Tune Mountain:

A Procedurally Generated Music Gamifier

A Major Qualifying Project report submitted to
the faculty of Worcester Polytechnic Institute
in partial fulfillment of the Degree of Bachelor of Science

Cem Alemdar, CS/IMGD

Leo Gonsalves, CS

Jarod Thompson, CS/IMGD

Submitted May 13, 2020

Advisors:
Professor Charlie Roberts, CS
Professor Brian Moriarty, IMGD

Abstract

Tune Mountain is a snowboarding game with procedurally-generated mountain courses. Using Spotify's Web API to obtain structural metadata of a song chosen by the player, we calculate a downhill slope conformed to the "shape" of the song, and allow the player to navigate this slope in a physics-driven simulation while listening to the associated music. Players accumulate points by successfully completing tricks and collecting snowballs, and are able to compare high scores on a leaderboard, as well as view replays of their sessions.

Contents

1. Introduction

Tune Mountain is a snowboarding game played with procedurally generated mountain terrain and environmental objects that react to a song of the player's choosing. The experience we tried to create was to visually represent the movements and melodic features present in a song, synchronized to the gameplay.

We chose Spotify's Web API to be able to make use of their song library of over 50 million songs. We initially wanted players to be able to input whatever song or sound file they wanted, but after further research we realized that the analysis of sound files would require the development of digital signal processing code that would take significant time away from building the game itself. By using Spotify, we were able not only to stream any song the player chose, but also leverage the service's detailed metadata about the structure of each song.

The game is a Web-based application accessible to anyone via a browser, preferably Google Chrome, and a Spotify Premium account. We used an iterative design process, creating separate builds of the project for successive milestones, including showcase events like IMGD's AlphaFest, end-of-term evaluations, or general playtesting sessions. We conducted two formal evaluation sessions on campus, gathering peer feedback and opinions which helped shape our final product.

Tune Mountain is now an open-source project can be played at www.tune-mountain.com. See Section 6.1 for links to the GitHub repositories related to the project.

This report describes the design, development and evaluation we undertook for the project. It explains our inspiration from previous works, how we designed the systems, which technologies we used and why we chose them, and how we learned from our mistakes and from player feedback.

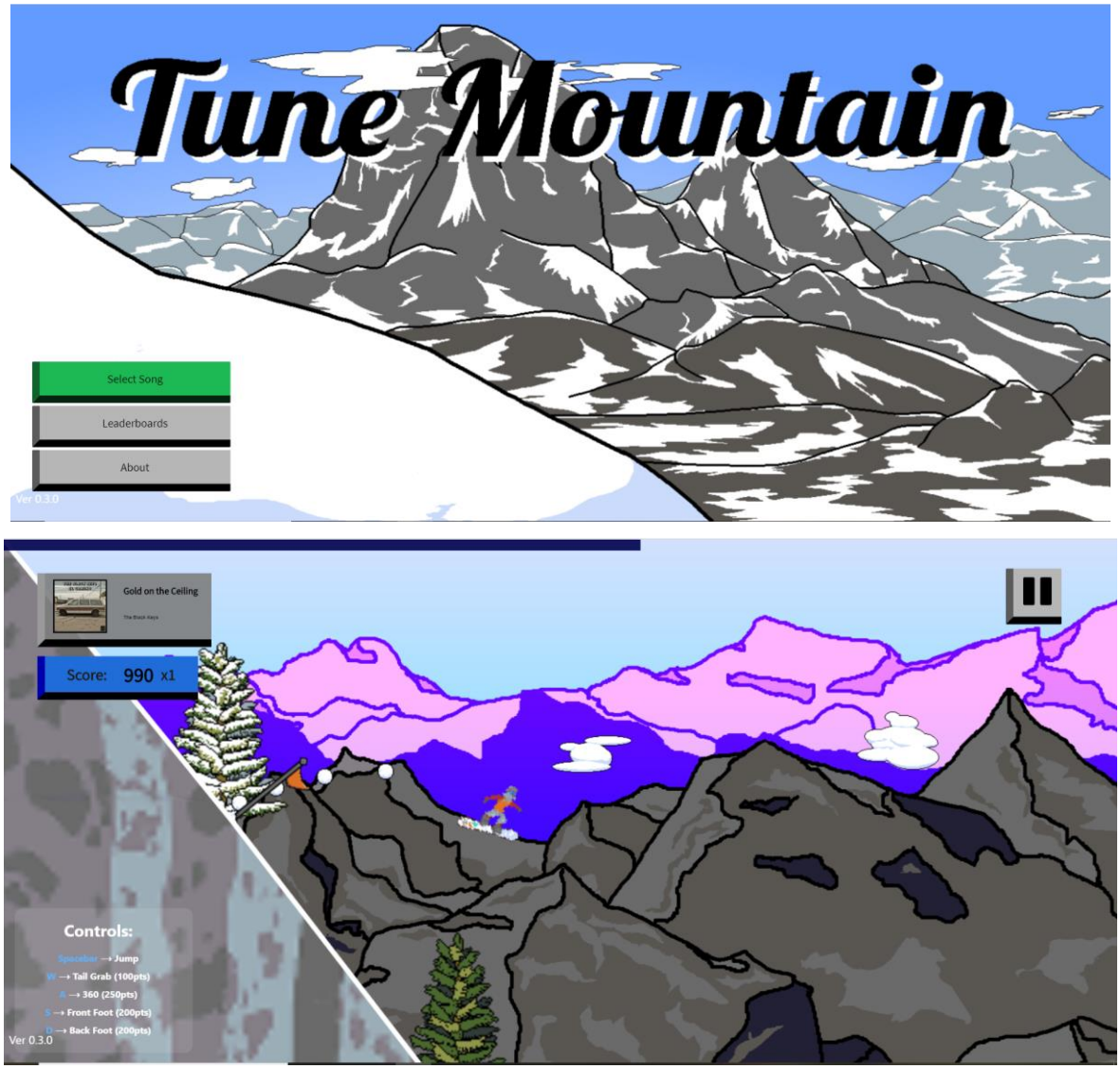


Figure 1. Home page and gameplay from *Tune Mountain.* Source: Screen capture.

We distributed project responsibilities based on each other's skills and ambitions. Cem Alemdar filled the role of gameplay designer, project manager, and graphics programmer. Leo Gonsalves worked as a full stack Web developer, designing and building the front end and the back end of the web application. Jarod Thompson acted as the engine developer and pipeline manager.

Figure 1 shows images from the final version of *Tune Mountain*.

2. Background research

While doing background research for this project, we looked at a variety of different games related to what we are trying to accomplish involving music, gameplay and procedural generation. We looked at rhythm games like *Guitar Hero* and *Beat Saber*, as well as games involving going down snow-covered mountains like *Alto's Adventure* and *SSX*, to identify their key features.

Figure 2. *Audiosurf 2* screen capture. Source: URL.

2.1. *Audiosurf 2*

Audiosurf 2 (2015) is a rhythm game created by independent developer Dylan Fitterer. It combines gameplay and music visualization, generating interactive pathways for users to navigate while they listen to songs chosen from their own music library.

The game has many available modes, but the main one is a ship moving forward at a fast speed with the ability to move between three lanes, trying to hit blob-shaped objects on the beat and avoid spikes, as seen in Figure 2 above. There are other game modes which involve a person continuously running to the right while jumping over and sliding under large blocks, and a bullet-hell mode where you control a ship to avoid energy blasts.

One thing we noticed about *Audiosurf 2* is that there is no penalty for getting hit other than the impact on your score. We really liked this feature: for an audio visualizer, being punished for hitting a spike and having to replay the song would be frustrating. We also noticed that the scoring system involves hitting collectibles in time with the song, and felt this would be a good feature to implement in our game, possibly enhanced by using collectibles to unlock special features such as character skins. Finally, we noted that the speed of the player is constant, a design restriction that would simplify synchronization of the music and visuals.

Figure 3. Screenshot of *Alto's Adventure* gameplay. Source: URL.

2.2. *Alto's Adventure*

Alto's Adventure (2015) is a side-scrolling endless runner originally released on mobile devices by Snowman, a three-person independent development team. It puts you in control of a snowboarder sliding down a gigantic mountain, tapping the screen to jump and perform tricks to earn points and upgrades. While playing, we noticed that the game's Zen mode produces an experience similar to *Audiosurf 2*, where falling doesn't impact you or restart the level.

As shown in Figure 3 above, there are interactive objects the players can slide along other than the mountain slope, a feature which adds variety. We especially liked the visual effect of parallax-scrolling mountain layers in the background.

Figure 4. Screenshot of *SSX* gameplay. Source: URL.

2.3. *SSX*

SSX (2000-12) is a series of snowboarding racing games developed by Electronic Arts. Racers are challenged to pile up as many points as possible by combining ridiculous tricks impossible to perform in the real world. We looked to *SSX* for inspiration for tricks that could be implemented in *Tune Mountain*.

The game's power bar mechanic, shown in Figure 4 above, is particularly interesting. It suggested the possibility of having tricks award more points if players were able to complete a minimum number in a row without failing.

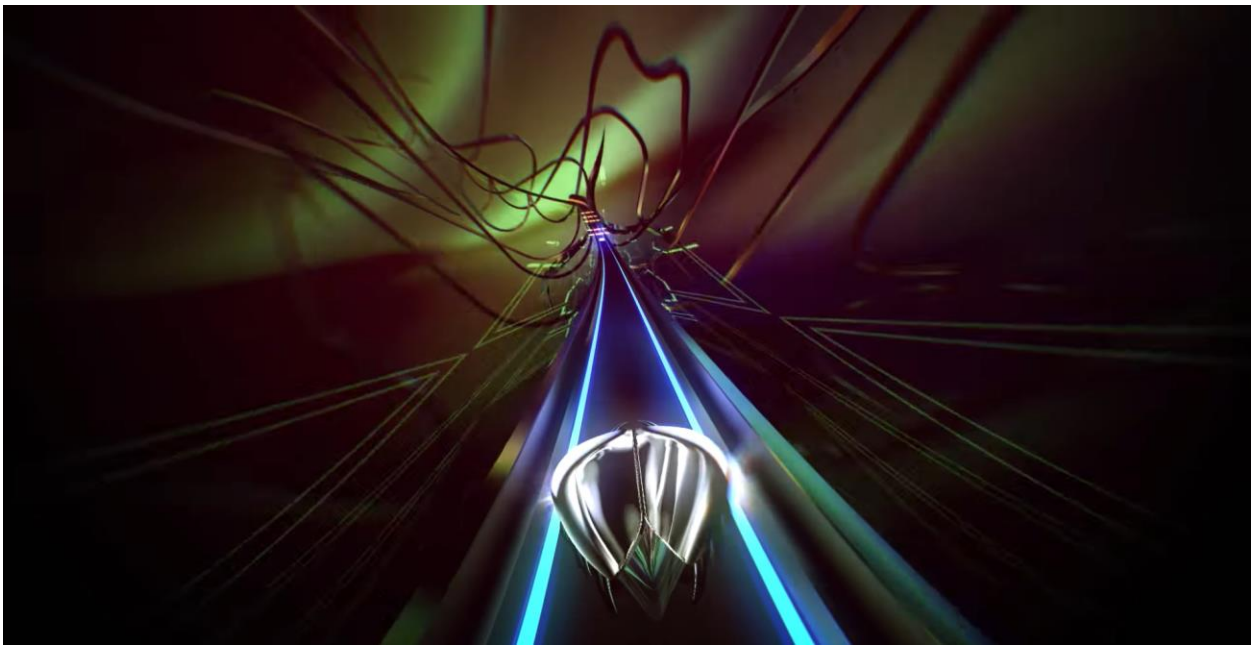


Figure 5. Screenshot of *Thumper* gameplay. Source: URL.

2.4. *Thumper*

Thumper (Drool, 2016) is a rhythm game in which the player guides a beetle-like creature forward down a track through horrific but brightly-colored worlds, with carefully timed button presses and directional inputs (“A Rhythm Violence Game”, 2013). Like *Audiosurf 2*, *Thumper* has no fail state for incorrect moves. Only your score for that section of the level is affected.

We were intrigued by the visual effects in the game, shown in Figure 5 above. They suggested the possibility of modifying our game’s color palette and/or shader overlays based on the song being played.

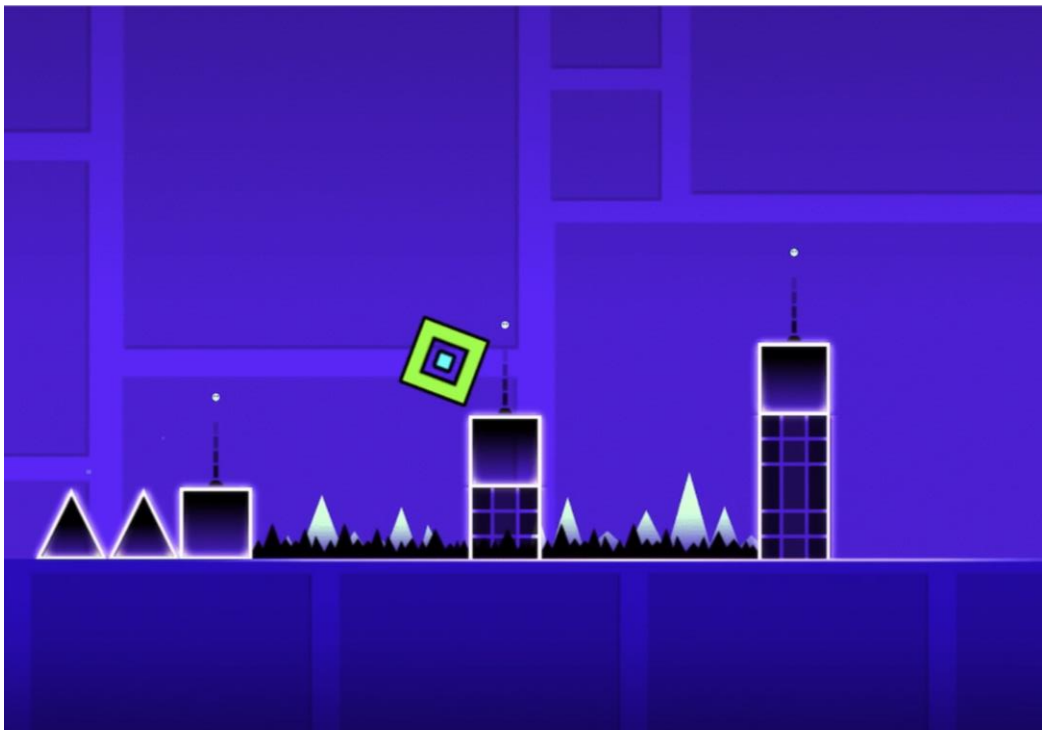


Figure 6. Screenshot of *Geometry Dash.* Source: Screen capture.

2.5. *Geometry Dash*

Geometry Dash (RobTop Games, 2013) is a rhythm platforming game, originally released on mobile devices, in which the player controls a sliding cube which must avoid obstacles with perfectly timed taps on the screen (Figure 6 above). The level designs are often misleading, making you feel as if you need to jump when doing so will send you into a spike.

This game emphasizes level mastery, because if you hit anything, you die and have to restart the current level. While appropriate for the type of experience *Geometry Dash* tries to create, we felt this would be an unwelcome feature for players wanting to listen to Spotify songs in *Tune Mountain*.

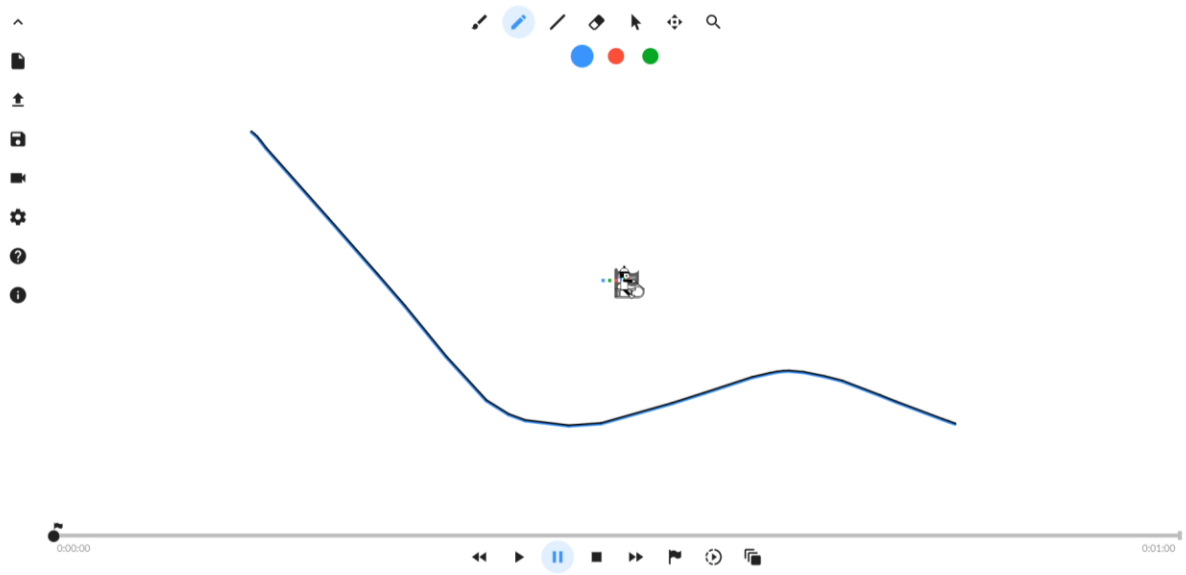


Figure 7: *Line Rider* screen capture. Source: URL.

2.6. *Line Rider*

Line Rider (2006) is a Web application created by Slovenian student Boštjan Čadež. Shown in Figure 7 above, it invites users to draw lines directly on the screen, creating slopes for a sledder to navigate. This is less of a game and more of a creative tool. It is often used to produce levels with animation closely synchronized to music, a laborious process that can takes several weeks/months to complete. Videos of these levels are widely available on Youtube (Huang, 2019).

An interesting property of *Line Rider* is the lack of user input once a level is started. The action is self-running, completely dependent on the interaction between the physics engine and the shape of the slopes drawn. While an interesting application to watch, *Line Runner* lacks the direct, gamelike interactivity we wanted for *Tune Mountain.*

Figure 8. *Guitar Hero* screen capture. Source: URL.

2.7. *Guitar Hero/Rock Band*

Guitar Hero (RedOctane, 2005) and *Rock Band* (MTV Games, 2007) are different rhythm game franchises that feature nearly identical gameplay, probably because both products originated from the same company, Harmonix. Players must hit the correct matching colored note at the right time to earn enough points to play each song in the game correctly at each difficulty level, as shown in Figure 8 above. These notes are timed perfectly to the music, and if you miss a note, the song doesn't immediately end but the game does not actually play the note that you missed, therefore the only way to hear a song the way it was intended is to play it perfectly.

These games had set song lists, as music synchronization this perfect could not be done through procedural generation. While this makes a great experience for players familiar with the songs beforehand, people who don't know (or like) a particular song will have a hard time completing it. These games also encourage mastery, since you can fail a song if you miss enough notes, and the tracks get extremely hard to play for someone with average hand-eye coordination as the difficulty level increases.

Figure 9: Screenshot of gameplay from *Beat Saber*. Source: URL.

2.8. *Beat Saber*

Beat Saber (Beat Games, 2019) is a VR rhythm game similar to *Guitar Hero* and *Rock Band*, but with some key differences. Instead of pressing buttons on an instrument-type controller, players swing virtual lightsabers in time with the music to destroy blocks, with different directional movement necessary to destroy the blocks, as seen in Figure 9 above. The game also requires occasional head movement to avoid spiked balls or laser walls.

Beat Saber shares the limited set lists of *Guitar Hero* and *Rock Band.* Players with poor reflexes or hand-eye coordination face a significant challenge on higher difficulty levels.

2.9. Past research

While searching for music- and audio-based games, we came across “Music Based Procedural Content Generation For Games” (Oliveira, 2015). This research paper was extremely helpful for understanding the possibilities of procedurally creating content for a given song. The paper focuses on creating a tile-based 2D stealth game, which players must traverse without getting caught.

Each level is generated using a .wav or .mp3 file to feed a “Music Features Extractor,” which derives information from the audio such as amplitude, pitch, tempo, and valence. This paper made it clear how complicated and time-consuming it would be to create our own song analyzer. However, it also gave us insight into which musical parameters would be appropriate to use for procedural generation.

We wanted to develop an algorithm to generate a single large curve, that would act as the ground for our generated level. We found a helpful paper about the procedural generation of Bezier curves (“Bezier Curves,” 2020). This paper talks about using small deviations to Bezier curve points to generate chandelier, lightning, candlestick, and glass models. The paper focuses on how to generate a single curve with multiple Bezier curves, segmenting it to look similar to a given example curve. This paper helped us pick a method to generate our curve structure and supplied us with a starting point for our curve generation.

2.10. Research summary

We took many elements from our research on other games related to music and procedural generation into *Tune Mountain*. *Audiosurf 2* inspired a good number of features throughout the development process, such as encouraging players to collect an object on the track in time with the beat of the song. We felt the fact that the song isn’t stopped and restarted when you crash was a very important feature to notice, since we do not want to force players to gain mastery of their favorite songs, but rather enjoy the ride down the mountain. *Alto’s Adventure* inspired our main gameplay mechanic of going down a snowy mountain slope, while *SSX* showed us how implementing tricks

and a combo meter could really enhance the gameplay of *Tune Mountain*. All of the flashing colors in *Thumper* made us wonder what the effect of applying different shaders to our sprites would affect the player experience, and *Geometry Dash* showed us that we absolutely do not want to reset players back at the top of the mountain if they fail a trick. Finally, *Guitar Hero*, *Rock Band* and *Beat Saber*, while not as cruel as *Geometry Dash*, all still encourage mastery of a particular song and can require a high level of hand-eye coordination; this helped further establish for us that we wanted *Tune Mountain* to be a casual, enjoyable experience.

3. Methodology

Over seven months of development, we went through four different development builds. We started by collecting research, designing the experience, and choosing the frameworks that would fit our goal. Then we started building the server backend, the game, and the art in parallel.

3.1. Initial concept

The first concept we had for our project was to gamify an audio visualizer. Audio visualizers have been around for decades from media players to light shows. The experience we envisioned would be composed of an initial analysis of the audio, generation of a unique level built by that analysis, and ability to play this unique level while listening to the audio in the background. This experience would use techniques such as audio analysis, procedural level generation, dynamic visuals, synchronization, and beat detection. The main goal behind this project is to create a link between the experience of listening to your favorite song, and interaction with that song. We want players to get lost in the experience we created, sculpted around the song of their choice.

3.1.1. Experience goal

We want to procedurally generate a unique mountain from any song selected from Spotify that the player snowboards down, creating a distinctive audio visual

experience for every song. Our target audience is Spotify users that enjoy music based games. The player should be able to do tricks to gain points and try to get high scores on their favorite songs by comboing tricks and not crash landing on the mountain, easily comparing themselves to their friends through leaderboards. The player movement speed, background mountains and clouds, pulsing trees, snowball coin placement, and many more are all synced to various aspects of the song, giving each player the sense that they are riding down a mountain created out of their favorite tunes.

3.1.2. Inspirations

After considering many different options like platforming, parkouring, vertical jumping, etc., we decided to create a procedural mountain that the players could snowboard/ski down. This inspiration came primarily from *Alto's Adventure* (see section 2.2). *Alto's Adventure* has a very simple and effective style of gameplay; just allow the players to focus on jumping at the appropriate moments during a level. We wanted to recreate the simplicity of just sliding down a hill, and making the music the driver of what that hill looks like. This idea fit into our scope after finding out that we didn't have to do a lot of music analysis because the Spotify Web API allows us to get song analysis for any given song, and stream it through their web player. This meant that our scope was to create a procedural generation algorithm for our mountain slopes, making the game visually pleasing and in sync with the music, and making it accessible on the web.

3.2. Chosen frameworks

3.2.1. Server and Web site

Due to the large scope of this project, we intended to modularize every part of the project so features could be implemented and function independently. The four major modules in this project are the user interface, the input manager, the state controller and the back end server. The user interface is responsible for displaying game (score, progress, etc) and initialization information (choosing a song, logging in, etc.). The input manager is responsible for abstracting the keyboard events into game actions so that all modules can listen and handle game actions independently and without the worry of dealing with specific key bindings. The state controller is responsible for communicating state changes in the game to the website, as well as allowing the website to send state change requests to the game. Finally, the back end server is responsible for handling the OAuth 2.0 authentication script, serving the website, and proxying data to be stored on the SQL database, as well as the web application itself that ties all of these modules together.

Each of these modules has been created around a central framework. The UI within the web app, as well as the application itself, are built using *ReactJS* (“React – A JavaScript Library…); this choice is influenced by the fact that React has a very useful paradigm – state-driven updating of HTML nodes (“React Component Documentation.”). This allows not only the website to have a single page nature, minimizing the HTTP requests to the back end, but also minimizes the number of times the game module itself has to be loaded. React also allows the UI components that overlay the screen when the game is running to be dynamically updated when there are state changes in the game. The reason for choosing this particular framework over the competitors, mainly Vue.js or Angular.js, is mainly developer familiarity and the sheer amount of documentation available online. React is the most widely used single-page application framework (“React Usage Statistics.”, 2020), and also one of the fastest (TechMagic, 2020). The drawbacks to this choice include a lack of clarity in what are considered “best practices” in a React application, the consequences of which will be explored further in the post-mortem section of this paper.

Before beginning development, the team discussed and planned the top-level architecture of this application. Figure 10 shows the relationship between the React application, game module, and input manager that was agreed upon as the best way to go about modularizing the development so that each programmer would be able to work independently on their responsibilities without needing to know the details of what the others were implementing.

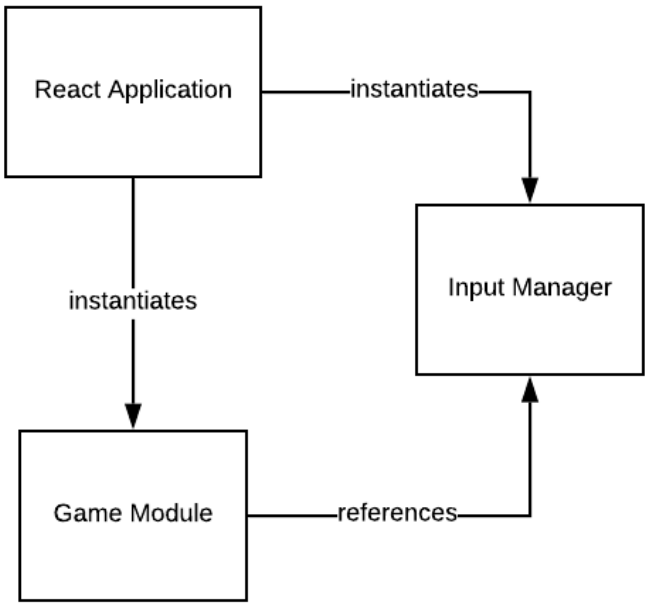


Figure 10. Relationships between main application components.

The input manager and the state controller lean heavily on the *observer-observable* pattern (Gamma et. al, 326), and the framework that helps implement this pattern the easiest is *RXJS* (ReactiveX, 2015). This comes to life in several ways: the input stream is a conversion of the vanilla JavaScript events into observables that can be subscribed to. The *state controller* implements a two-way system of observables that allows the game to communicate internal state changes that are relevant to the UI, and the UI to request state changes to the game. Figure 11 gives a detailed flow of the State Controller and of the Input Manager.

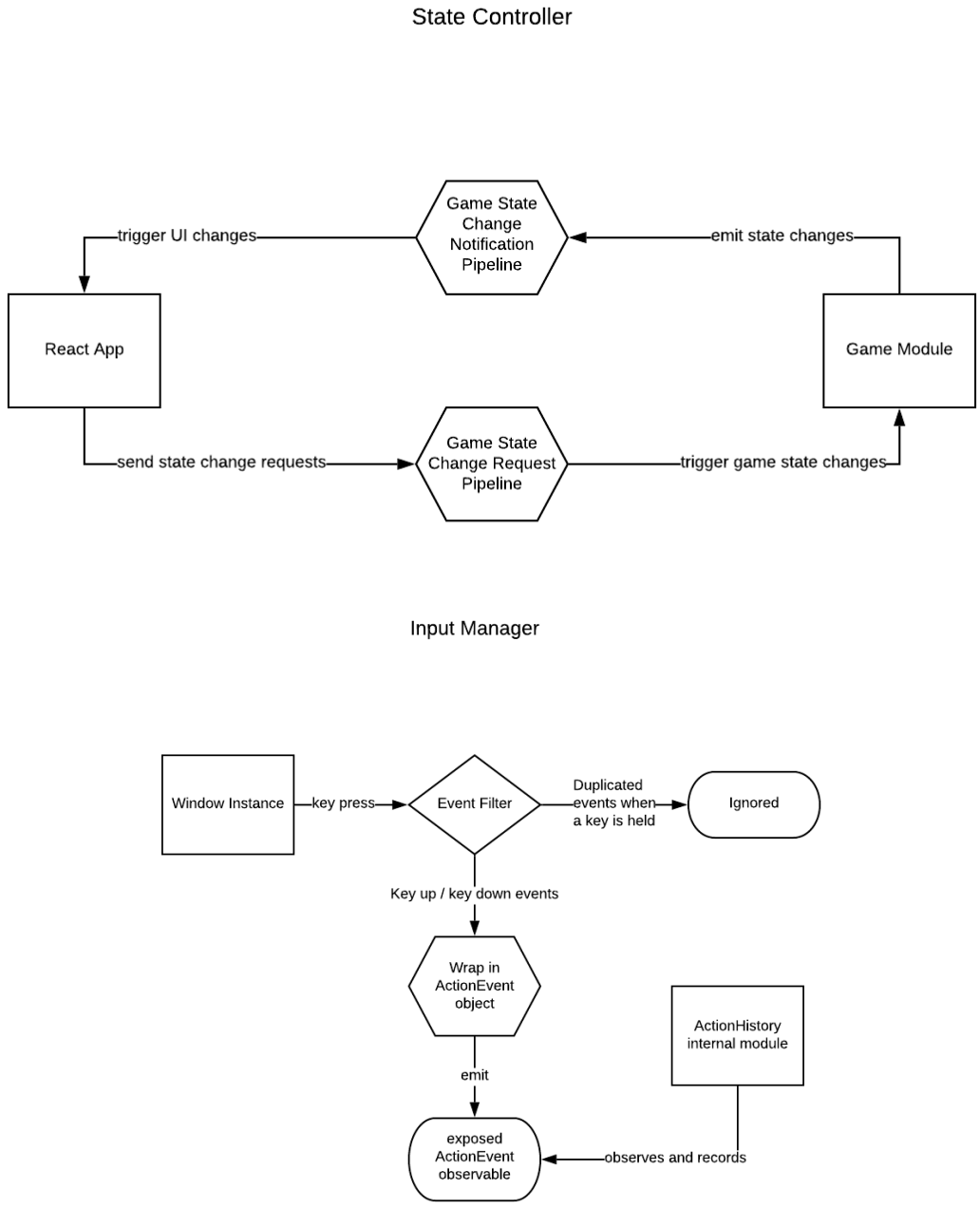


Figure 11. Workflow of state controller and input manager.

The back end server is powered by NodeJS (OpenJS Foundation), and utilizes the *express* package to create and handle HTTP requests to its endpoints. It implements OAuth 2.0, the current industry standard protocol for user authentication (Parecki, 2012), to the Spotify services, including token acquisition and automatic refreshes, as well as proxying data fetches and stores in a separate dedicated SQL

server. This database is needed to store replays, sessions, users, and responses to the feedback form we provide to users at the end of a play session.

3.2.2. Game engine frameworks

When this project idea was originally conceived in C term of 2019, the game was planned to be created in Unity. There are no required courses for IMGD Tech majors that involve using Unity, so we were interested in learning how to use a new game engine to us. However, we were a little concerned with how much work was going to need to be done by only two programmers, so we decided to find another team member if we could. Once Leo joined our team with his extensive background in web development, we decided to shift our game to be developed in browser. This allowed Leo to focus on the website/server aspect of our project and Cem and Jarod to mainly focus on game development. With this switch to web development, and the need to learn Unity gone, we discussed available options for our rendering framework with Professor Moriarty and decided to use PIXI.js ("Pixijs/Pixi.js"). Features of the framework include:

- Built for displaying things in 2D
- The ability to easily load in and create animating sprites
- Multi-platform support
- The ability to integrate WebGL filters and shaders
- An auto-detecting renderer that works correctly on all browsers based on whether WebGL is available (Groves, 2020)

PIXI.js also has an accessible documentation website (*PixiJS API Documentation*), demonstration pages that show simple examples on how to do essential PIXI tasks (*PixiJS Examples*), and a separate framework pixi-viewport that integrates seamlessly to give us better control of the camera (Figatner). The other main framework we were considering was Phaser (*Phaser*), as it has a lot of similar features to PIXI.js such as the auto-detecting renderer and easy creation of animated sprites, and is more of an overall game engine, as it also has an integrated physics engine and input manager. It would be interesting to go back and use Phaser, however at the time

we decided that the art system of the engine was too tile-based for what we wanted to do with our project.

In order to try to make our game feel authentic as you snowboard down the mountain however, we couldn't rely on just PIXI.js, we needed our game to have realistic physics. Since we did not want to try to make an entire physics engine for our project, we discussed relevant options for JavasScript physics engines with both Professor Moriarty and Professor Roberts and decided to use Planck.js ("Shakiba/Planck.js."). We knew that we needed a 2D physics engine that didn't necessarily need a visual component to it since we knew we were using PIXI.js for our rendering. We decided to use Planck.js since it was a rewritten JavaScript version of the C++ physics library Box2D. See Section 3.3.4 for more information about the specific work done with Planck.js. While in the end we got the project working, Planck.js might not have been the easiest choice simply because there is very little documentation on how it works. Unless the specific problem we were trying to solve was showcased in one of the example files, the best we could do was comb through the humongous Planck.js file. Because of this, if we could go back in time and restart the project, we maybe would have chosen to use the Matter.js framework instead, since at the very least they have good documentation right on their main website ("Matter.js"). However, in the end, the combination of Planck.js and PIXI.js worked well, as both engines came to depend on one another – all physics interactions had to be calculated prior to rendering each sprites movement, however we were only able to confirm they were running first by plugging it into PIXI.js's ticker functionality. This coexistence is the backbone of our game.

Figure 12: Initial character design and idle frame, created by Peter Griffiths.

3.2.3. Art design

The art for our game is a mix of pixel-art images and PIXI.js elements blended into a cohesive and clear visual experience to deepen a player's engagement with the sound. Our game is a pixel-art 2D style, with a consistent color palette of grays and blues highlighted by the bright colors of the player character and game objects. The restraints imposed by using pixel art made it more involved to create a clean, clear set of animations, props, and textures, but supported the development of a more consistent style. We had the pleasure of working with three artists on our MQP: Peter Griffiths, Joy Tartaglia, and Ali Saeed.

A and B term was spent working with Peter, defining the art style and workflow for the artists on the project. Peter spent a lot of time working on the character design, and the various mountain textures such as the mountain slope itself and the parallaxing background mountain layers. Figure 12 shows his initial character design, while Figure 13 shows the first set of background mountain layers compared to the final set of layers.

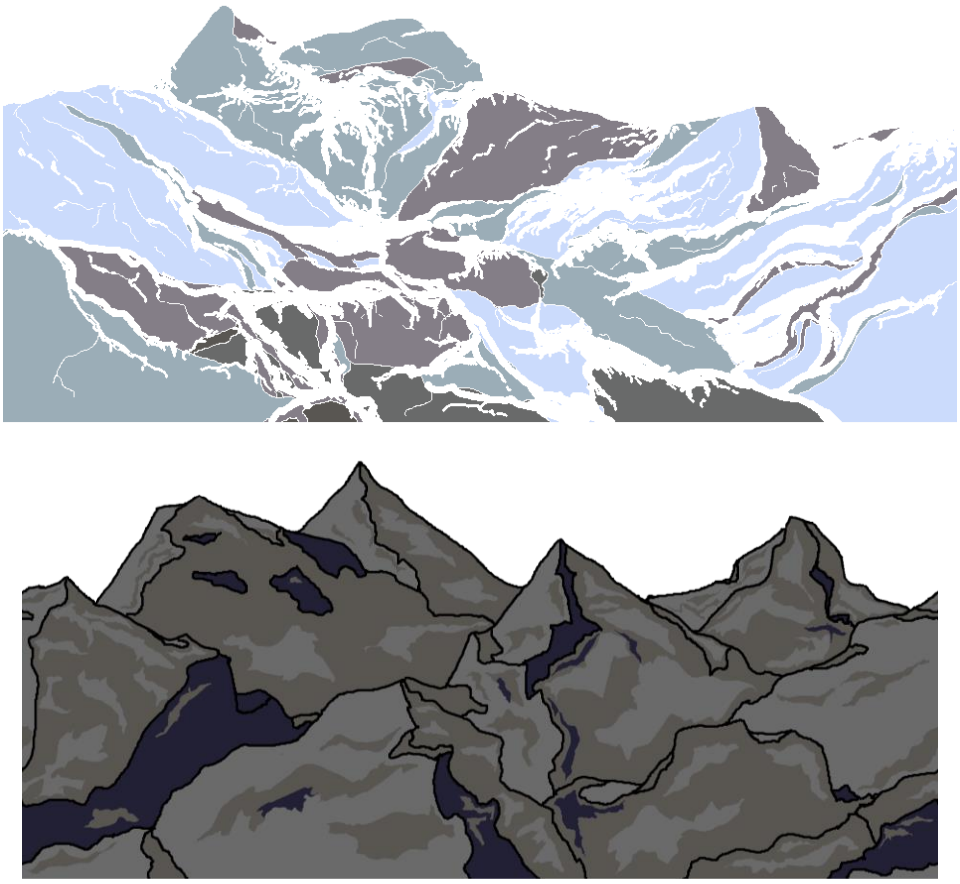


Figure 13. Comparison of mountain background layers, created by Peter Griffiths.

C and D terms were spent working with Joy and Ali. Joy worked on various environmental objects such as trees, snowballs, flags and clouds, as well as creating our title image. Ali did all of our various animations such as the jump and our four different tricks. All three artists did their work using the Aseprite application (Capello, 2001), allowing easy exporting of the PNG files we needed to create the sprites in PIXI. Figure 14 shows various art from Joy while Figure 15 shows two of Ali's trick sprite sheets.

Figure 14. Environmental objects and home page image, created by Joy Tartaglia.

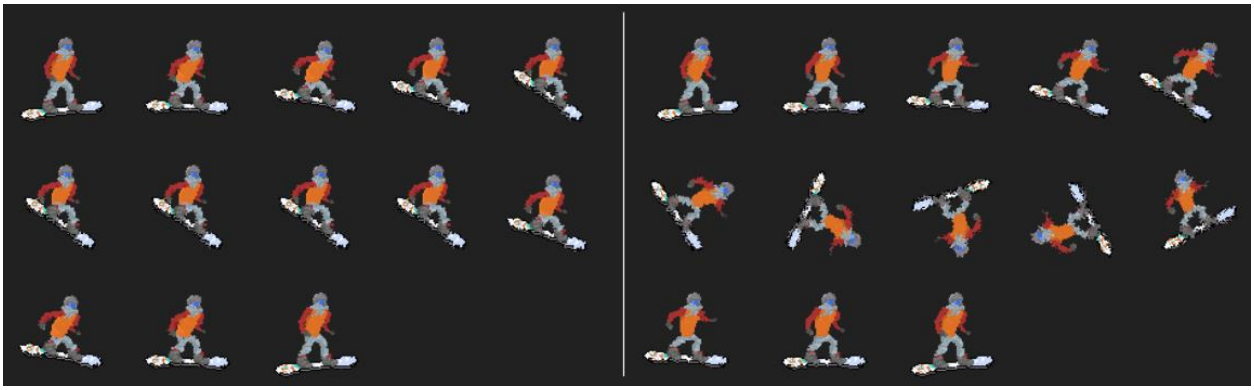


Figure 15: Sprite sheets for the tail grab and front flip tricks, created by Ali Saeed.

3.3. Game design

3.3.1. Gamifying audio visualization

We characterize the main parts of an audio visualizer as analysis, synchronization, visual feedback, and artistic translation from the audio to visuals. We wanted to add gameplay elements to the audio visualization. We decided early on that we wanted to use the motif of snowboarding down a mountain; therefore we could use the information received from Spotify to generate a unique curve for every song.

Going down the mountain on a snowboard made the decision to incorporate doing tricks quite simple. We wanted the player to be able to have a casual play experience while enjoying listening to their favorite song. So we decided to have tricks be the main interaction within the level. We score players based on the tricks they do, and there is a score multiplier for every five tricks players are able to land successfully without crashing on the slope. The score is then stored in their play session to compare in leaderboards. We also have collectible snowballs to direct players to jump. Some features that we planned to have but did not have time to include: other objects for players to snowboard on such as cable car lines and ramps as well as the mountain slope, and the ability to use snowballs collected on the slope to buy new snowboards and attire for their character.

3.3.2. Sound design

We initially planned to only have the only sound in the game be the Spotify music playing; but we realized that having audio feedback when a player lands a trick or collects snowballs could help add to the experience of interacting with the game. It lets the player realize that they have successfully done something, and the audio adds on top of the score as a reward for their action. We wanted to create a soundscape with the song that the player chooses. We ultimately decided to have a sound when the players collect snowballs for points, but not to include sounds for successful tricks completed. We felt that this would be too distracting to the player.

We chose a very short and low pitch snowball impact sound (Matthey, 2012) for collecting snowballs in order to make it feel like they are actually hitting a snowball, and this low pitch impact also created a layered beat on top of the music because the distance between the snowballs were always uniform.

Figure 16. Screenshot of *Tune Mountain*, early November 2019.

3.3.3. Terrain generation

Terrain generation is the main way we generate a unique level for the player using the song they chose. We use bezier curves to generate the mountain slope (“Bezier Curve”). Initially we tried to create an individual curve for every beat in the song. However, as shown in Figure 16 during early B term development, as a group we felt that those curves ended up too short and erratic, and not leaving enough time to slide on a particular curve. We decided that sections would be a better piece of data for us to use to create individual curves because they had a duration between three seconds and fifty seconds. It also had features such as energy, valence, loudness, key, time signature and more, as outlined in the Spotify API documentation, as shown in Appendix A, that we could use to create variety between each curve.

We created each curve using a cubic Bezier curve function, which has a start point, an end point, and two control points that dictates the direction and steepness to the curve. We started with an initial starting point at (0,0) and an end point where the displacement between the two points depends on the duration of the section, which affects the x-displacement, and loudness of the section, which affects the y-displacement. Afterwards a rectangle is generated using the start and end points to decide where to place the control points for the curve. The top left corner of the rectangle is the starting point of the curve, but it is shifted to the left by the valence variable, and shifted up by the energy variable. The bottom right corner of the rectangle is the end point, but shifted to the right by the valence variable, and shifted down by the

energy variable. Then the rectangle created is divided by the the time signature on the x-axis, and the first control point is placed using the bottom left corner of the rectangle and shifting it right by the first time signature, and the second control point it placed using the top right corner and shifting it left by the half of the entire time signature (see Figure 17).

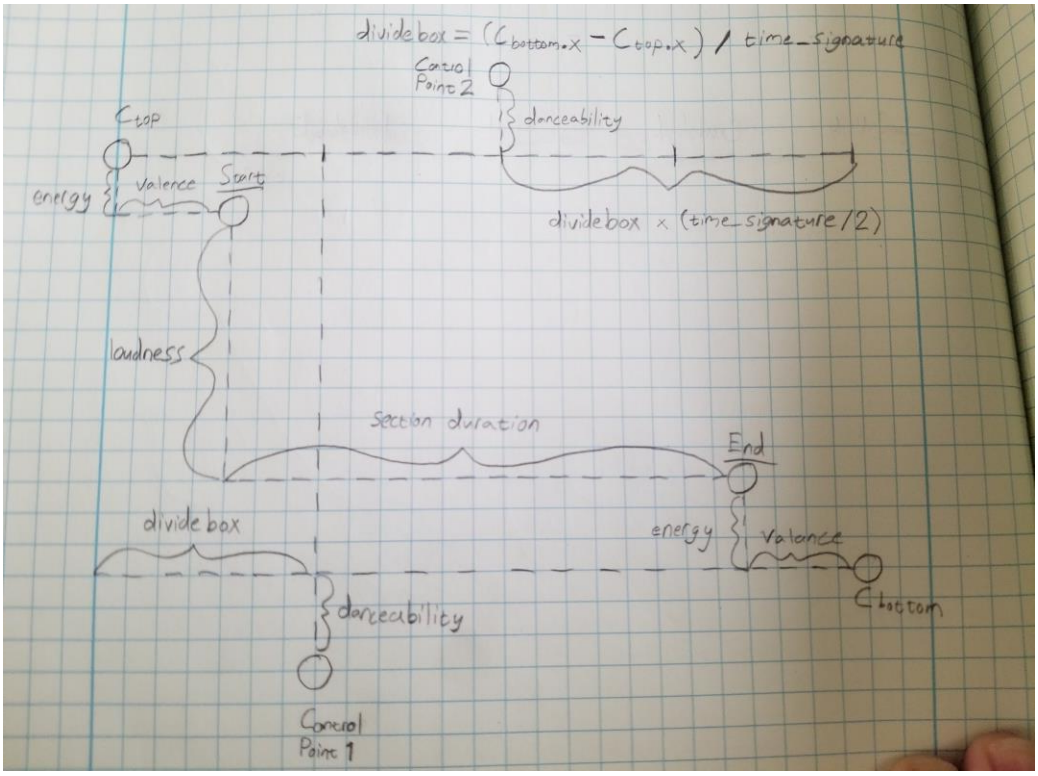


Figure 17. Generation algorithm diagram.

3.3.4. Physics

We thought that using Planck.js would allow us to create the same curves we created using our renderer because it is just a mathematical subdivision surface. We create each curve with a sixty points per curve resolution because Planck.js allowed using straight lines to create a floor surface, so we place a very small line in between two Bezier curve points, adjust the length and adjust the angle according to those two points so that there are no gaps between the lines (see Figure 18).

During initialization, we create the entire physical mountain from the points generated by our terrain generation algorithm. We implemented our character's physical object as a thin rectangular box initially, and adjusted the sprite rotation with the rotation of the physical object. Unfortunately that led to many issues with jumping and excessive rotation, so we changed the physical object to a circle, and calculated the rotation depending on our slope normal. This allowed smoother jumps, a better sliding feel, and got rid of a lot of bugs.

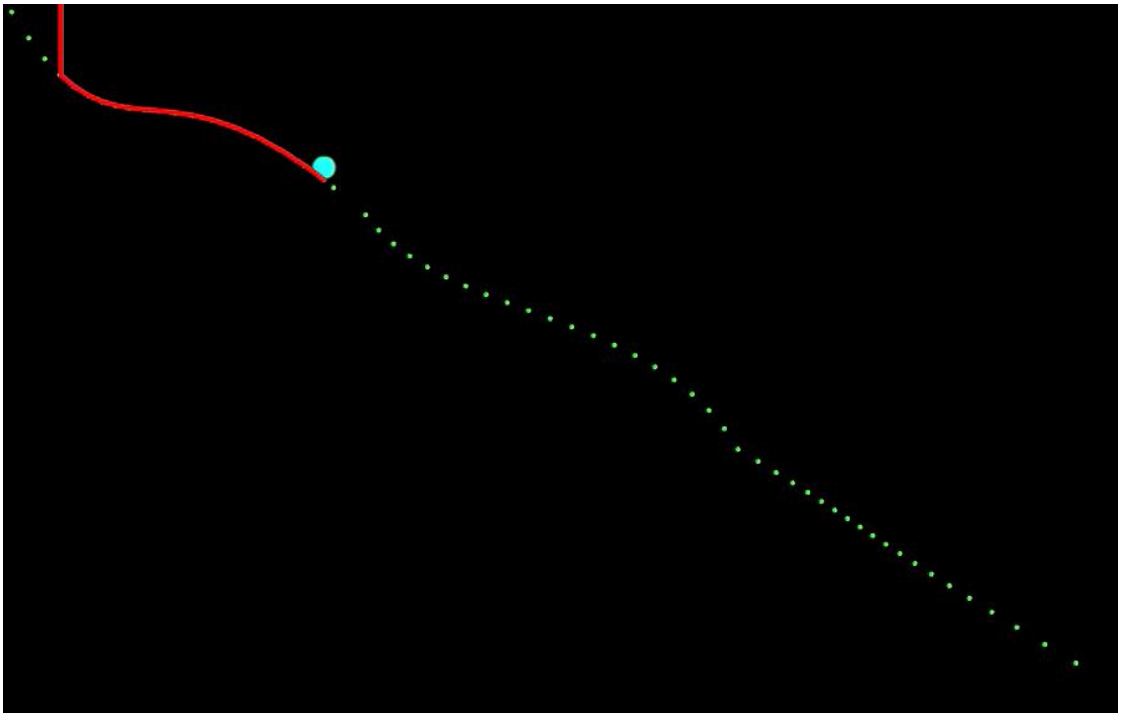


Figure 18. Screen capture of terrain generation in Planck.js.

A main issue we had with the physics was the speed of the player, while sliding down steep slopes. This was because the physical world we created was very large, and Planck.js has a maximum linear velocity threshold that the player could move in, which is twice the frame rate. We tried to create a smaller world, but shortening the curves made them more jumpy, and visually not like a mountain.

So we asked the community of people using Planck.js in their Discord server, and they informed us that the speed is relative to the world size vs the player size. So we decided to make the player smaller, and zoom in closer to the player using our pixi-viewport. This really helped us solve this major problem of giving the player a significant sense of speed, while not capping the velocity limit of the engine.

3.3.5. PIXI.js

This section of the report is written in first-person by Jarod Thompson for clarity. Before taking CS 4241: Webware: Computational Technology for Network Information Systems in A term 2019 along with the beginning of this MQP, my knowledge of web development was extremely limited. My only prior exposure to HTML and JavaScript was in CS 4731: Computer Graphics in C term 2019, which really only helped me learn the syntax as that class is all about learning WebGL. Therefore, A term became all about learning the basics of web development and how to integrate a basic PIXI.js canvas into our application. Originally, we learned how to use PIXI.js as a script tag with an init() function loaded onto the HTML page of our website, and writing individual scripts for each type of new object we were going to put on the screen. Leo helped us transition to a proper web development setup, using imports across JavaScript files so that we only have a single script tag for Game.js in our HTML page which calls all the relevant functions and files, and proper use of browserify and npm.

The first features I worked on were parallax scrolling and the integration of both the pixi-viewport framework and Leo's InputManager. Originally I tried following a PIXI.js tutorial ("Building a Parallax Scroller...", 2016) to develop the parallax scrolling, however the tutorial was for version 4 rather than version 5 of PIXI.js that we were using, so the functionality and syntax changed drastically between the two versions. Eventually I found the official PIXI.js examples page which gave a simpler explanation of TilingSprites, which made that process much easier. Integrating the pixi-viewport framework was also quite simple, as it has a very well written documentation page, and the viewport acts almost identically to the stage of a PIXI.Application. All the things we

want to be affected by any zoom or camera control, we put on the viewport and background objects like clouds and the mountain background layers belong on the static stage. Leo's InputManager was a simple import, setting up the game to subscribe to an observable and listen for any keyboard inputs of the right letter.

Next came the curve generation using the PIXI.Graphics class' bezierCurveTo() function to create smooth curves that match up with the static physical objects. This took a lot of work with Cem on the physics engine, but he was able to send me the exact points I needed for the bezierCurveTo() function, the x and y coordinates of the two control points of each curve, and the final position of each curve. Luckily, we didn't need to keep track of the start of each curve as well, since each PIXI.Graphics object has a position value if we needed it, and continues drawing from the last place it stopped drawing unless you specify you want it to move to a different location instead.

After some work with Leo to properly integrate our game into the website, setting up the application's Idle state when the website loads, and the ability to move from the Idle state to a GenerateMountain state at the appropriate time when he sends us a request from the website, I worked on importing art from Peter and figuring out how to do animations. I filled the mountain slope with a repeating texture rather than a simple white fill color, and started learning how to use the PIXI.Loader to take in the JSON files associated with the sprite sheets for our character. After making progress however, there were some errors with the JSON or image files created, and for the sake of time before Alphafest, I shifted gears and learned how to make PIXI.AnimatedSprites by creating an individual texture for every PNG file of an animation, and then created the animated sprite out of those textures. This was the final major feature I implemented before Alphafest. In the final version of our project, we would prefer to use the PIXI.Loader as it will in theory take less computational power to load one image file and one JSON file for each animation rather than the minimum of ten image files for every single animation in our application. In the end however, we did not notice any significant performance reasons for us to switch to importing animations using the PIXI.Loader.

At the start of C term 2020, Cem reworked the physics to make the game world smaller to try to combat our speed problem we were having. This necessitated a change in the way we were drawing the bezier curves for the mountain slope. Originally, Cem

would create the physical slope by drawing a bunch of very small individual lines, and then would send me the points for the overall bezier curves we were drawing for the PIXI rendering. We decided to get rid of this translation, and simply use the PIXI.Graphics lineTo function to draw really tiny lines between every individual point, and this worked to great effect.

My big project for the term however was to figure out how to properly swap sprites. We needed the character to be in his idle state by default, move to the appropriate animation when a jump or trick input is put in, and swap back once the player lands back onto the mountain again. This was a tricky problem as you cannot simply tell the PIXI.Application to remove something based on a name you give it; you actually have to figure out a way to keep track of the sprites you currently have on the screen. I decided to do this by using a sprites object that is kept in the main Game class that we created. By actually keeping the sprite objects themselves in this one singular object, it allowed me easy access whenever I wanted to add or remove a sprite from the Viewport. The character stays in the idle state when on the slope, switches to the jump animation when the jump button is pressed, switches to the appropriate animation when a trick button is pressed in the air, and switches back to the idle state when they land back on the slope. If you are in the middle of a trick and land back on the slope, there is a tumble animation before the character gets back up and keeps on going along with the song. Players are not able to jump and try more tricks while this tumble animation is playing, thus creating a reasonable penalty for when a player misjudges how much time they have to complete a trick before landing back down on the mountain. This tumble mechanic was implemented in D term 2020. This sprite swapping problem was a little more complicated that originally expected, since if I removed the player sprite that the pixi-viewport camera was following from the stage and then tried to have it follow the new one replacing it, this resulted in a very obvious flicker of the camera. To combat this, I had the camera follow an invisible sprite that is always at the same position of the character sprite. That way, I can easily remove and add sprites without affecting the camera. Other smaller tasks I completed this term involved figuring out how to offset the camera to give a better viewing experience for the player, implementing the play and pause states that Leo sends us when the website or music is paused by using the

PIXI.ticker stop and start functions, and implementing the actual trick functionality; giving the player an increase in score and switching back to the idle animation if the player goes through all the frames of an animation before landing back on the slope.

D term 2020 was all about polish and adding features. I fixed the bugs we found in our playtesting at the end of C term (see section 4.2) and prioritized the features we had thought of that made the most sense to implement in the final game. We decided that a key feature we wanted to include was a particle effect to show snow being kicked up from the board as the player goes down the mountain, as shown in Figure 19.

Figure 19. Screen capture of particle effect implemented with pixi-particles.

To do this, I found the pixi-particles framework (“Pixijs/Pixi-Particles.”), which I was able to easily integrate into the project like the pixi-viewport thanks to their easy to use documentation website (*Pixi Particles*) and their editor website (*PixiParticlesEditor*). Thanks to the editor site, I was able to adjust the values of the emitter quickly and see the results immediately without having to reload my development environment. Once it looked the appropriate way, I was able to implement it by placing the particle emitter at the back base of the character’s board and having the emitter move its position along

with the board, as well as pausing the emission when the player goes for a jump, and playing it once they land back on the slope.

The other features I implemented were the proper implementation of our four tricks along with their animations, the tumble mechanic mentioned previously, a score multiplier that increases for every five successfully executed tricks, and the implementation of random tents and animated flags, created by Joy, placed on the slope.

3.4. Website/backend development

3.4.1. Web application architecture

3.4.1a. User interface management

The UI Manager module is responsible for mounting, unmounting, and manipulating all user interface components on screen. The UI components are defined as *all components that are required for initializing the game, communicating the game state, and affecting the game state throughout the session.* It is important to note that during the development period, the UI manager also handles the feedback form, even though it does not fit the definition of UI elements.

In order to provide a smooth experience to the user and display fluid mounting/unmounting of components, this module implements the activity flow depicted in the diagram above. An attempt has been made to include CSS transitions into the React component lifecycle (“React Component Documentation”), albeit with insufficient robustness. This is achieved by piping all events that lead to an unmount through a specific pipeline, which is shown in Figure 20 below:

1. Save the intended action on the component state, i.e. encode “return to previous screen” or “move to screen Y” into a React state object.

2. On conclusion of the state update, trigger a transition in the transition component that wraps the elements of the given component (implemented as “GenericTransitionWrapper” in the codebase)

3. Hook an event listener for the “TransitionEnd” event that will be triggered by the previous action; this listener should trigger the correct action depending on the state stored in step 1, as if the intended action was “paused” while the transition component executed the CSS transition, only to be completed when said action ends.

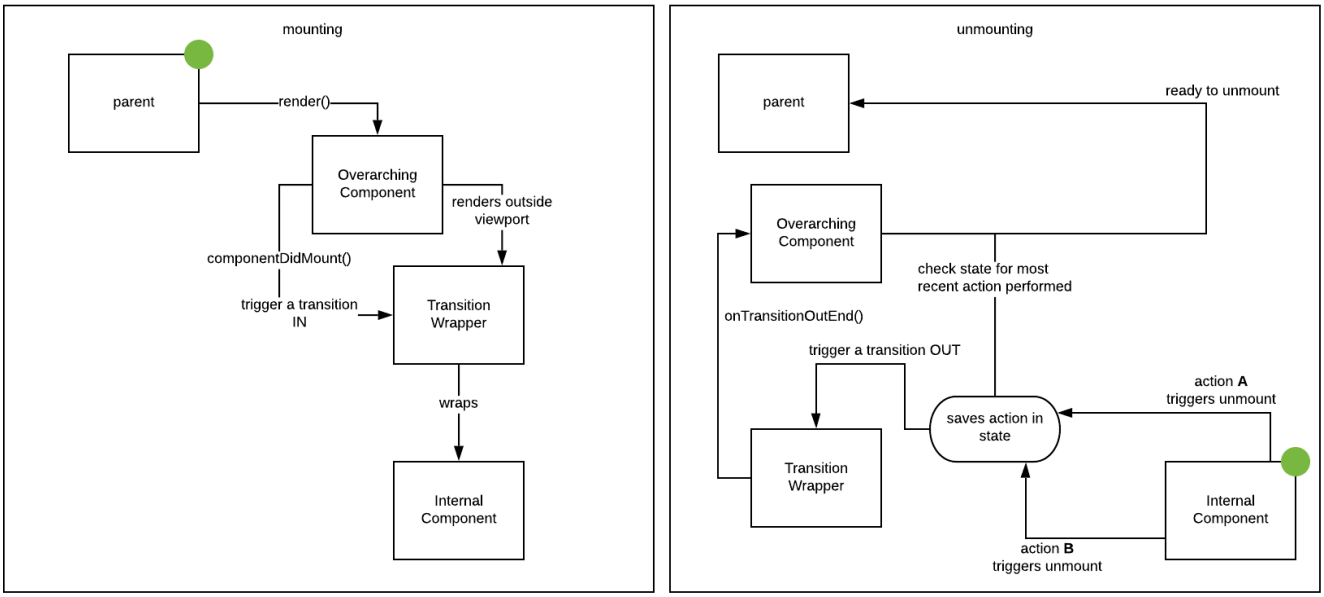


Figure 20. Workflow of transition wrapper logic.

The next piece of the puzzle was defining how the different UI screens would be structured and navigated. Due to the linear, “last-in-first-out” structure of the navigation, the stack data structure was chosen to contain the screens. Screens are mounted when a user moves forward in the navigation flow and unmounted when the user goes back. The whole stack is reset when the game session starts, that is, when the game module notifies the overarching web application that the mountain has been completely loaded and generated. The stack is restored when a game session ends, with the main menu component as the only item in the stack.

During gameplay, the application needed to implement a series of overlays, such as a progress bar, a pause/play/instructions overlay, error handling overlay, and loading overlays. Since each overlay, when needed, only exists on the screen on its own, the application implements a priority modal pattern to decide when to mount each overlay. The priority hierarchy is exclusive (only one of these overlay components may be present at once), and is defined as such in Table 1.

Priority Rank	Component	Associated Flag[1]
1	ErrorComponent	displayError
2	CompletionForm	displayCompletionForm (displayed only if not in replay mode)
3	PauseOverlay	displayPauseOverlay (modified props when in replayMode)
4	WelcomeOverlay	displayWelcomeOverlay

Table 1: Overlay priority ranking.

Certain elements in the UI, such as the pause button, song-generation button, search button, etc. are hooked to other internal modules responsible for interaction with outside frameworks. This modularity allows us to modify the implementation of how the communication happens without modifying the UI Manager itself. Outside communication hooks are defined in Table 2.

[1] This value refers to what kind of flag will be checked to determine if the said overlay will be rendered. If an overlay with higher priority rank has met their associated criteria, this value will not be checked.

Element / Component	Screen	Action	Responsible Module
Login button	Main Menu	Redirect to Spotify authorization page	APIUtil
SearchOverlay	Song Search Menu	Request strings to be searched in the Spotify database	SpotifyUtil
Generate Level Button	Song Select Menu	Fetch song metadata Send metadata to game	SpotifyUtil
		Request game to initiate level generation	StateController
Pause / Play Button	Pause Overlay	Request Spotify SDK to pause playback	SpotifyUtil
		Request game module to pause gameplay	StateController
Form Submission Button	Form Overlay	Submits form data to backend	APIUtil

Table 2: Various lines of communication within the application.

Just as elements of the User Interface trigger outgoing communication with other modules and utilities, some of the external utilities are hooked to trigger React state updates on emission of information. The details of all of these instances will be outlined here.

The progression of screens from song selection to gameplay is hinged on two separate modules: the Spotify Utility and the State Controller. This is by far the most complex chain of asynchronous events; when the user requests the song to be generated, the UI manager signals a state update indicating that the loading process for

initializing a level has started[2], then emits a request to the Spotify Utility to fetch the selected song's analysis and features from the Spotify database. Once the response arrives asynchronously, the UI manager emits a request through the State Controller to the game containing the song information, and immediately disables and resets the menu stack. That means that in the case that anything fails after the song is fetched and the game begins to generate the level, the player will be taken to the main menu. Otherwise, the loading state is disabled once the game notifies the UI manager through the state controller that the game is ready to play. A listener, that was set up on construction, upon receiving this state notification, disables the loading screen, enables the progress bar, the song information display, and the pause overlay. It also signals to the Spotify Utility to begin playback of the selected song. This leads to the game to be playable with the appropriate overlays mounted, and the main menu to be inaccessible. This flow is demonstrated in Figure 21.

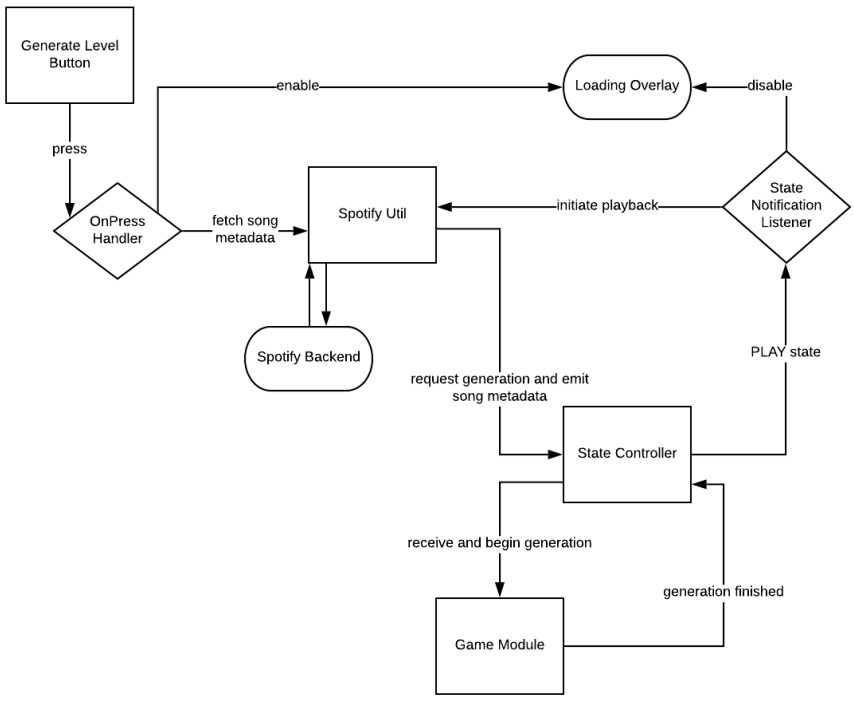


Figure 21. Workflow of selecting a song and generating the level.

[2] Which in turn, triggers a render cycle that displays the loading overlay

The progress bar overlay's state is strongly dictated by the state of the Spotify Web player as well as the state of the game. According to the Spotify Web Playback SDK API documentation, the web player can emit a certain number of events, as well as notifying state changes. It is important to note, as well, that the SDK can also randomly emit the web player's event, even if no changes have been applied to it. Accessing the Spotify Web Player's (SWP) state is an asynchronous operation that relies on contacting external services over the internet, therefore making it unreliable as an accurate way to synchronize any UI elements with the progress of the song; this operation also takes non-negligible processing power to execute, something that is unacceptable when most of the computer's processing power should be dedicated to the game engine.

The solution found to this conundrum was to take advantage of built-in CSS transitions and stipulate with low frequency scheduled checks of the SWP's state what the song's progress rate is. For example, if checks are done every 5 seconds, and the first check shows that the playback position is 0, I can assume that the bar should move until the position of 0 + 5 seconds, and that transition should happen for 5 seconds, therefore making the bar move 1 second per second–real time.

An issue with this structure is that updates to the bar are done not only when the progress bar overlay requests it, but also when the SWP's state changes for any reason, be it external or internal. So the element may receive an update to the playback position sooner than expected, therefore making the bar progress slightly faster than real time, but moving behind what the real position should be. This, however, is barely noticeable, as players are not required to make any precise measurements or derive any data from the progress bar. It is simply there to provide an approximate measurement of how much they have played through the level. It is relevant to mention that the progress bar element also contains a display of the player's score, which is updated reactively when a notification of change in score, encoded by the enumerator *SCORE_CHANGED* (Appendix B), is received by the element.

3.4.1b. Static Web site

The rest of the website is a fairly simple piece of software in comparison to the rest of the project. It displays two additional pages, a *leaderboards* page and an *about* page. The leaderboards page simply fetches the top 10 scoring sessions from the REST API and displays them in order on the screen, alongside the player who reached that score and the song chosen.

The about page utilizes a Markdown parser to render a summarized version of this report to any users who are curious about how this game came to be created. It provides links to the creators' profiles and more information about the game.

3.4.1c. API utility

An important part of software development that developers strive for is to create something in which a change can be accommodated by only having to modify a single module (VanHilst, 1996). A great example is any part of the application that requires communication with a REST API (detailed in section 3.4.5). If communications with an API aren't abstracted into a package function, any changes to REST requests or endpoints may break the application in ways that are difficult to fix: how can one find every single place in the code where a certain endpoint may be referenced? This is the goal of this utility: not only to abstract backend endpoint interfacing into a simple function, but to also preprocess outgoing data and post-process incoming data as needed.

3.4.2. Using the Spotify Web API

Spotify provides an api for track search, track metadata, and other track information, as well as user authentication, web playback, and other functionality. The operations needed for the basic function of our project were broken up into two categories: back end and front end, and were all encapsulated in their respective versions of a Spotify Utility module.

There were several reasons for choosing the Spotify APIs to accomplish the tasks outlined above; the most noteworthy ones are the ease of use and breadth of information available. The APIs were designed so that developers can use its resources (song playback, analysis) without needing to host all the song files or building an audio analyzer themselves. These APIs also offer a lot of information about the vast majority of songs in the Spotify databases, and as outlined in Section 3.3.3, this information is central for the procedural building process involved in creating the levels. An alternative to this would be to use a competing music provider service, but no other competitor offers as much song metadata as Spotify, or requires a similarly low bar of entry for developers to make use of their services. Another alternative would be implementing an in-house audio analyzer, but the scope of this alternative would overshadow the goal of the project.

The front end Spotify Utility (FESU) is a wrapper for not only RESTful operations on the Spotify API, such as searching tracks and fetching metadata, but also for initializing the Spotify Web Playback SDK (SWPSDK), queueing tracks, and toggling playback. Through this module, other parts of the application can have access to the operations above without the overhead of the several lines of code needed to mount, connect, and initiate playback in the standard SWPSDK.

The FESU also implements an observable stream of SWPSDK notifications. Any state changes regarding the SWP are piped through this observable; this is what is referred to in section 2.6.1a. when describing how the progress bar is updated asynchronously. These notifications range from loading the package, to initializing the web player, to connecting, to other changes in the playback such as pausing, playing, changing song, and any errors.

The back end Spotify Utility (BESU) centers around wrapping the whole sequence of RESTful calls needed to properly execute the OAuth 2.0 process as outlined by the official Spotify authorization documentation (“Spotify Auth. Guide”). It is only utilized when authorizing new users and validating existing users, meaning any other Spotify service needs are piped directly to Spotify’s back end, and not proxied through the website’s backend server. Figure 22 illustrates the implementation of this service.

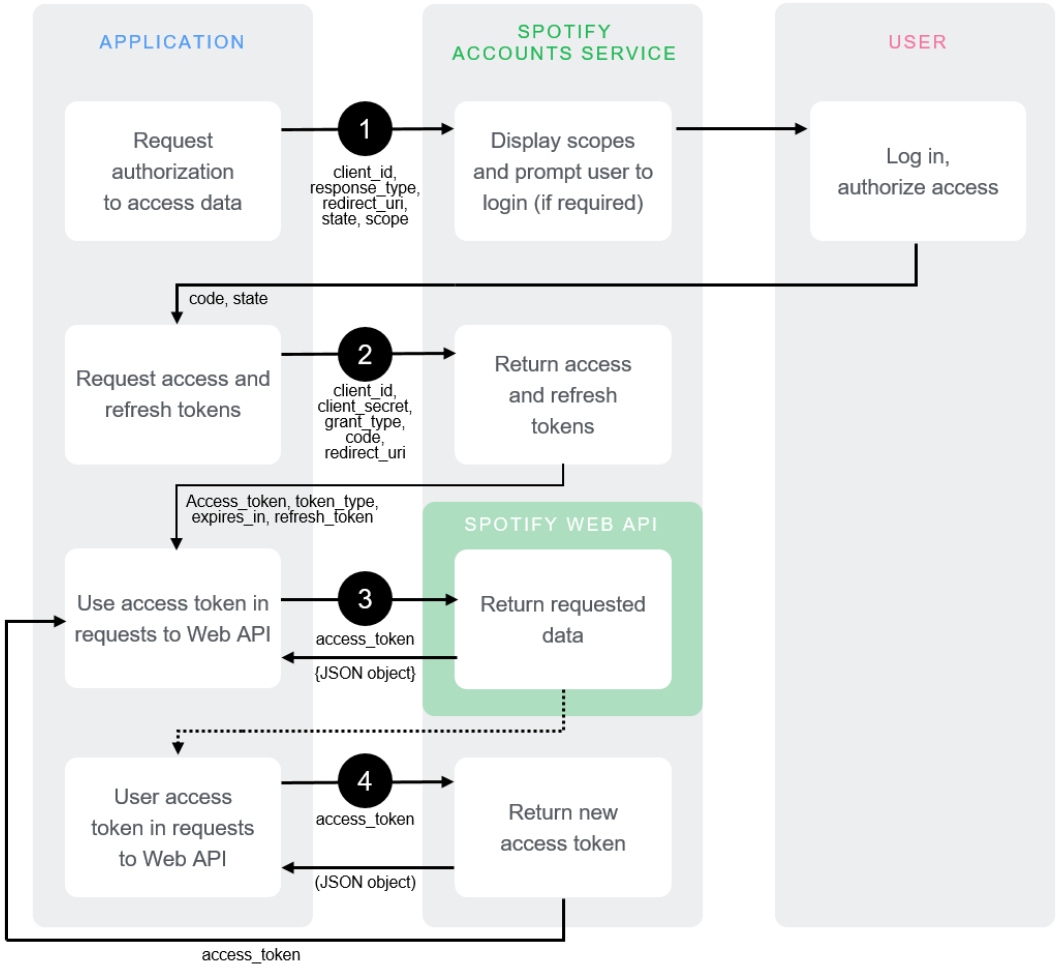


Figure 22. Spotify Web API implementation. Source: URL.

3.4.3. Input manager

The input manager module is used to abstract key press events into game actions, thereby removing the need for the game module to assign input handling directly to a specific key press. The benefits to this abstraction do not end there— by centralizing game inputs, wrapped in a defined format, through a single observable, allows the developers to store and replay a sequence of inputs, as well as binding new inputs on the fly. It also permits the game to handle actions instead of specific key presses, meaning the game module doesn't need to differentiate between different key assignments or between real-time gameplay or a replay.

```
function emitEventAt(index) {

        // will throw error if indexed out of bounds
        const currentMove = this.actions[index];

        // emit move
        this.emit(currentMove);

        // check for next move
        let nextMove;

        // if index is appropriate
        if (index + 1 < this.actions.length) {

                // set next move
                nextMove = this.actions[index + 1];

        }

        // if we have a next move, chain it
        if (nextMove) {
            this.currentTimeout = setTimeout(
                this.emitEventAt,
                nextMove.timestamp - currentMove.timestamp,
                index + 1
            );
        } else {
            // if not, call the end function
            this.onEndOfInputs();
            this.terminateReplay();
        }
}
```

Figure 23. Code snippet in the context of ReplayUtil.

The input manager provides functionality for recording inputs, pausing said recording, resetting and fetching the saved history of inputs. The input manager also relativizes the timestamp of the recorded input history to the start of the session, therefore making it easier to synchronize the inputs to the point in the song in which they were emitted. The reverse process is also an intended feature; the input manager can also receive an input history in the form of an array of inputs and replay them in the correct times by recursively chaining vanilla JS timeout calls. See a snippet of this functionality in Figure 23 above. The full module is provided in Appendix C.

There is an important discussion to be had with regards to the way this replay feature was implemented. First, it is important to outline the tradeoff involved in the choice of using **`setTimeout()`** to create this feature. The current implementation allows the module to be completely independent of the game engine to the point where the game does not need to be aware of the context in which it is receiving an action, whether it is a replay or not. This allows for the game to be loosely coupled with the window event stream, meaning it doesn't need to abide to any specifications other than the one defined by the team when determining the IM-Game interface, and therefore made it easier and quicker for the developers to implement input observers.

The big drawback here is the notorious inaccuracy of this JS feature (Edwards, 2010), along with the uncertainties that add up (getting the precise time in which a key was pressed, getting the exact time in which that time compares to the position of the song, playing that input back in the precise frame in the game, etc), make up for a completely inconsistent replay system. An alternative to this system, or possible solution, would be to transfer the responsibility of time-keeping to the frame ticker of the game. By implementing an *action stack* that gets an action popped at every frame, the team could have frame-perfect tracking of the time in which an action was performed. This "n^{th}" frame could be saved as another object property (or replace *timestamp*) and be used to replay actions by comparing whether the action on the top of the stack has a frame number that matches the current frame that is being played, and performing it if the comparison passes. This would require a complete restructuring of the way actions are tracked in-game, as well as rethinking the role of the InputManager in the project, both of which would be impossible in the scope of this project, but encouraged if a different team decides to pick up the torch.

3.4.4. State controller

The state controller module implements a two-way asynchronous notification framework between the Game Module and the UI Management Module. It does so by abstracting existing rxJS Subject operations and subscriptions into more verbose methods; the emission of information through the Game-to-UI stream is wrapped in the method *notify*, because it's intended purpose is to notify the UI of a state change. Additionally, a subscription to said stream is achieved by calling the method *onNotificationOf* and choosing which state change enumerator should be filtered for. This is useful for keeping the UI up to date to whatever is going on in the game module side (game finishes loading, notifies app to initiate track playback). The reverse is also implemented. The emission of information through the UI-toGame stream is wrapped in the method *request*, because it's intended purpose is to request the game to change its state or perform an operation. Additionally, a subscription to said stream is achieved by calling the method *onRequestTo* and selecting which state change enumerator should be filtered for.

3.4.5. Backend server architecture

The back end NodeJS powered server serves three purposes: serving static data to the client application, implementing OAuth 2.0 for Spotify authorization, and proxying data to an external SQL server.

Serving static data is fairly simple, as *express.js* provides an easy API for creating RESTful API endpoints. This is done to serve any GET requests and to send the client application when a user reaches www.tune-mountain.com. The authentication flow is replicated inside the Spotify Service Utility module, and the details of implementation are outlined in Section 2.6.2. The proxy functionality is defined in a proxy router module, and it simply redirects GET and POST requests to the SQL API endpoints defined on GitHub. The responses are then sent back to the client.

4. Evaluation

4.1. AlphaFest

We had a good start with playtesting at AlphaFest, with thirty people playing our game, each of them playing at least two songs and filling out our survey of questions (see Appendix D). The development of the project over the course of A and B term led to the following features included in the project by the event: our initial procedural generation algorithm of the mountain slope, complete Spotify integration allowing us to pick any song from Spotify, collectible coins laid on the slope in two different patterns, and the ability to jump. One of the main pieces of feedback that we received from players was that they didn't feel like they needed to do anything: they didn't really have an objective other than watch the snowboarder go down the slope. This was because we had no score, no tricks, and no instructions for the player.

The other main feedback that we got was that the game felt very slow, and that it did not really fit the speed or the energy of the song they were listening to. This was the problem described in Section 3.3.4. The ways we got around this problem included making the player much smaller, zooming the viewport in on the player, and adding trees in the background that the player could pass by to give a better impression of how fast they were going. The two patterns of coins we were testing was a coin on every beat and a couple of groups of four coins on beat per section. Players didn't particularly care for either pattern however; they felt meaningless to collect when they were on beat and no one really understood the purpose of the group of four coins pattern. Most players were not impressed by the jump in this iteration of the project; it didn't seem to have a purpose and many testers complained that it was too weak. We added a little more oomph to the jump in future iterations. Finally, we learned that only a little over half of our playtesters had a Spotify Premium account. Since you need a Premium account in order for the music to play, this was a fairly significant detail; however we decided that the amount of work it would take to shift tactics and not use Spotify's Web API would not be worth it.

4.2. C-Term playtesting

By the end of C term, we were ready for another round of playtesting. Features included in this round of testing were: the faster player thanks to the smaller and zoomed in physical world, sprite swapping, play/pause functionality, and one trick that increased the score when landed correctly. The whole project was also now hosted on a website, allowing for easy testing and distribution.

In addition, we decided to code in our survey questions, which can be found in Appendix E, and have them appear immediately once the song is finished playing, allowing for a simpler process than having to switch over to a Google form or Qualtrics survey after every play session. Since by this point we had submitted our project to the IRB, that meant we had to follow proper IRB protocol while administering these playtesting sessions. Before the beginning of each session, one of us read the opening briefing found in Appendix F and had the playtesters fill out the informed consent form found in Appendix G. This allowed us to use specific information from the survey in this report as long as we did not associate any information with any playtester's names.

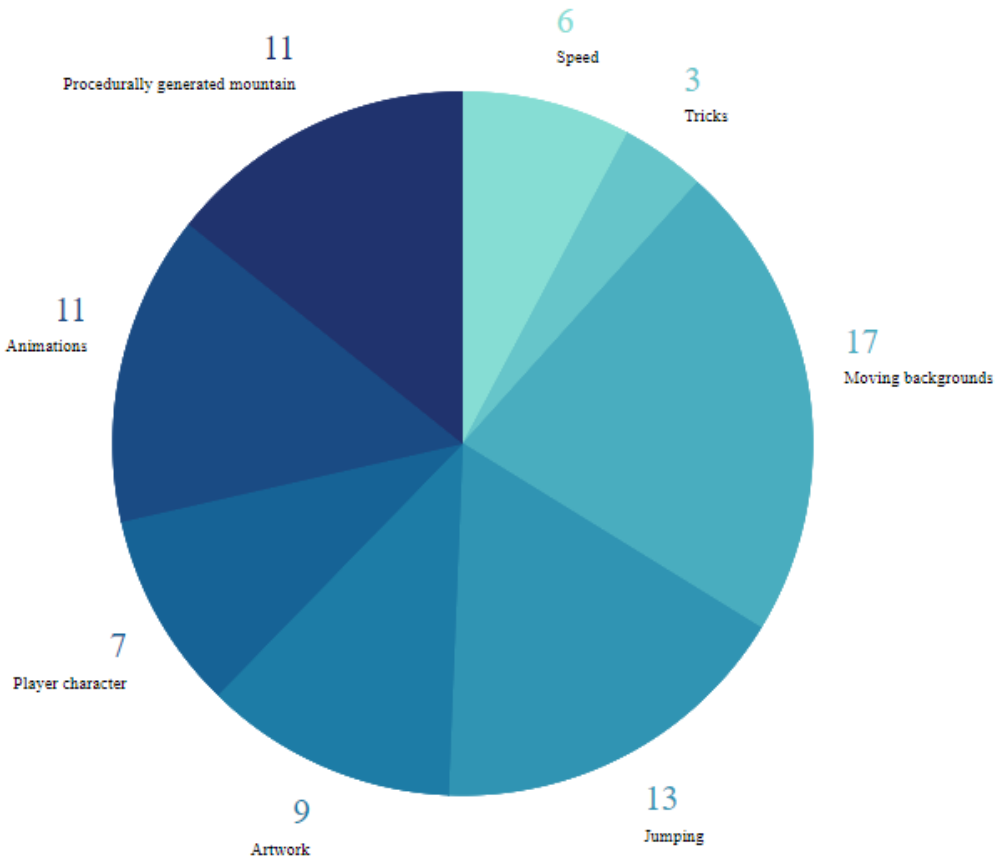


Figure 23: Responses of what playtesters enjoyed about *Tune Mountain.*

Overall impressions of the game were much more positive now compared to AlphaFest; Figure 23 above shows the parts of the application playtesters enjoyed during their session, from 28 form submissions. Most players enjoyed the various artwork and animations, and the amount of people who said they liked the jumping was encouraging, showing that it had been improved since the AlphaFest playtesting. Even though the jumping now had a purpose, allowing players to perform tricks, it made sense that very few people said they enjoyed the tricks at this stage of the project. At this point, due to unforeseen circumstances, we were only able to include one dummy trick in the game. It worked properly, swapping to the animated sprite and giving points if completed successfully; however the animation was just cycling through the words Trick 1 to Trick 10 before swapping back to the idle animation. The fact that only six people said they enjoyed the speed was interesting to us, as this was not quite backed

up by the response to question 7 in our survey, "Did you feel like you were moving at an appropriate speed for the type of song you chose?". As shown in Figure 24, 17 out of 28 responses felt that the speed for their song was just right, and not a single person felt it was too fast.

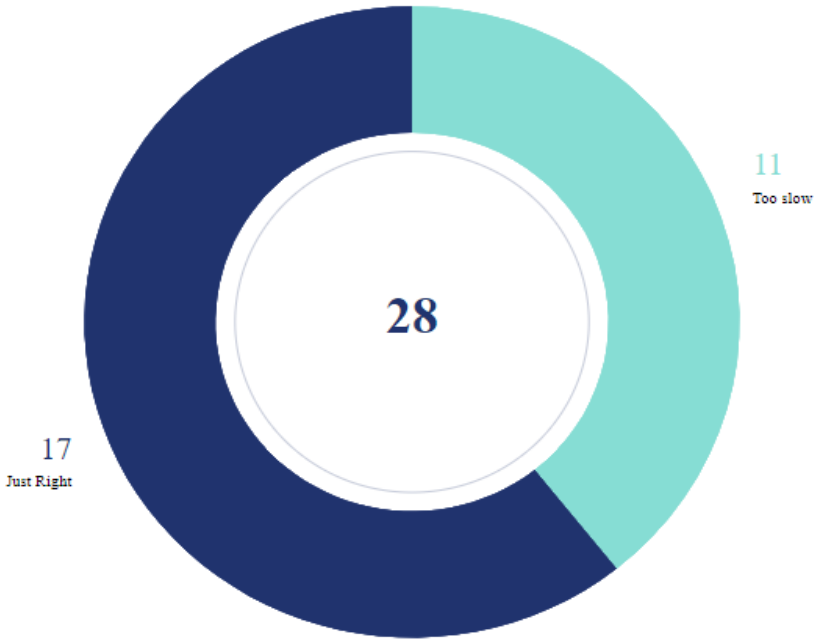


Figure 24: Responses of how playtesters felt about the speed of *Tune Mountain.*

Since one of the major focuses of this project was the synchronization with the music, we felt it was important to ask our playtesters about the topic. We asked them "To what degree did you feel the elements in the level were in sync with the music you were listening to?" on a 1-6 scale, and their answers can be found in Figure 25 below.

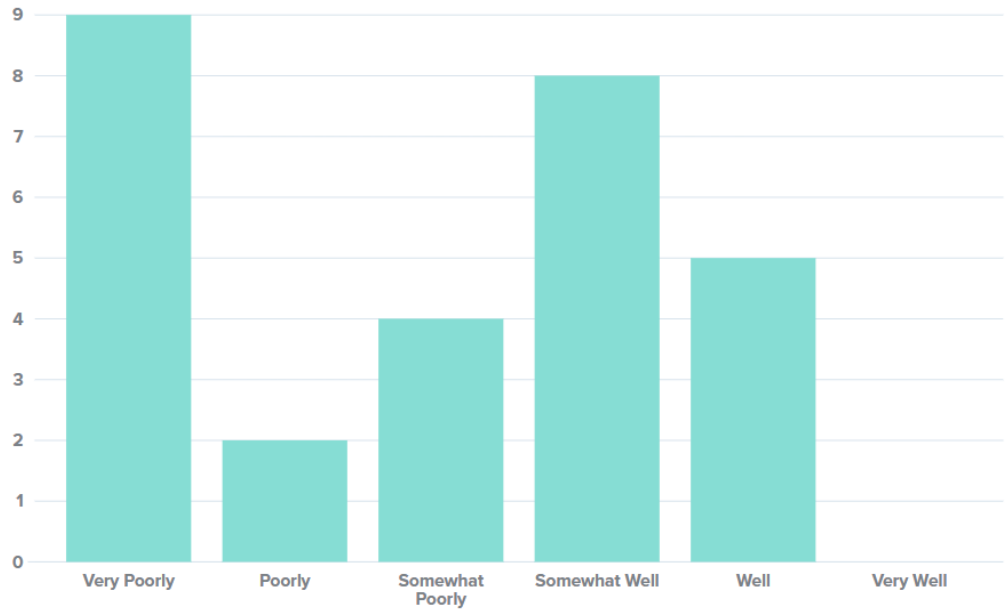


Figure 25: Responses of how playtesters felt about the synchronization of elements in the level to the music of *Tune Mountain.*

The opinions were pretty split, with 46% of the playtesters giving a positive response to the synchronization and 54% giving a negative response. At this point in the project, we did not have a shader on some of the background mountain layers or trees on the slope of the mountain, both moving in time with various values from the Spotify API. So at this point in the project, this question is essentially asking how they felt about the mountain slope matching up with the song. We understood these responses, since basically the mountain would either end way too early, or there would be more mountain on screen with the song ending. We ended up polishing our mountain curve generation in D term to better fix it for songs with consistent tempos, however a limitation of the Spotify API is that it could not give us tempo values for each section of the song, so we could only fix so much in the limited time we had without a major restructure of our algorithm.

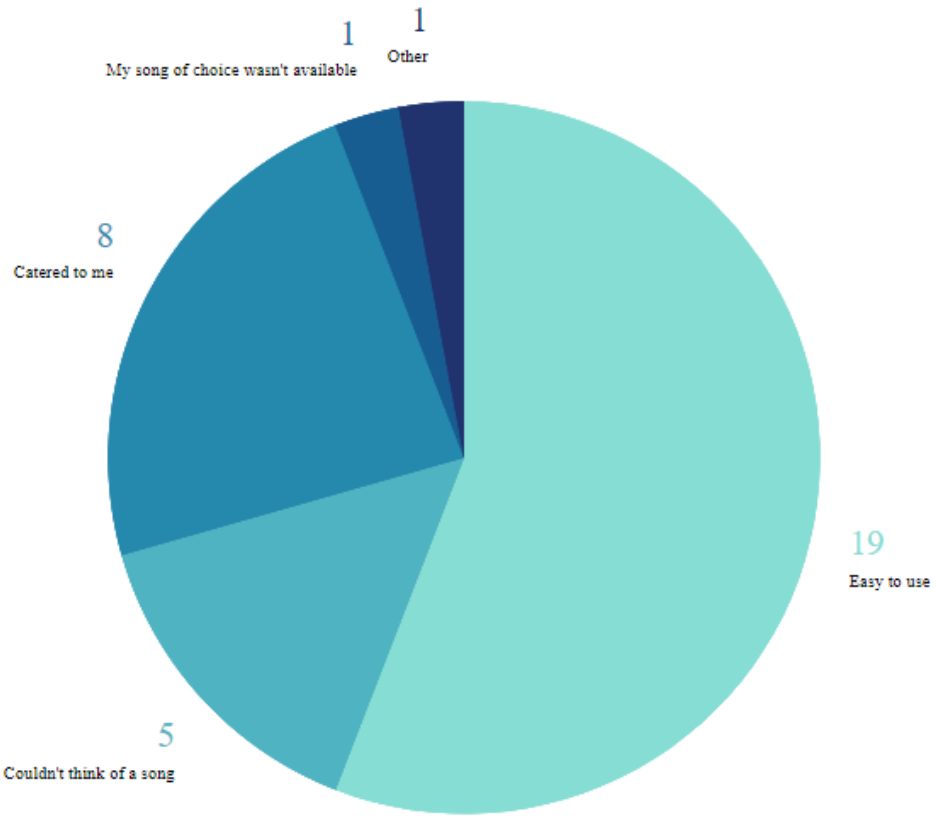


Figure 26. Responses of how playtesters felt about the song selection process of *Tune Mountain.*

The last major topic we wanted to get opinions on from playtesters was the song selection process. We asked them "Select all that describe your experience choosing a song", and their responses are found in Figure 26 above.

We were pleased to find that at least 68% of our users found the process of searching by a song by name to be quite intuitive, showing us that we did not need to change that process. A couple users had trouble thinking of a song initially; however they were able to think of a song to use after a few minutes or after pulling up a music app on their phone.

Finally, the other major advantage of conducting playtesting sessions is that it can point out bugs you did not discover on your own while developing, and playtesters can give ideas for new things to implement into the project. There were two major problems discovered by watching our playtesters interact with *Tune Mountain*, both associated with tricks. The first bug was that tricks were able to be initiated on the slope rather than only in the air. This was easily fixed by adding a check for if the player was on the mountain slope when the game is trying to process an action input. The second bug was that the character would stay on the first frame of their trick if the corresponding keyboard button was held down. This was also fixed with a simple solution, by adding a check for if a trick animation was currently playing. Some suggestions we liked from our playtesters included introducing a combo meter for consecutive successful tricks, and showing the instructions on screen at all times, rather than only on the loading screen and pause menu. Showing the instructions on the screen the entire time while playing makes it much easier for new players to pick up the gameplay, and adding a combo meter would give players more of an incentive to time their tricks successfully. Both features were implemented in D term.

5. Postmortem

5.1. What went right, what went wrong

This project was very fulfilling because it explored an idea that has not been widely explored in game development. It was exciting to take existing building blocks (web frameworks, past audio visualizer games, procedural generation techniques), and mix them into something unique. This was especially interesting because it was built with the purpose of being accessible to anyone that has a web browser, and therefore it is readily available for anyone with access to a computer and a web browser.

Tune Mountain was successful in exploring the limits of existing web services and frameworks, because we accomplished our experience goal and made it extremely accessible for people to enjoy it. The team put in a lot of effort learning the depths of the tools and used them creatively to tie them together in a way that had not been done before. There were, however, several downsides to the team's plan to go about this project. The most glaring is the fact that the scope of *Tune Mountain* was in many ways unprecedented, so there were very little resources to help with the planning and implementation of the game, and the 'best practices' had to be found by the team through trial and error. An example of this is the business logic for the screen stack used in the song selection UI flow. This could've been done more robustly by using flags in the React component state to render the appropriate components, rather than saving the components themselves in the React state. In other words, more development cycles and/or a better understanding as well as more experience with the technology stack would result in a more refined product.

As mentioned in Section 3.2.2, there were two big decisions made early on in this project that had a significant impact on our workflow: the decisions to choose PIXI.js and Planck.js as the frameworks of our application. A lot of our work during A and B term involved solving the problem of combining these two frameworks and getting them to work simultaneously; getting the curves of the PIXI.Graphics object to line up appropriately to the curves of the Planck physics world. This was much more of a math problem than initially anticipated, especially with the limited amount of information we

had on Planck.js. Maybe the problem would have been easier solved if we had used Matter.js instead thanks to its documentation, or how much the project would have changed from an art standpoint if we had decided to use Phaser.

Using Spotify's web API helped us immensely to create the experience we were aiming for; however, it comes with its restrictions. Even though we were able to stream the song through the web playback SDK, we were not able to acquire any audio information using web audio features. Not only does Spotify explicitly restrict any information from being cached or analyzed, but also the Spotify web playback SDK is a black box that only allows play, pause, and seek features. So we could not acquire any FFT analysis of the music that is playing even if we wanted to. This meant we were limited to the data that they decided to provide us with, which proved to be not as much as we would have hoped during development. In order for our current mountain generation algorithm to work properly with all songs, we would need tempo information for each section of a song, instead of a single tempo value for the entire song. As it is right now, our project works best on songs that have a consistent tempo throughout the song.

One unfortunate problem we faced was the change into a fully remote development process for the last term of the academic year. Due to COVID-19, many things have been affected, including our development cycle and our ability to have dedicated work meetings. It was still fortunate that we were able to publish the game on the website before March, so that we were able to get playtesting while continuing our development. This situation pushed us to focus more on polish of the features we had, instead of working on other features we planned on having.

5.2. Future plans and possible expansions

Future additions and improvement plans for this project would include changes and expansions to the three main components. For the React application, the most glaring issues come from the monolithic `HUDOverlayManager` component. A thorough refactor and modularization would improve performance and get the codebase closer to the React standard, and it would make the code more readable. For the

InputManager, as outlined in Section 3.4.3, dramatic structural changes to both the `ReplayUtil` and the way the Game module reads inputs would be needed in order to create a more robust and accurate replay feature; a possible effective approach would be to build an adapter (Gamma et al., p. 157) between the interfacing modules in order to convert inputs from an asynchronous event-based solution into a queue, where the game checks for inputs on a frame-by-frame basis. Our team also understands that the documentation for a lot of parts of this project is lacking, and overhauling that would help future efforts to add to the game.

An idea that we could not explore enough was to have multiple dynamic shaders for the background, which would take in the song analysis data we get from the Spotify API, and use them to make the background “dance” to the music. We feel that we could have tackled the audio visualization aspect of the game in a more varying, and interesting way. Since PIXI allowed us to attach multiple shaders to a single sprite, we definitely could have explored making interesting, dynamic visuals for the background, but unfortunately it fell out of priority. Ideally we were thinking about separate shaders for different genres of songs.

There were many other gameplay elements that we thought of that had to be cut due to scoping reasons, and thus could be implemented in the future. The biggest one was including other gameplay elements for the player to slide down other than the mountain slope, such as ski lift lines and ramps. These new objects would create opportunities for players to get extra points since they followed a line of coins onto a ski lift line, or boosted off a ramp in order to get some extra air time and complete more complicated tricks. These were something that we really wanted to include, and even had the art made by Joy, however due to the amount of effort using Planck.js took and the COVID-19 crisis affecting our workflow, this cool but unnecessary game mechanic never made it in.

Another idea we considered were powerups for the player, such as a jump boost and a speed boost, however we felt like that would mess too much with the synchronization of the mountain length, and would take way too long to implement correctly. We also wanted to allow players to spend coins they collected to buy different outfits for the player character. Finally, we could try to incorporate a theme or story a bit more into the gameplay. We thought of a theme early in development that our snowboarder character is dreaming and going down a mountain of their music with gameplay elements such as bouncy clouds for them to jump on and music notes to collect. Adding more of these types of elements could help enhance the overall experience for the player.

6. Conclusion

6.1. Prospects

The game *Tune Mountain* achieved many of the design goals set for the project, and has been successfully published in a browser for easy access. The entire game is live and working at this URL, as of May 2020. The entire codebase is distributed between four repositories: one for the website, one for the game, one for the Spotify client, and one for the input manager. We decided to make it this way to keep the entire project more modularized, and we also made the input manager into an NPM package. We also put off many things in our development that were mentioned in the previous section that can be added to the game.

We had hoped to spread the word of this project, and wanted to promote it on different platforms or events outside of WPI; however, we found an article in the Spotify Developer Terms of Service that reads as follows:

"Games and trivia quizzes. Unless you receive Spotify's written approval, you shall not use the Spotify Platform to incorporate Spotify Content into any game functionality (including trivia quizzes)."

“Synchronization. You may not synchronize any sound recordings accessed through the Spotify Platform with any visual media, including any advertising, film, television program, slideshow, video, or similar content.

i. Mixing, overlapping and re-mixing. You may not, and you may not permit any device or system used in connection with the Spotify Service to, segue, mix, re-mix, or overlap any Spotify Content with any other audio content (including other Spotify Content)”

We understand that these restrictions would prevent such a project, but because this is purely an academic and educational project, with no revenue or commercial value tied to it, it did not affect our development. It did mean that we wouldn’t be able to show it off in platforms like YouTube, or events like PAX East, since a commercial success would not be beneficial for the continuation of this project.

6.2. Takeaways

This project has taught us a great deal. We have learned how to publish a website under a specific domain. We have had great experience designing, developing, and improving a game from the ground up. But the biggest takeaway we had from this project will be the ability to cope with different working environments, creating timelines that would fit everyone's schedules, and creating a working, stable, and sustainable programming project. Being able to explore each other’s creativity, problem solving skills, and handle issues that came up has helped us to be more collaborative and understanding individuals. It has made us learn to respect each other's time and effort, and focus on coming up with a solution rather than focusing on who or what caused the problem. We were all proud of ourselves for accomplishing a product that met our expectations, and our advisors, as well as meeting our experience goal of creating an engaging audio visual experience.

Works cited

“A Rhythm Violence Game.” *Thumper*, 10 October 2016. Web. URL. Retrieved 30 April 2020.

Aga1n. “AudioSurf 2 :: Marshmello - Alone [Ninja Turbo].” YouTube, 1 June 2016. Web. URL. Retrieved 30 April 2020.

Alto's Adventure. 19 February 2015. Web. URL. Retrieved 30 April 2020.

Baker, Harry, et al. “How To Download And Install New Custom Songs On Beat Saber.” *UploadVR*, 29 Apr. 2020, URL. Retrieved 30 April 2020.

Beat Saber. 1 May 2018. Web. URL. Retrieved 30 April 2020.

“Bezier Curve.” The Modern JavaScript Tutorial, 29 Mar. 2020, URL. Retrieved 4 May 2020.

“Building a Parallax Scroller with Pixi.js: Part 1.” *Yeah But Is It Flash RSS*, 27 Sept. 2016, URL. Retrieved 2 September 2019.

Capello, David. “Aseprite.” *Aseprite*, 2001. URL. Retrieved 30 April 2020.

Dink, Bah. *Guitar Hero - Cliffs of Dover - Expert 100% FC - HD. Youtube*, 7 June 2012, URL. Retrieved 30 April 2020.

Edwards, James. “Creating Accurate Timers in JavaScript.” *SitePoint*, SitePoint, 23 June 2010, URL. Retrieved 30 April 2020.

Emergent Studios. “Line Rider.” *Line Rider*, URL. Retrieved 30 April 2020.

Fitterer, Dylan. Audiosurf 2: Ride Your Music, 2015, URL. Retrieved 30 April 2020.

Figatner, David. *Pixi-Viewport API Documentation*, URL. Retrieved 18 September 2019.

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 2015. Retrieved 30 April 2020.

Groves, Mat. “PixiJS.” PixiJS, 2020, URL. Retrieved 30 April 2020.

GXZ95, director. SSX (PS2 Gameplay). Youtube, 11 Feb. 2012, URL. Retrieved 30 April 2020.

Huang, Aaron, director. *Line Rider - Bohemian Rhapsody | SYNCED | LR Vanilla. Youtube*, 5 Mar. 2019, URL. Retrieved 30 April 2020.

“Matter.js.” Code by @Liabru, URL. Retrieved 30 April 2020.

Matthey, Julien. “JM_IMPACT_01c - Snow on cement”, *Freesound,* 10 October, 2012, URL. Retrieved 4 May 2020.

Oliveira, Nuno Filipe Bernardino. "Music-Based Procedural Content Generation for Games." 7 August 2015. Retrieved 4 February 2020.

OpenJS Foundation. “Express JS 4.x API Documentation.” Express 4.x - API Reference, OpenJS Foundation, URL. Retrieved 30 April 2020.

OpenJS Foundation. “NodeJS.” NodeJS, OpenJS Foundation, 27 May 2009, URL. Retrieved 30 April 2020.

Parecki, Aaron. “OAuth 2.0.” OAuth, OAuth, Oct. 2012, URL. Retrieved 30 April 2020.

Pixi Particles. URL Retrieved 4 April 2020.

“Pixijs/Pixi.js.” *GitHub*, URL. Retrieved 13 September 2019.

“Pixijs/Pixi-Particles.” *GitHub*, URL. Retrieved 4 April 2020.

PixiJS API Documentation, URL. Retrieved 13 September 2019.

PixiJS Examples, URL. Retrieved 13 September 2019.

PixiParticlesEditor, URL. Retrieved 4 April 2020.

“React Component Documentation.” *React*, URL. Retrieved May 4, 2020.

“React Usage Statistics.” *BuiltWith*, URL. Retrieved 4 May 2020.

“React – A JavaScript Library for Building User Interfaces.” *ReactJS*, URL. Retrieved 4 May 2020.

ReactiveX. “Reactive Extensions Library for JavaScript.” RxJS, ReactiveX, 2 Sept. 2015, URL. Retrieved 4 February 2020.

Rrvirus, director. Alto's Adventure (By Snowman) - IOS / Android - Gameplay Video. *Youtube*, 19 Feb. 2015, URL. Retrieved 30 April 2020.

“Shakiba/Planck.js.” *GitHub*, URL Retrieved 30 April 2020.

Spotify. “Web Playback SDK Reference: Spotify for Developers.” Web Playback SDK Reference | Spotify for Developers, Spotify, URL. Retrieved 2 September 2019.

“Spotify Authorization Guide.” *Spotify for Developers*, URL. Retrieved 2 September 2019.

Sir James. “Thumper Gameplay Level 1.” *Youtube*, 11 Oct. 2016, URL. Retrieved 30 April 2020.

TechMagic. “React vs Angular vs Vue.js-What to Choose in 2019? (Updated).” *Medium*, Medium, 24 Apr. 2020, URL. Retrieved 8 May 2020.

VanHilst, Michael and Notkin, David. *1996. Decoupling change from design. SIGSOFT Softw. Eng. Notes 21, 6 (October 1996), 58–69. URL*. Retrieved 30 April 2020.

Appendix A. Spotify analysis & features objects

Audio analysis object

KEY	VALUE TYPE	VALUE DESCRIPTION
bars	an array of time interval objects	The time intervals of the bars throughout the track. A bar (or measure) is a segment of time defined as a given number of beats. Bar offsets also indicate downbeats, the first beat of the measure.
beats	an array of time interval objects	The time intervals of beats throughout the track. A beat is the basic time unit of a piece of music; for example, each tick of a metronome. Beats are typically multiples of tatums.
sections	an array of section objects	Sections are defined by large variations in rhythm or timbre, e.g. chorus, verse, bridge, guitar solo, etc. Each section contains its own descriptions of tempo, key, mode, time_signature, and loudness.
segments	an array of segment objects	Audio segments attempts to subdivide a song into many segments, with each segment containing a roughly consistent sound throughout its duration.

tatums	an array of time interval objects	A tatum represents the lowest regular pulse train that a listener intuitively infers from the timing of perceived musical events (segments). For more information about tatums, see Rhythm (below).

Time interval object

This is a generic object used to represent various time intervals within Audio Analysis. For information about Bars, Beats, Tatums, Sections, and Segments are determined, please see the Rhythm section below.

KEY	VALUE TYPE	VALUE DESCRIPTION
start	float	The starting point (in seconds) of the time interval.
duration	float	The duration (in seconds) of the time interval.
confidence	float	The confidence, from 0.0 to 1.0, of the reliability of the interval.

Section object

KEY	VALUE TYPE	VALUE DESCRIPTION
start	float	The starting point (in seconds) of the section.

duration	float	The duration (in seconds) of the section.
confidence	float	The confidence, from `0.0` to `1.0`, of the reliability of the section's "designation".
loudness	float	The overall loudness of the section in decibels (dB). Loudness values are useful for comparing relative loudness of sections within tracks.
tempo	float	The overall estimated tempo of the section in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.
tempo_confidence	float	The confidence, from `0.0` to `1.0`, of the reliability of the *tempo*. Some tracks contain tempo changes or sounds which don't contain tempo (like pure speech) which would correspond to a low value in this field.

key	integer	The estimated overall key of the section. The values in this field ranging from `0` to `11` mapping to pitches using standard Pitch Class notation (E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on). If no key was detected, the value is `-1`.
key_confidence	float	The confidence, from `0.0` to `1.0`, of the reliability of the *key*. Songs with many key changes may correspond to low values in this field.
mode	integer	Indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. This field will contain a `0` for “minor”, a `1` for “major”, or a `-1` for no result. *Note that the major key (e.g. C major) could more likely be confused with the minor key at 3 semitones lower (e.g. A minor) as both keys carry the same pitches.*
mode_confidence	float	The confidence, from `0.0` to `1.0`, of the reliability of the *mode*.

time_signature	integer	An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). The time signature ranges from `3` to `7` indicating time signatures of “3/4”, to “7/4”.
time_signature_confidence	float	The confidence, from `0.0` to `1.0`, of the reliability of the *time_signature*. Sections with time signature changes may correspond to low values in this field.

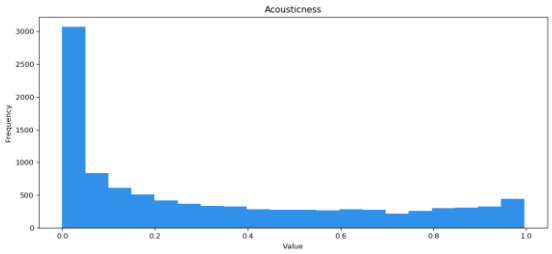
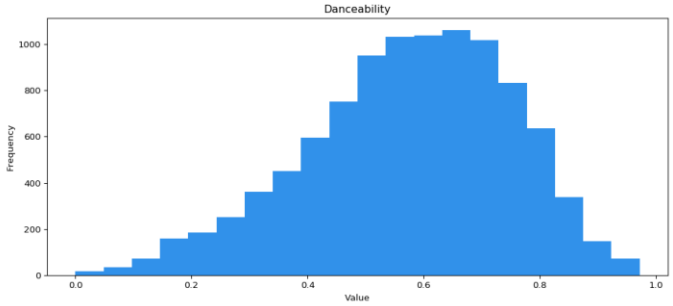
Segment object

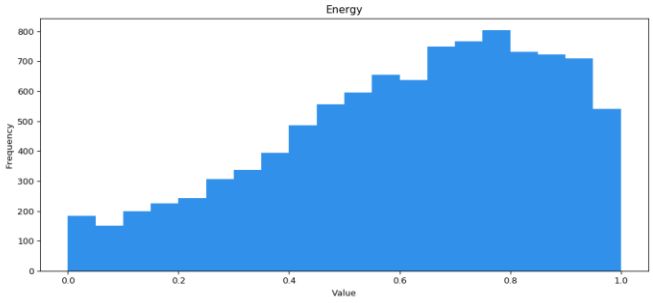
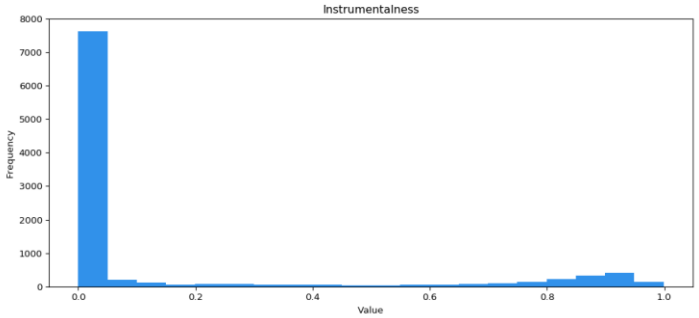
KEY	VALUE TYPE	VALUE DESCRIPTION
start	float	The starting point (in seconds) of the segment.
duration	float	The duration (in seconds) of the segment.
confidence	float	The confidence, from `0.0` to `1.0`, of the reliability of the segmentation. Segments of the song which are difficult to logically segment (e.g: noise) may correspond to low values in this field.
loudness_start	float	The onset loudness of the segment in decibels (dB). Combined with `loudness_max` and `loudness_max_time`, these components can be used to describe the “attack” of the segment.
loudness_max	float	The peak loudness of the segment in decibels (dB). Combined with `loudness_start` and `loudness_max_time`, these components can be used to describe the “attack” of the segment.

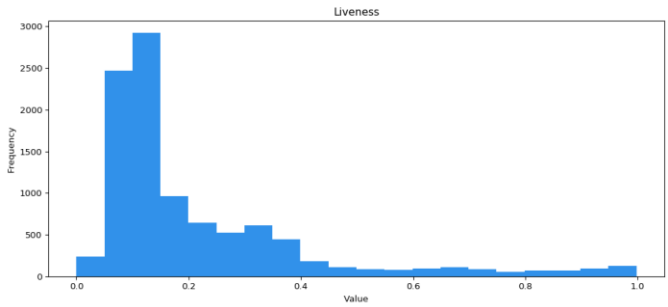
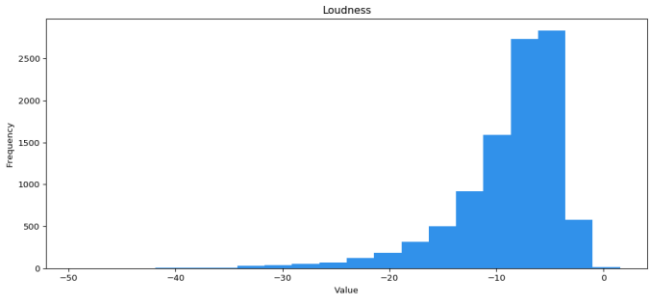
loudness_max_time	float	The segment-relative offset of the segment peak loudness in seconds. Combined with `loudness_start` and `loudness_max`, these components can be used to describe the “attack” of the segment.
loudness_end	float	The offset loudness of the segment in decibels (dB). This value should be equivalent to the `loudness_start` of the following segment.
pitches	array of floats	A “chroma” vector representing the pitch content of the segment, corresponding to the 12 pitch classes C, C#, D to B, with values ranging from 0 to 1 that describe the relative dominance of every pitch in the chromatic scale.
timbre	array of floats	*Timbre* is the quality of a musical note or sound that distinguishes different types of musical instruments, or voices. Timbre vectors are best used in comparison with each other.

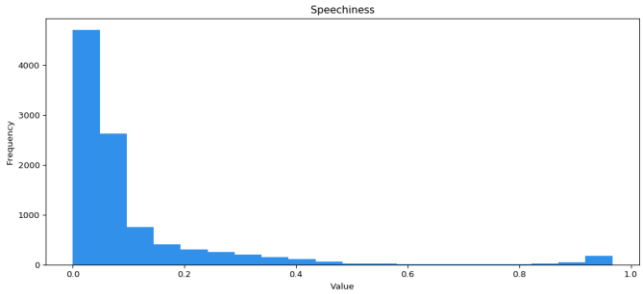
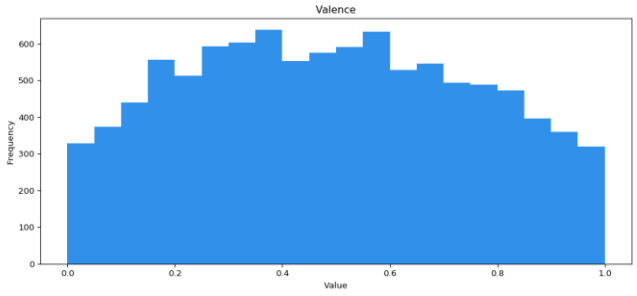
Audio features object

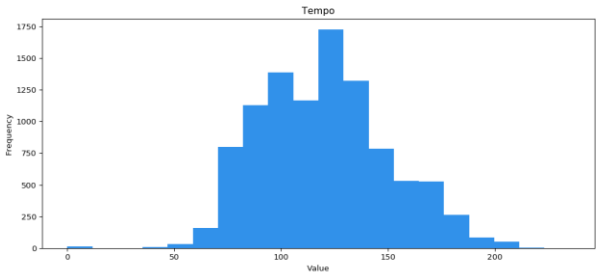
KEY	VALUE TYPE	VALUE DESCRIPTION
duration_ms	int	The duration of the track in milliseconds.
key	int	The estimated overall key of the track. Integers map to pitches using standard Pitch Class notation . E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1.
mode	int	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
time_signature	int	An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

acousticness	float	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic. The distribution of values for this feature look like this: 
danceability	float	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable. The distribution of values for this feature look like this: 

energy	float	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy. The distribution of values for this feature look like this: 
instrumentalness	float	Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”. The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0. The distribution of values for this feature look like this: 

liveness	float	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live. The distribution of values for this feature look like this: 
loudness	float	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typical range between -60 and 0 db. The distribution of values for this feature look like this: 

speechiness	float	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks. The distribution of values for this feature look like this: 
valence	float	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry). The distribution of values for this feature look like this: 

tempo	float	The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration. The distribution of values for this feature look like this: 
id	string	The Spotify ID for the track.
uri	string	The Spotify URI for the track.
track_href	string	A link to the Web API endpoint providing full details of the track.
analysis_url	string	An HTTP URL to access the full audio analysis of this track. An access token is required to access this data.
type	string	The object type: “audio_features”

Appendix B. State controller enumerators

These enumerators imply different meanings depending on the context they are used. This is a brief summary of their uses.

```
// DEFINITION OF ENUMS IN SOURCE CODE
const GameStateEnums = Object.freeze({
    'IDLE': 'IDLE',
    'GENERATE': 'GENERATE',
    'PLAY': 'PLAY',
    'PAUSE': 'PAUSE',
    'ERROR': 'ERROR',
    'SCORE_CHANGED': 'SCORE_CHANGED'
});
```

IDLE

(Emitted exclusively from the UI → Game)

Requests the game to move to an idle state. If a reason is specified in the body of this request, it means that this IDLE state was explicitly triggered by the front end logic.

GENERATE

(Emitted exclusively from the UI → Game)

Requests game to begin level generation. The body of this requests contains an object containing two JSON objects: analysis and features. This is sent to the game directly from the Spotify API, through the Spotify Service.

PLAY

(Emitted both ways)

When sent from the Game module, this notification intends to trigger the playback of the selected track. It also signals that the game has finished generating a mountain and will begin playing. When sent from the UI, it requests the game to begin playing, presumably coming from a PAUSE state.

PAUSE

(Emitted exclusively from the UI → Game)

Requests game to pause, normally due to a Spotify Web Player state change.

ERROR

(Emitted exclusively from the Game → UI)

The purpose of this state enum is to emit errors from the game module to be displayed in the UI. No implementation to any part of this feature (Error Pipeline) has been implemented as of publication.

SCORE_CHANGED

(Emitted exclusively from the Game → UI)

When emitted from the Game, this notification contains two values in the notification body: *score* and *multiplier*. These values represent the most up to date values for these parameters in the game, and should be used to replace the old values displayed in the GUI.

Appendix C. Replay utility source code

Here is attached a copy of the Replay Utility internal module of the input manager at the time of publication. For the most recent version, follow this link. For this specific version, follow this link.

```
/**
 * Wraps code for chaining timeout calls to emit replays.
 */
class ReplayUtil {

    /**
     * Constructor for replay utility. Must be initialized with an emitter function.
     * This function will be called on every action at the time the action should be performed.
     *
     * This is usually passed as some Rx.Subject.next(), but I decided to keep it flexible, because this
     * code seems useful.
     *
     * @param {Function} emitterFunction called with every action event at the time the event is
replayed. Only parameter is the `ActionEvent` object itself.
     * @param {Function} onConclusion called when there are no more inputs
     */
    constructor(emitterFunction, onConclusion) {

        // type check
        if (typeof emitterFunction !== 'function') throw new Error(`Emitter function parameter is of
wrong type. Type: ${typeof emitterFunction}.`);
        if (typeof onConclusion !== 'function') throw new Error(`End of input stream handler
(onConclusion) parameter is of wrong type. Type: ${typeof onConclusion}.`);

        this.actions = [];
        this.emit = emitterFunction;
        this.onEndOfInputs = onConclusion;
        this.currentTimeout = null;

        // bind funcs
        this.emitEventAt = this.emitEventAt.bind(this);
        this.loadActions = this.loadActions.bind(this);
        this.terminateReplay = this.terminateReplay.bind(this);
        this.beginReplay = this.beginReplay.bind(this);

    }

    /**
     * Getter to determine if replay util is currently replaying.
     *
     * @returns {boolean} true if timeout is ongoing, false if timeout has been cleared.
     */
    get isReplaying() {
        return !!this.currentTimeout;
    }

    /**
```

```
     * Emits event at index, and if there is a next move, set an appropriate
     * timeout for said move.
     *
     * @param {Number} index index of the move to be emitted.
     */
    emitEventAt(index) {

        // will throw error if indexed out of bounds
        const currentMove = this.actions[index];

        // emit move
        this.emit(currentMove);

        // check for next move
        let nextMove;

        // if index is appropriate
        if (index + 1 < this.actions.length) {

            // set next move
            nextMove = this.actions[index + 1];

        }

        // if we have a next move, chain it
        if (nextMove) {

            this.currentTimeout = setTimeout(
                this.emitEventAt,
                nextMove.timestamp - currentMove.timestamp,
                index + 1
            );

        } else {

            // if not, call the end function
            this.onEndOfInputs();
            this.terminateReplay();

        }

    }

    /**
     * Loads actions into the module.
     * @param {Array<ActionEvent>} actionArray events in list should have a timestamp field
     */
    loadActions(actionArray) {

        // check for presence of timestamp field
        actionArray.forEach((action, index) => {
            if (!action.timestamp) throw new Error(`An action at index ${index} is missing the timestamp
field. Aborting.`);
        });

        // set local variable, convert it to a proper ActionEvent
        this.actions = actionArray.map(input => new ActionEvent({
            ...input
        }));
```

```
    }

    /**
     * Cancels timeout chain. This type of termination (forced) does not call onEndOfInput();
     */
    terminateReplay() {

        clearTimeout(this.currentTimeout);

        this.currentTimeout = null;

    }

    /**
     * Begins timeout chain with the loaded actions. If replay is not forcibly terminated,
     * the module will call "onEndOfInput" at the end of the stream.
     */
    beginReplay() {

        // possibility for pausing? restarting at a given arbitrary input would require accounting for
when the pause happened.
        const beginIndex = 0;

        if (!this.actions || this.actions.length < 1) {

            this.onEndOfInputs();

            return;

        }

        const firstMove = this.actions[beginIndex];

        // begin timeout chain
        setTimeout(
            this.emitEventAt,
            firstMove.timestamp,
            0
        );

    }

}

module.exports = ReplayUtil;
```

Appendix D. AlphaFest survey questions

1. How many songs were you able to play?
2. How would you rate your overall enjoyment with the game? (1-6 scale)
3. How well did the coins line up with the song on the Macbook? (1-6 scale)
4. How well did the coins line up with the song on the Dell? (1-6 scale)
5. Did you like the feeling of collecting the coins better on the:
    a. Macbook
    b. Dell
    c. No preference
6. Did you feel that your speed matched the song?
    a. Yes, it felt great
    b. Only sometimes
    c. Not at all
7. How did the jump feel? Choose all that apply.
    a. Too floaty
    b. Too high
    c. Too low
    d. Too weak
    e. Felt good
    f. Other (Explain why)
8. Do you have Spotify Premium? (Yes/No)
9. What did you enjoy about our game? Did the music and game elements feel in sync? (Long answer)
10. Are you a musician? (Yes/No)
11. Please share if you came across any bugs. (Long answer)
12. What would you like to see added to this game? (Long answer)

Appendix E. C-Term survey questions

1. What elements did you particularly enjoy? Choose all that apply.
    a. The speed
    b. The tricks
    c. The moving backgrounds
    d. The jumping
    e. The artwork
    f. The player character
    g. The animations
    h. The procedurally generated mountain
2. What elements did you dislike/have problems with? Choose all that apply.
    a. The speed
    b. The tricks
    c. The moving backgrounds
    d. The jumping
    e. The artwork
    f. The player character
    g. The animations
    h. The procedurally generated mountain
3. Did you feel that this experience helped enhance the music you were listening to? (Yes/No)
4. To what degree did you feel the elements in the level were in sync with the music you were listening to? (1-6 scale)
5. To what extent did the moving background help create the feeling of going down a mountain? (1-6 scale)
6. How much did the moving background help give you a sense of speed as you went down the mountain? (1-6 scale)
7. Did you feel like you were moving at an appropriate speed for the type of song you chose?
    a. Too slow

    b. Just right
    c. Too fast
8. To what extent did the background elements positively complement your visual experience? (1-6 scale)
9. Did being able to choose your own song help enhance your experience over having to select a song from a given playlist? (Yes/No)
10. Select all that describe your experience choosing a song.
    a. Easy to use
    b. Overwhelming
    c. Couldn’t think of a song
    d. Catered to me
    e. My choice wasn’t there
    f. Other
11. Would you like to see song recommendations based on what you listened to previously? (Yes/No)
12. If there was one thing you could improve about this game what would it be? (Long answer)
13. Do you have a Spotify Premium account? (Yes/No)
14. Do you have any other comments about our game? (Long answer)

Appendix F. IRB Purpose of Study & Protocol

Purpose of study

To obtain playtest feedback in order to locate/address operational bugs, and to identify opportunities for design improvement.

Study protocol

Participants are provided a computer on which to play the sample game. Investigator observes participants during play, answering questions and providing guidance as needed. Afterward, participants are asked to fill out a short survey to characterize their subjective experience.

Opening briefing for testers

“Hello, and thank you for volunteering to test our game. Before we begin, could you please read and sign this Informed Consent form? [Tester signs IC form.] Thank you. During your test session, the game will be recording a variety of metrics about your play activity. When your session is complete, we will ask you to complete a brief survey about your play experience. At no point during your play session, or in the survey after, will any sort of personal and/or identifying information about you be recorded. Please begin playing when you feel ready.”

Appendix G. IRB Informed Consent Form

Informed Consent Agreement for Participation in a WPI Research Study

Investigator: Brian Moriarty, IMGD Professor of Practice

Contact Information: bmoriarty@wpi.edu, 508 831-5638

Title of Research Study: Tune Mountain

Sponsor: WPI

Introduction: You are being asked to participate in a research study. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation. This form presents information about the study so that you may make a fully informed decision regarding your participation.

Purpose of the study: The purpose of this study is to obtain feedback on the MQP project in order to facilitate design improvements and find/address operational bugs.

Procedures to be followed: You will be asked to play a brief game lasting less than ten minutes. Instrumentation in the game software will anonymously record your activity during play. After completing the game, you will be asked to complete a brief, anonymous survey describing your subjective experience.

Risks to study participants: There are no foreseeable risks associated with this research study.

Benefits to research participants and others: You will have an opportunity to enjoy and comment on a new game under active development. Your feedback will help improve the game experience for future players.

Record keeping and confidentiality: Records of your participation in this study will be held confidential so far as permitted by law. However, the study investigators and, under certain circumstances, the Worcester Polytechnic Institute Institutional Review Board (WPI IRB) will be able to inspect and have access to confidential data that identify you by name. Any publication or presentation of the data will not identify you.

Compensation or treatment in the event of injury: There is no foreseeable risk of injury associated with this research study. Nevertheless, you do not give up any of your legal rights by signing this statement.

For more information about this research or about the rights of research participants, or in case of research-related injury, contact the Investigator listed

at the top of this form. You may also contact the IRB Chair (Professor Kent Rissmiller, Tel. 508-831-5019, Email: kjr@wpi.edu) and the University Compliance Officer (Jon Bartelson, Tel. 508-831-5725, Email: jonb@wpi.edu).

Your participation in this research is voluntary. Your refusal to participate will not result in any penalty to you or any loss of benefits to which you may otherwise be entitled. You may decide to stop participating in the research at any time without penalty or loss of other benefits. The project investigators retain the right to cancel or postpone the experimental procedures at any time they see fit.

By signing below, you acknowledge that you have been informed about and consent to be a participant in the study described above. Make sure that your questions are answered to your satisfaction before signing. You are entitled to retain a copy of this consent agreement.

_____ Date: _____

Study Participant Signature

_____

Study Participant Name (Please print)

_____ Date: _____

Signature of Person who explained this study