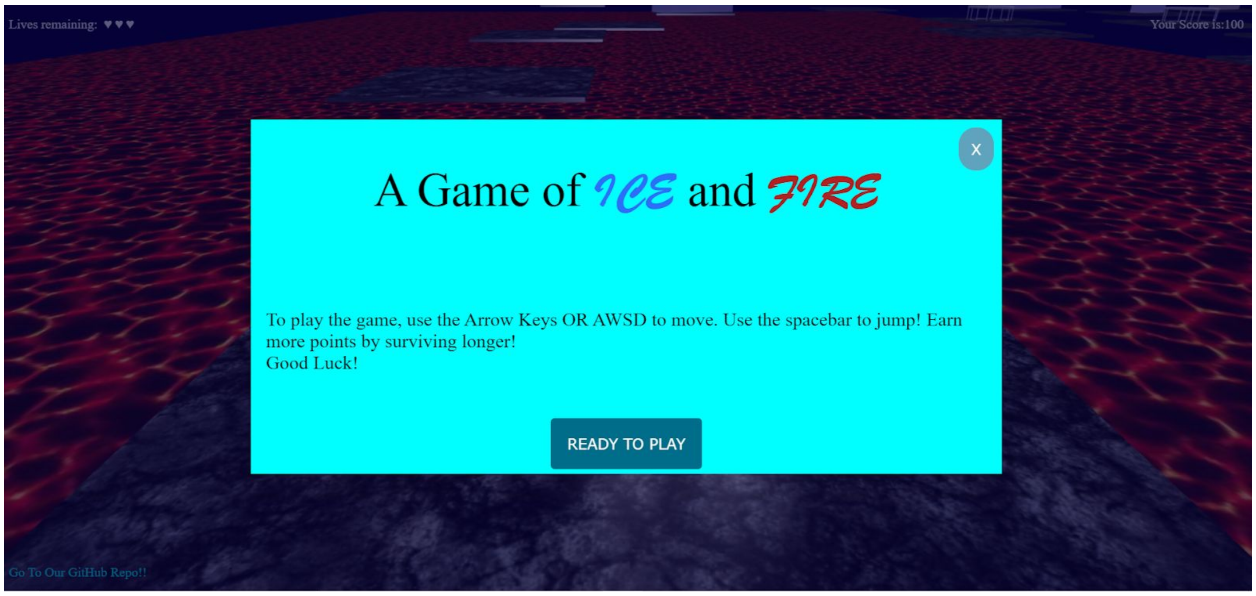
A GAME OF ICE AND FIRE

Ofek Peres, Dovid Braverman, and Gabe Roth

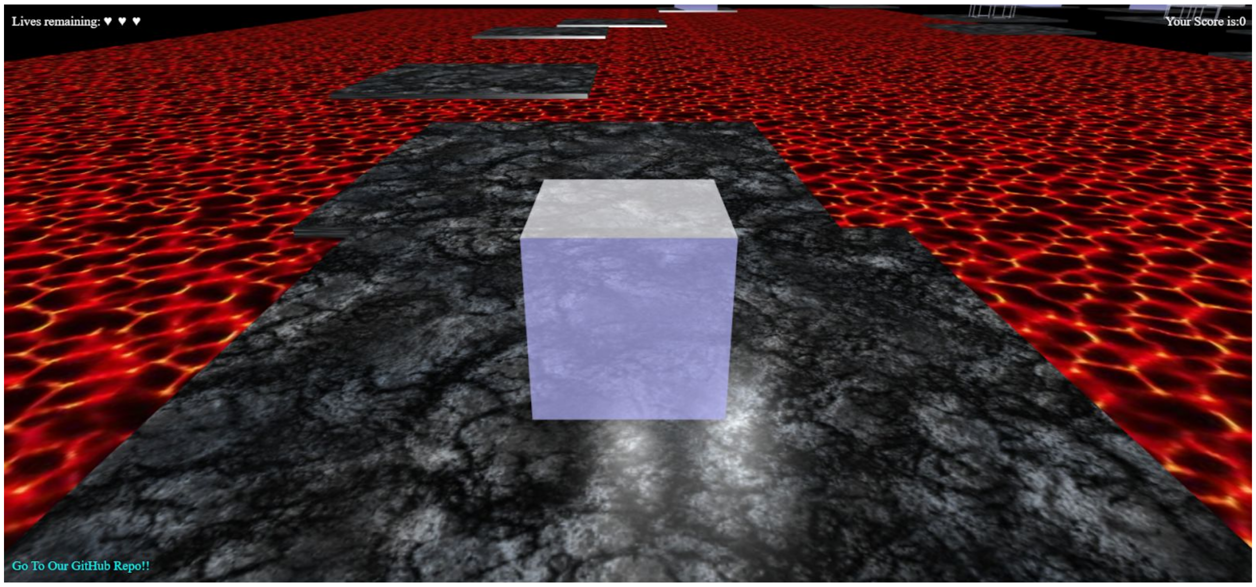
Abstract

We created an infinite runner game where the player is an ice cube traveling along volcanic ash platforms in order to evade the sea of lava beneath. The ice and fire themed game is entitled "A Game of Ice and Fire," a spin off of George R. R. Martin's epic fantasy novel series "A Song of Ice and Fire." In the game, the player has the ability to move forward, backward, left, right, and jump; the goal of the game, like any infinite runner game, is to make as much forward progress as possible. The game design includes randomly placed volcanic ash platforms, rotating iceberg obstacles, ice trampolines, and storms of giant hail. As the player advances, both the aesthetics and difficulty of the game develop. Bablylon.JS and Cannon.JS were utilized to create a game that balanced user interface simplicity, game complexity, visual aesthetics, and smooth graphics rendering.



To play the game, use the Arrow Keys OR AWSD to move. Use the spacebar to jump! Earn more points by surviving longer!
Good Luck!

1 Introduction



1.1 Goal

Our goal was to design a game that was intuitive to use, visually aesthetic, and ideally, somewhat addicting. Though it was important for our game to have a cohesive theme, we decided to prioritize the mechanics and features of the game. We believe that the skeletal design of a game--the controls, the physics, the movement, the obstacles, and the objective-- influences the player's enjoyment, more so than the game's theme. We allowed our ideas for the game features to later conjure up a theme to tie all of the features into a visually cohesive experience. This strategy enabled our game features to be driven by a balance of accessibility, difficulty, and intrigue, instead of being constrained by a predetermined theme. In fact, our theme changed multiple times throughout the game development process.

1.2 Previous Work

This game was inspired by popular computer and smartphone games from our own experiences, including Subway Surfers, Temple Run, Doodle Jump, and Cube Runner. In addition, we were impressed by the simplicity, cleverness, and addictive-nature of Collideoscope, one of the 2019 COS 426 final project winners, created by Noah Moss and Evan Wildenhain.

Like all of these games, we knew we wanted ours to have the straightforward objective of progressing forward as much as possible. This simple objective of venturing into the unknown of the depths of the screen keeps the game player engaged and curious. We toyed with the concept of discrete lanes drawn from Subway Surfers, but eventually adapted it to continuous side-to-side variability between discretely designed lanes. We considered the option of turning, which exists in Temple Run, but we decided against it due to the disorienting effect of constantly rotating the players perspective by 90 degrees. Though Doodle Jump is constrained to 2D and we wanted our game to provide a 3D visualization, we derived our idea to incorporate a trampoline from the spring objects in Doodle Jump. Objects in an infinite runner game often impede the player's success, but we were fond of the idea of "power-up" objects like a spring, which provides an extra impulse for jumping. The main concept we used from Cube Runner and Collideoscope was to start with a simple implementation--a cube for a player, and cubes for obstacles. Cube Runner and Collideoscope demonstrate that even simple games with simple primitive objects and a minimalistic theme can be incredibly engaging and addicting. This inspired us to develop our game gradually and to focus first and foremost on the gaming experience.

Our main deviation from all of these previous games is that our game did not set the player with a constant speed into the screen. We decided that it would be interesting for the game player to control the pace of advancement. This allows for variable strategies of caution when playing the game. The player may have to reconcile eagerness to increase their score with playing carefully and conservatively to avoid drowning in the molten lava. Thus, in addition to drawing inspiration from previous games, we made sure that our game was unique in its structural and aesthetic design.

1.3 Design

The framework for the project was predetermined, even before we began working on A Game of Ice and Fire. ThreeJS and BabylonJS were both considered, as we utilized ThreeJS throughout COS 426, and we developed a 3D snake game using BabylonJS to see the possible advantages and pitfalls of the framework. The 3D snake game can be found at https://ofekperes.github.io/3DSnake/. Additionally, React was considered for its state management as well as its convenient manipulation of elements on the page. Ultimately, React was discarded due to the overly cumbersome challenge of connecting the BabylonJS or ThreeJS engines to the canvas. React-Three-Fiber was also considered but discarded due to its steep learning curve and the short time period available for the final project.

The 3D snake game was an excellent learning experience as it highlighted a few key takeaways to bring to A Game of Ice and Fire. 3D motion is very challenging for the user. 3D Snake is orders of magnitude harder than 2D snake because keeping the orientation of the snake in mind as the user plays can become very convoluted. Additionally, the fluidity of the game was hampered when shadows were implemented. While the performance hit was not enormous, it was noticeable enough to be noted as something to avoid if concerns for smooth gameplay arose. Lastly, a basic menu with click and keyboard events was implemented using BabylonJS's keyboard listener and click events activated by mouse press.

From the discoveries made in the 3D Snake game, we made some decisions about our game early on. It was to have simple controls that require no learning curve and a simple objective. We also opted for simplicity and sparsity in our meshes, to improve gameplay. Once we completed basic functionality, we would (and did), add further functionality and complexity to the game.

One thing we decided on early was to recycle meshes, because it is much more efficient than constantly creating and throwing out new meshes. However, we also wanted the game to appear continually new to the player--even while reusing the same meshes. To accomplish this, we settled on a basic game plan: to implement 4 distinct Lane objects, each composed of 10 Platform objects. Upon initialization of the game, each platform is randomly assigned an obstacle (or nothing), and the frequency with which each obstacle appears can be easily tweaked. Then, each platform is *randomly* assigned a position within a given window. After each platform is behind the player, that platform is moved to the front (far into the screen) with the *same* obstacle it had on it, but with a *new* position. Depending on the obstacle which is atop the platform, it may need to be reassembled in some manner, but this is a relatively low intensity task. Thus, the game varies, because new platforms become easily accessible or inaccessible--for example, what was once an easy jump off of a trampoline now requires a double jump with careful timing to make it past one of the spinning iceberg obstacles. Thus, by updating the platforms' positions with randomly selected coordinates within a fixed range, and randomly choosing the obstacles each time the game is played, we have a very efficient way of keeping the game new and exciting.

2 Implementation

The essence of a game is the updates that happen in every call of the render loop. It was important to us to keep our code clean and avoid cluttering the render loop. In order to accomplish this, we encapsulated all game level logic in a Game object. The Game has a myriad of variables that need to constantly be checked, reset, and refreshed. These variables include whether the player has hit one of the many surfaces (e.g. a platform, certain meshes, or a trampoline), how recently the player has hit a surface, whether the player is below a certain height (e.g. fallen into lava), and whether the player has achieved a certain score threshold. Our Game object contains a player object, a Gamestate object, an array of Lane objects, and a Scene object. The Scene object comes from BabylonJS and simply maintains access to all meshes, textures, lighting, etc. in the game. The Scene object is updated every iteration of the render loop. The Gamestate was a struct which contained the number of jumps taken since it hit the last platform (or obstacle on a platform), the number of lives remaining for the player, and the state of the game (e.g. is the game over or not). The constructor for Game calls the relevant methods for initializing the Lanes, Platforms (and their obstacles), adds an instructions modal, a game over modal, a scoreboard, and a display for the number of lives remaining to the DOM. It contains the methods for handling jumps, keyboard presses, extending the platform forward as the player moves forward, checking if the player has fallen, and so on. It also defines a series of constants for gameplay--the number of platforms per lane, the dimensions of each platform, the dimensions of each lane, as well as the speed and direction of general movement.

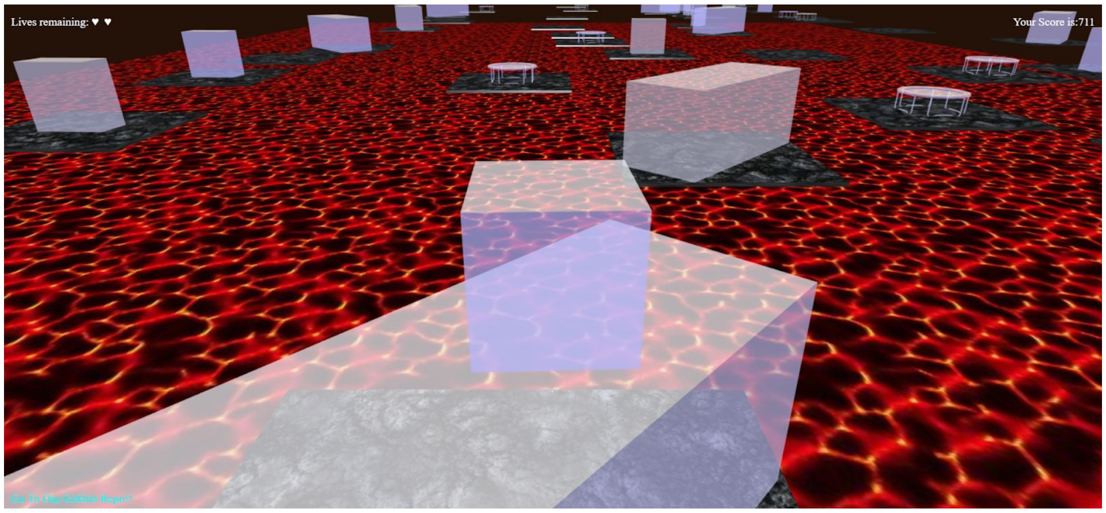
The next object in the hierarchy is the Lane object. The Lane objects serve the function of organizing the Platform objects and providing clean functionality for Lane-wide updates (e.g. reset the entire lane when the game is over). The Platform objects are where most of the logistics of the obstacles are ultimately resolved. The Platform contains an instance variable named “platform,” which represents a custom-made Box-Mesh in BabylonJS, the dimensions for the given platform, boolean variables for determining which obstacles the platform has, and reference variables for each of the obstacles that it contains. The Platform object contains methods for creating obstacles, maintaining obstacles (e.g. update the raining hail meshes), and resetting obstacles. The Platform object also has a method for checking if the player has intersected any of the obstacles on a platform or the platform itself.

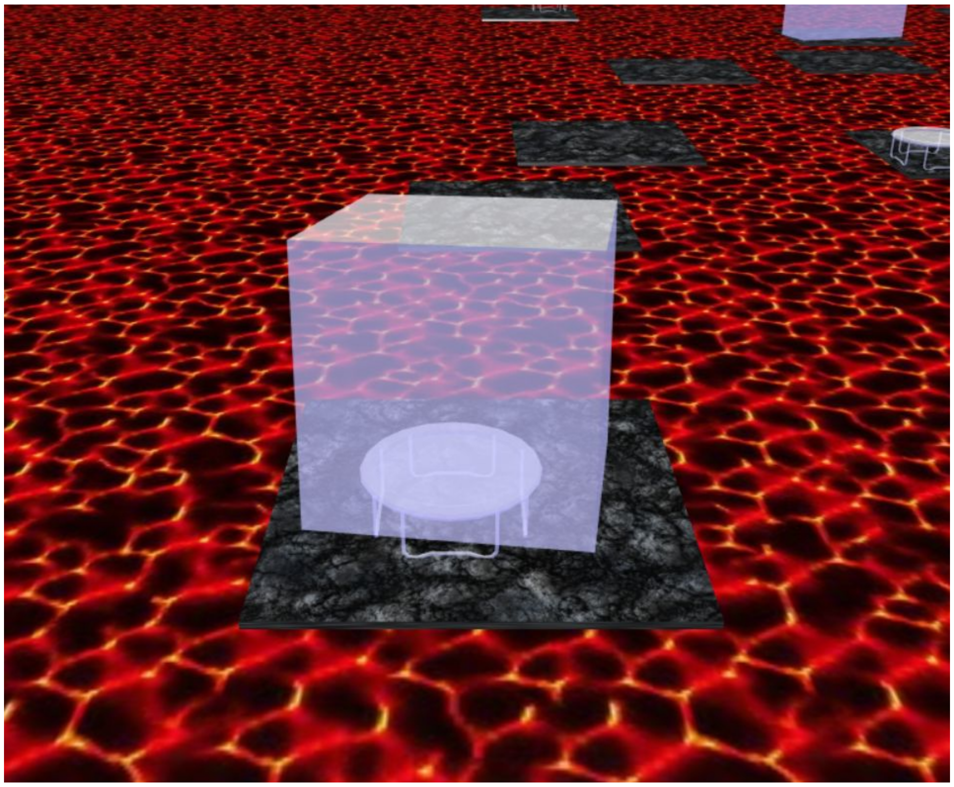
As mentioned previously, keeping our code clean and concise was one of our main goals and by wrapping all of the render updates into our Game object, we were able to simply call Game.update() to cleanly update all components of the game. Every update executed the following tasks:

- Check for player-platform intersections
- Check if the player had fallen into the lava
- Update the scoreboard and lives remaining
- Extend the lane objects to appear infinite (this involved calling the platform.reset() function to bring all of the obstacles associated with a platform to their new position)

- Update the lava floor to appear infinite
- Make the hailstorm rain continuously (by recycling the hail when they drown in the lava)
- Halt player movement if the game was over

The game has several obstacles and objects for the player to interact with. Our only unsuccessful, but also exciting, obstacle was our attempt at creating a breakable brick wall which the player could crash through. The idea was for the player to feel the impact of the wall, and be somewhat thrown back by the physics of the interaction. The wall worked quite nicely, and was an enjoyable feature of the game. However, the CannonJS physics engine in conjunction with BabylonJS seemed to be quirky. Even after much experimentation, we found that the set up was inherently unstable--when we placed two identical blocks, directly on top of each other upon a flat platform, there were slight vibrations between the two objects that would slowly grow until the bricks fell off. In the end, we decided to remove the breakable wall as an obstacle because on most playthroughs, there would be walls that were broken down before the player even interacted with them.

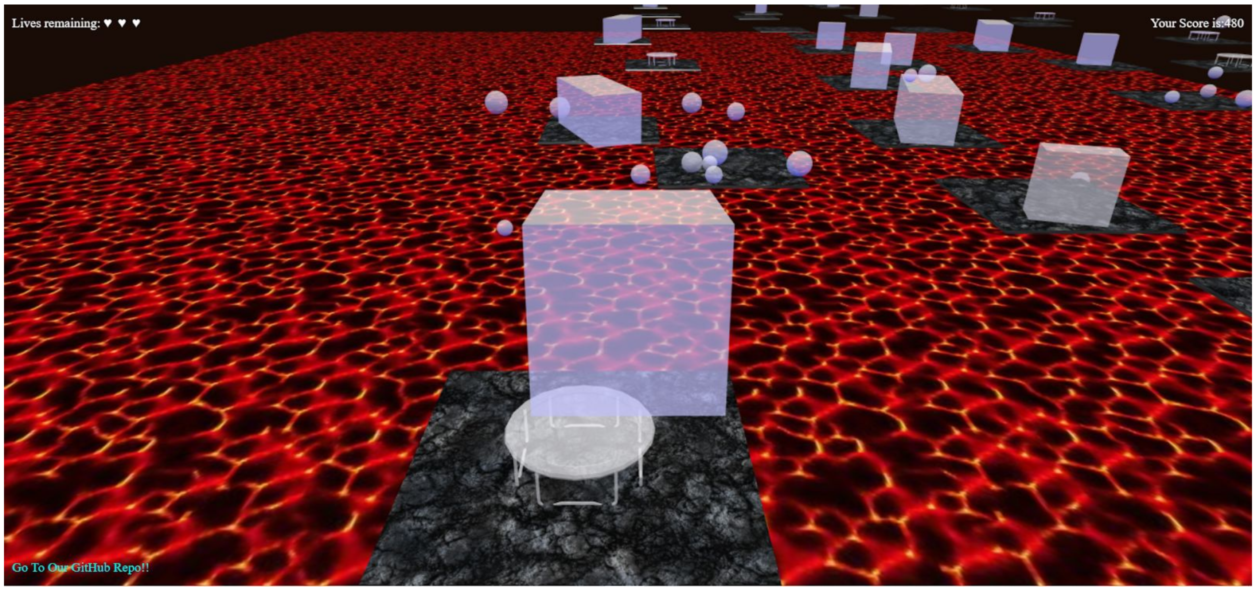




Our staple features are Launcher objects (with imported trampoline meshes from Google Poly [https://poly.google.com/view/44njxNdC0gt]) that propel the player into space. With the exception of importing the mesh into the game, the launchers were very straightforward to utilize. The launcher went through several versions: originally, we made a moving box, which moved fast enough to propel the player into space. We changed that to a flat red square on the platform, which, when touched by the player, applied an upward impulse to the player, before lastly, using the imported trampoline meshes to launch the player upwards upon contact. The mesh's .obj file was imported using BabylonJS's AssetManager. While the file is being loaded, a loading screen lets the user know why there is a delay. Then, once the AssetManager completes its task, it initializes the game and begins running the render loop. There was some difficulty with accessing the mesh upon deploying the game to the github link, even though locally, everything worked correctly. It seemed that this was a webpack issue and was ultimately solved by accessing the raw .obj file from our github repo, rather than bundling it with webpack.

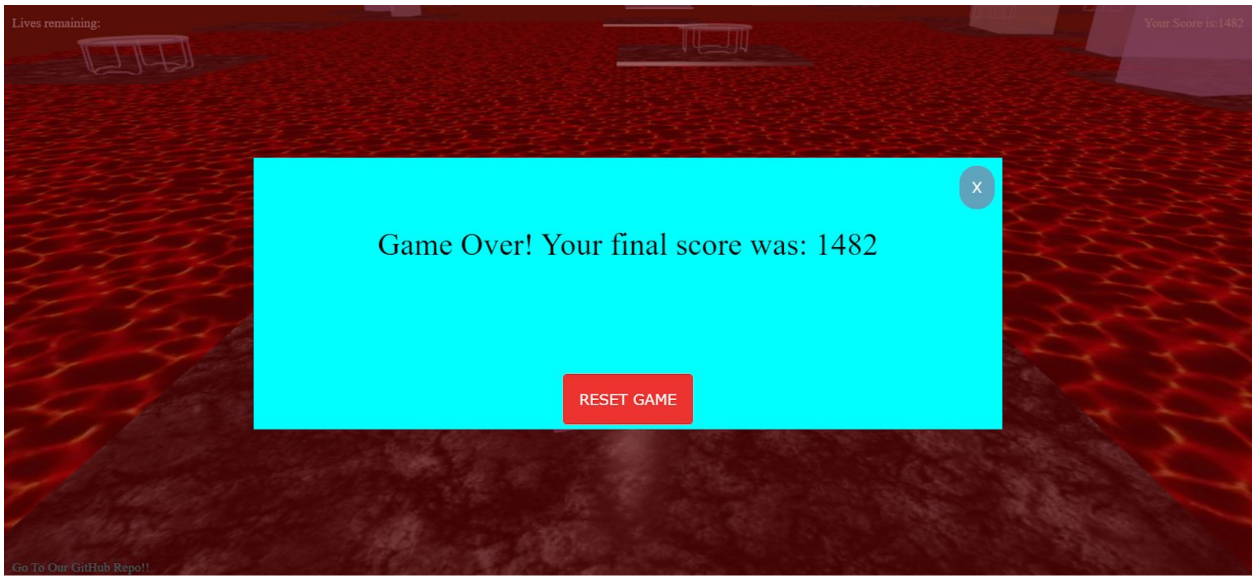
The game's obstacles included two different types of rotating boxes: smaller ones that spun around very quickly, and larger ones, with a slower angular velocity. These were BabylonJS Box Meshes with physics imposters to control their angular momentum and give them mass properties. The challenge was resetting, as well as ensuring long-term stability of these obstacles (i.e. making sure they did not slide off the platforms). There was also initially some quirky behavior when these rotating obstacles would get "reset," or moved along with the rest of their platform to the front. We found that if we set the friction to zero, it dealt with the sliding problem, and resetting the linear velocity to zero helped deal with any instabilities that arose over time.

The final obstacle we created was a hail storm that rains down on a platform. The hail storm consists of a random number of large sphere meshes that are randomly positioned above and around the platform, and then simply dropped. As they fall, they can collide with each other, the rotating box meshes, the platform, and the player, creating a more hazardous environment. This obstacle scales in difficulty as the player progresses through the game because the mass of each sphere is proportional to the player's score. This feature made the game harder as the player advanced.



Implementing a double jumping feature ended up being surprisingly difficult, and we had to settle for a limited, but reasonable functionality. We originally wanted to allow two jumps from *anywhere*--if for example, you fell off the side, you got two jumps--and these two jumps would be renewed every time you touched a relevant "surface" (e.g. a trampoline, a platform, or an obstacle). However, this implementation turned out to be unachievable given our framework. The issue was that keyboard events and the render loop seemed to happen in parallel--thus if you jumped *while on the ground* the keyboard event would be registered and the player would jump, but then the render loop would register the intersection with the ground and the number of allowed jumps would be reset to 2. In effect, this means that jumping off of the ground directly would always allow for one more jump than jumping from anywhere else. So in the end we decided to restrict jumping from mid-air to only one jump, while jumping from the ground would get two jumps.

Initially, before we settled on a theme, the player, the platforms, and the platform objects had random colors. However, once we chose our theme of ice and fire, we decided that the player should resemble an ice cube, the platforms should resemble volcanic ash, the spinning obstacles should resemble icebergs, the trampolines should be made of ice, and the abyss beneath the platforms should be a lake of lava. These visual effects would make our game more visually compelling and aesthetically pleasing. The lava lake was implemented using a tiled plane. The plane was given a material that had a repeating texture of lava. By picking the size of each tile and the overall size of the tiled plane, an illusion of an infinite lava lake was created. Keeping the lava lake stationary and only updating it on large player displacements helped maintain the infinite illusion. To implement the volcanic ash platforms that the player hopped on, each platform mesh was given a texture of a dark basalt and a dark gray emissive light property. Finally, to implement the ice like texture for the player, glaciers, and hail, a Physics Based Rendering Material was used. The material was defined to have an alpha of 0.7, an ambient and emissive color of (0,0,[0,1]) (a random blue color), and an index of refraction of 0.2. Many hours were invested on making the material look even more realistic. While we were able to find many examples of implementing refraction and reflection with BabylonJS, we were not able to implement it for the ice texture in time for this report. Thus our ice material can definitely be improved upon as a future step.



Game Over! Your final score was: 1482

Lastly, there are a few hidden easter eggs throughout the game! Upon reaching certain milestones, the game adapts! For example, upon hitting a score of 5000, the platforms' position starts changing in the vertical direction, in addition to movement within the plane. As the player progresses, the background color of the game will change. The different colors represent the most challenging zone a player has reached and is another way to measure success, in addition to the score. There are a few other fun hidden secrets that you will discover upon reaching even higher scores!

3 Results

A Game of Ice and Fire was largely a success considering we were able to accomplish our goals in the time allotted for the project. We implemented a dynamic course of platforms and multiple types of platform objects for the player to interact with. We also developed a theme that was able to combine the game features together in a visually meaningful way.

Our main source of external feedback was classmates and friends who tried playing the game. We shared the game link with a group of friends and asked them for comments as well as their ranking of the game from 1 to 5, where 1 is a bad game, and 5 is an awesome and addicting game. Although the feedback was quantitatively limited, given the short period of time between deployment of the game and writing this report, the feedback was quite positive. Six people ranked the game a 4 and three people ranked the game a 5. Feedback included comments such as "Woah so cool! Graphics are awesome :)" and "It looks really cool." Thus, based on our own experience and the limited feedback from our peers, we believe we designed a successful game. A Song of Ice and Fire seems to strike a healthy balance between difficulty, accessibility, intrigue, and visual style. Furthermore, we are very proud of the quality, structure, and readability of our code, which will allow us to easily improve the game in the future.

4 Discussion

Overall, by keeping our implementation approach modular, organized, and well commented, we built ourselves an excellent foundation for the game. Adding new features, tweaking old ones, and maintaining the game will continue to be very achievable due to our initial investment in code structure and architecture. To follow up on the current work that was accomplished, more realism can be added to the gameplay as well as additional types of obstacles. Another interesting feature would be variable difficulty levels based on user input.

Throughout the coding process, we were inhibited by our inability to use a larger number of meshes, as well as our inability to perform more calculations per time step. Though we were able to work around these limitations, they still limited the functionality of our game. Restricting ourselves to a small number of meshes for example, meant that the relative density of different obstacles was fixed for any given game. It also meant that some more elaborate obstacles were impractical. The limit on the number of calculations we could easily perform per time step limited our ability to ensure a certain quality of game play. One way of seeing this is to note that we could have controlled not just the frequency, but also the relative density of different obstacles. This could be paired with a more precise parametrization of the distances between any two given platforms. In short, the kind of meticulous parameter defining and constraining, which is done for industrial games, would ensure that the game is both challenging and possible at *every* moment.

Our limitation on the number of meshes also manifested itself in our ability to check whether or not a player was on the ground or on a surface: we checked if it intersected the material, but not which side of the material. For example, hitting the *bottom* of a platform should not grant someone another jump, but in our implementation it does. This is not a dramatic, or prevalent issue, but allows for strange behavior in those rare situations.

This project gave us an opportunity to apply many of the techniques we learned throughout the course. We learned how to import publicly available meshes, add textures and styles to them, and improved our coding, architectural, and modularity skills. More importantly we learned how to actualize an idea from conceptual design to complete implementation.

5 Conclusion

Overall, we are proud of the game we designed. We created a fun game that is visually aesthetic, challenging, and dynamic. Internally, we were pleased with our organized, well-structured, and clean implementation of the game. Future development for the game will include improving the photorealism of the game objects, solving minor corner case inconsistencies, and more closely regulating the randomness in the game to ensure a perfect level of difficulty. This project was a fun challenge that summoned the computer graphics skills we developed in COS 426, while also encouraging us to explore well beyond our prior experience within the course.

Contributions

- Ofek was responsible for the overall architecture of the project. This included the initial testing of BabylonJS with a 3D Snake game as well as confirming the simplicity of integrating BabylonJS with CannonJS. Ofek was also responsible for the initial file structure in the repo that included a basic scene with a box mesh that the user could interact with using the keyboard as well as implementing many of the game features. Ofek worked daily with Gabe and Dovid through zoom to help debug any issues that arose and continuously implement new features and improve old ones. Ofek also was responsible for the game overlays (score and health bar) and menu screens (instructions and game over).
- Dovid took responsibility for much of the game functionality. This included most of the obstacles: the different rotating blocks, changing the implementation of the launchers, as well as the unsuccessful attempt at encoding breakable wall obstacles--the behavior of all of these obstacles, as well as properly resetting them. He assisted with the jump and double-jump functionality, and helped structure the original Platform/Lane architecture, the "infinite extension" property, which moved the platforms to the front, and also wrote some of the code for randomizing the position of platforms. He also collaborated with Ofek to help properly import meshes to our game, and participated with both Gabe and Ofek in making high level decisions about the game and code.
- Gabe was the lead on the creative, thematic, and graphical aspects of the game. He came up with multiple themes which evolved as the game features themselves evolved. He was

responsible for the lava texture, the volcanic ash platform texture, and the ice cube texture. He attempted to implement refraction and reflection to achieve a more photorealistic rendering for the ice texture, but was not able to accomplish this in time. He also came up with the idea for a launcher object and a hailstorm obstacle. In terms of the implementation of the game features themselves, Gabe helped implement the infinite Lane object, the randomization of platform location and launcher location on the platforms, and the hailstorm obstacles. Gabe also spent time fine-tuning the probability variables that control aspects of the game to make sure it was both challenging and playable. Finally, Gabe came up with the idea to vary the background color and increase the mass of the hailstorm obstacles as the player advances in the game. These subtle features provide dynamic aesthetics and difficulty throughout the gaming experience.

This paper represents our own work in accordance with University regulations.

Works Cited

1. BabylonJS: https://www.babylonjs.com/
2. CannonJS: https://schteppe.github.io/cannon.js/
3. Collideoscope: https://collideoscope.github.io/
4. Cube Runner: https://vevegames.com/cube-runner
5. Doodle Jump: https://apps.apple.com/us/app/doodle-jump/id307727765
6. Google Poly: https://poly.google.com/view/44njxNdC0gt
7. Subway Surfers: https://apps.apple.com/us/app/subway-surfers/id512939461
8. Temple Run: https://apps.apple.com/us/app/temple-run/id420009108