HYPER GROOVE: AN INTERACTIVE MUSIC GAME

Interactive Media and Game Development

A Major Qualifying Project
Submitted to the faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted by Cian Rice

Advised by
Brian Moriarty
IMGD Professor of Practice

Abstract

Hyper Groove is an original music and rhythm game, written in C++, in which players must juggle two instrument tracks while trying to score as many points as possible. This report explains the design process that lead to the creation of *Hyper Groove*, details of its engineering and production, and final thoughts on what went well and what could be improved upon in future works.

Acknowledgements

To begin, I would like to thank Professor Moriarty for providing me with the guidance to complete this project. I would also like to take the time to thank my father for providing me with moral support when I felt frustrated while working. Also I would like to thank Alex Harris, a friend of my father, for helping me on technical side by setting me down the right road when it came to solving some of the major problems encountered.

The art in *Hyper Groove* would not be as good as it is if not for my friend Dan Cherkassky, who has been a great friend and was of immense help during the project. Professor Zizza's audio knowledge was also invaluable towards the end of the project. Finally, I would like to thank Sasha Abdurazak for his excellent voiceover work for the tutorial and song count-ins. They really add to the experience of the game.

Table of Contents

List of Figures

1. Introduction

Hyper Groove is a fast-paced “beat-matching” music and rhythm game in which players must juggle two instrument tracks while trying to score as many points as possible. The game features bizarre imagery and a “chaotic” art style that helps it stand out among the countless music and rhythm games available in the marketplace.

Particular goals drove the development of the game. First, it was important to create a game that could act as a centerpiece to a professional portfolio – something that would really stand out as a high-quality piece of work. The second goal was to develop a full game in C++, a language commonly used in the game industry, from the ground up in order to get a better understanding of commonly used practices in the game industry. The final goal was to develop a music game that would offer a fair amount of replay and make players want to strive to become better at it and showcase their mastery of the game.

With these three goals in mind, development began in earnest in August 2013. Development ended in May 2014. The game’s design evolved constantly during the development, but the core design was cemented at the end of A-Term.

2. Design Process

2.1 Original Ideas

There were three ideas that preceded the concept for *Hyper Groove*. These ideas were wildly different from each other, but all involved music heavily. The three ideas each had their own working title – *Beside You in Time*, *Elite Beat Band*, and *SchizoRhythm*.

2.1.1 Beside You in Time

The first idea was titled *Beside You in Time* (BYiT) after the song and live concert movie by industrial rock band Nine Inch Nails. The concept was simple, but core ideas in it permeated every subsequent game concept. The game was a split-screen auto-runner, a kind of side scrolling game where players are always moving forward, and only have control of actions such as attack, jump, and duck. The twist with BYiT was that players are controlling two characters that are running towards each other, trying to meet. Both characters have to navigate a hazardous world by jumping and attacking in time with the music.

Musically speaking, the game would have a sort of post-industrial meets shoegazer style, again inspired by the song it was named after. Visually it would feature a pixel-art style inspired by the indie game *Fez*. While this project would have been feasible from a technical and design perspective, the art requirements were too demanding for a solo MQP.

2.1.2 Elite Beat Band

Realizing that BYiT wouldn’t be feasible, a game that could work with simpler art was needed. It was then that the idea to merge two “beat matching” music games came about. Beat matching refers to the concept of pressing buttons in time with an instrument in a song. The two games in question, *Elite Beat Agents* (EBA) and *Rock Band*, are two radically different interpretations of beat matching. In EBA, a game for the Nintendo DS handheld system, players are tasked with tapping circles on the screen as a shrinking ring hits each circle. *Rock Band* is a cooperative multiplayer game focused on replicating the experience of being in a band. The idea behind *Elite Beat Band* was to combine the style of *Elite Beat Agents* with the cooperative experience of *Rock Band.*

In this concept, each player would choose an instrument and then play a game similar to EBA, but tailored to a game controller instead of a touch device. There would be bonuses for hitting notes perfectly at the same time, and the goal would be to get the highest score possible as a band. However, while this design is interesting, it presents a lot of mechanical issues. First and foremost, the game would be much easier on a controller than a touch device – part of the experience of EBA is the reflex of tapping the screen at the right place in time. Additionally, there was a big problem of how to present the game – would it be split screen with each player having to focus on a small section of the screen? That would certainly be the easiest method, but it also kind of detracted from the experience of being a band. These uncertainties lead to the idea being scrapped.

2.1.3 SchizoRhythm

Finally, there was *SchizoRhythm*, a much simpler idea tailored to one player. The idea was that a player would use a keyboard or Xbox controller and try to match beats on two instrument tracks simultaneously with two buttons or keys for each instrument. Of the scrapped ideas, this was the only one that was prototyped out.

The concept sounded difficult, but playable on paper. However, a quick prototype revealed that the game was nearly impossible to play. For this reason the idea was quickly scrapped, but from it came *Hyper Groove*.

2.2 The Final Idea

The idea to play multiple instruments permeated through each design idea – be it playing them simultaneously or with friends. *Hyper Groove* takes this idea and approaches it from a familiar perspective to fans of Harmonix titles such as *Amplitude*, *Frequency*, *Rock Band Unplugged*, or *Rock Band Blitz*. The concept is rather simple – players have multiple instrument tracks with notes scrolling down, but players can only play one track at a time. In *Hyper Groove*, players have two instrument tracks that they have to switch between. The game is presented in a format similar to Japanese or Korean music games such as *Beatmania*, *Pop'n Music*, or *DJ MAX Portable FEVER*. This essentially means that instead of a 3D track with notes coming towards the player, there is a 2D track with notes moving towards the bottom of the screen.

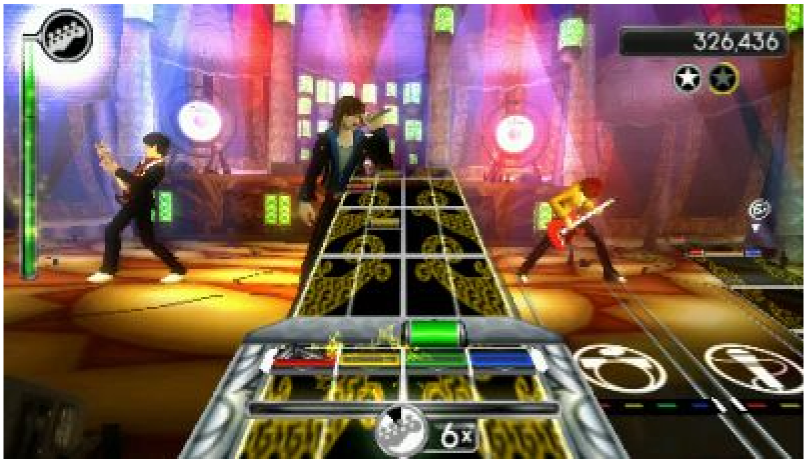


Figure 1: Screenshot of *Rock Band Unplugged* for Playstation Portable.
http://upload.wikimedia.org/wikipedia/en/d/db/RBUnplugged_screen.jpg

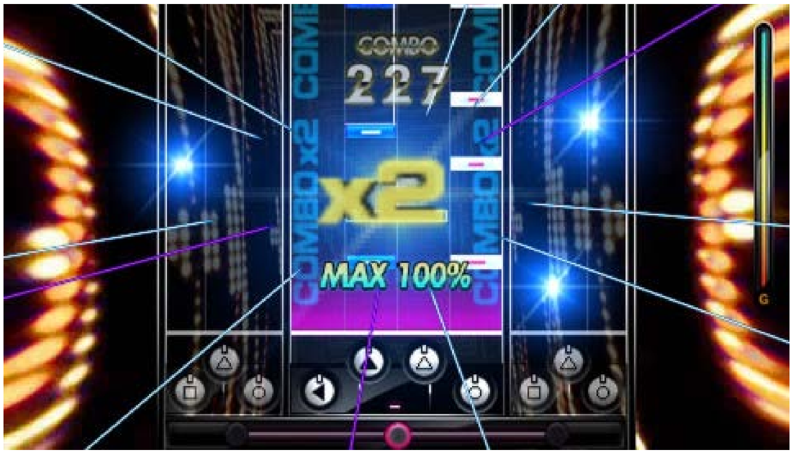


Figure 2: Screenshot of *DJ MAX Portable 3* for Playstation Portable.
http://2.bp.blogspot.com/-ySC0Iyuf3QQ/UWIENAEOaSI/AAAAAAAAA38/OPcNGqWy0Bs/s1600/
dj-max-portable-3-playstation-portable-psp-008.jpg

3. Gameplay

As previously mentioned, *Hyper Groove* features two instrument tracks. Each track has three note "lanes," which represent different notes being played. Players have to switch between the two tracks as the song progresses in order to not only score as many points as possible, but also to simply complete the song without failing out. Each track has its own life meter, and as the player plays on the currently active track a timer begins to countdown. Once the timer hits 0, the life meter for the track the player is not currently playing begins to deplete with each missed note. Once the life meter is fully depleted the player loses a life, represented on the screen by a shield icon. If the player has no shield icons left, they will fail the song upon a meter depleting.

The game features another hallmark of Japanese and Korean music games – an accuracy indicator. Not only do players need to hit the notes as they reach the bottom of the screen, they also need to try and hit the note as accurately as possible. The more accurately a player hits the note, the more points they are rewarded. The points rewarded are determined by the base amount of points multiplied by the accuracy indicator's multiplier. They are as follows:

- “GREAT” (x2 points)
- “NICE” (x1.5 points)
- “GOOD” (base value)
- “POOR” (x.5 points)
- “MISS” (0 points)

In addition, each level of accuracy adds and subtracts a different amount of life.

- “GREAT” (+3)
- “NICE” (+ 2)
- “GOOD” (+ 1)
- “POOR” (- 1)
- “MISS” (- 2)

3.2 Goal & Win/Lose Conditions

The player’s goal in the game is rather simple – score as high a score as possible without failing. As an incentive, the top five scores for a given song are listed on the song select screen under “Top Scores.” The win condition plays off of this – simply complete the song without failing. The lose condition is met when players lose all lives and have depleted their life meter for a fourth time due to missing notes.

3. 3 Affordances

There are a few affordances open to the player. First and foremost players can press the button corresponding to the proper note at the proper time to score points, and gain some life back if the life meter is not full. Alternatively the player can fail to do this and lose some life. The final affordance is the ability to switch between the two tracks, thus enabling the player to change which part of the song they are playing along with.

3.3 Gameplay Modifiers

There are two special modes that change the core of the game. First is the “No Fail” mode, in which players do not have to worry about failing out of a song. Because of this they can focus solely on one track without having to switch, they will only have to deal with the minor annoyance of a red flashing effect when one of the life meters is low.

The second special mode is called “Doom Mode.” In Doom Mode, players are given three skull icons for each life. Missing a note removes a skull icon. Once all three skull icons are gone missing more notes begins to deplete the player’s score. In this mode it is possible for players to hit negative scores if they lack the proper skills. Doom Mode began as a joke – a thought about what would happen if a game developer tried to make “the *Dark Souls* of music games.” Infamous for its incredibly punishing gameplay, *Dark Souls* is a popular game amongst many hardcore gamers. While intended as a joke, the implementation of the mode was fairly simple, so it was included in the final game.

3.4 Learning to Play

Tutorials are important for any type of game, especially with a genre where players of vastly different skill levels play. With this in mind it was important to have some form of a tutorial within *Hyper Groove*. Due to time constraints, the tutorial was put together at the last minute, and lacks interactivity. The tutorial features images set to the backdrop of an audio walkthrough, with voice work provided by Sasha Abdurazak.

4. Development Process

4.1 Tools

Hyper Groove was built from the ground up in Visual Studio 2012, which at the beginning of development was the latest version of an industry standard IDE. As one of the goals of the project, *Hyper Groove* was developed in Visual C++ 11, with several libraries to aid development of the core game engine. For rendering images to the screen and managing input, I used Simple Direct Media Library. For parsing MIDI files a subset of the J.U.C.E (Jule's Utility Class Extensions) audio plug-in library was used. For loading and playing audio, and for some simple digital signal processing, FMOD Studio was utilized, specifically its Programmer's API. Lua 5.1 was the scripting language of choice, and to let Lua access information from classes, a wrapper called LuaBridge was employed. Some header files were used to allow the engine to read and write XML files and tween objects. The XML was done via tinyXML, while the tweening was courtesy of dbTweener.

Many MQPs are built in ready-made game engines, or use custom game engines built in languages such as Java or C#. So why build a custom C++ engine? The reasoning for this was that it would make for a stronger portfolio piece. Not only that, but developing the game and its accompanying engine in C++ was a valuable learning experience.

For art, Photoshop CS6 was used. Photoshop is currently an industry standard tool for digital 2D art, and was easily available. It was used to create the game's user interface and modify images provided by the game's artist as necessary. Any audio mixing and editing, along with the authoring of MIDI files was done in Reaper.fm, a versatile multi-track audio editing tool.

4.2 Building the Core

4.2.1 Finding the Right Development Libraries

In order to build the engine that would power *Hyper Groove*, it was important to find several libraries that would be a good fit. There were a few requirements that the "core" library, or rather the library that would do most of the heavy lifting, would need to fulfill:

1. Input Handling
2. Graphics Handling
3. Support for C++

There was also a fourth optional feature, audio handling. This was optional due to the fact that FMOD was being considered as potential audio engine. With these requirements in mind, a search began to find the right core library. Professor Moriarty first suggested Simple Direct Media Library, otherwise known as SDL. Several commercial and indie games have used SDL – the website for the library lists titles such as *EDGE*, *Cave Story+*, *FTL: Faster than Light*, and many more. There were also many resources on the internet for support. However, there were a few cons. The latest version of SDL, version 2.0, was poorly documented. Some of the help files still listed descriptions of functions as works in progress. Additionally, upon research of further libraries, it became apparent that SDL was a lower-level library than the rest. Essentially this meant a lot more work was required for basic functionality in SDL. Finally, SDL only natively supports bitmaps and required an add-on library to load any other kind of image. This was a minor issue however, as the other library, known as SDL-Image, was well documented and very similar syntactically.

Another popular C++ library was Simple and Fast Multimedia Library, otherwise known as SFML. SFML had a strong community backing it. Additionally it also had built-in audio support. There weren't really many drawbacks to SFML as it was well-documented and had a thriving community. Finally there was AngelCode, a relatively new 2D game engine. That was the first plus for AngelCode – it was a game engine. In addition it had the highest level of abstraction of all the libraries looked at due to this fact. It also had built-in Lua support. However there were several cons. First and foremost it was very new, meaning not a lot of people could really vet it. Second, the community for it appeared to be rather small. Finally, it was a game-engine, meaning a lot of the work was already done.

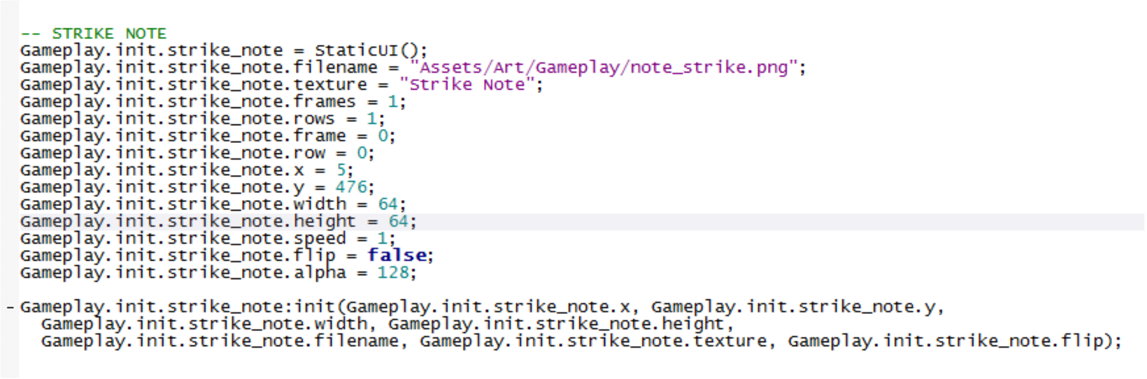
In the end, the decision was made to use SDL. While SFML was a very strong contender, SDL seemed like it would be a better tool for building out a game engine.

4.2.2 Building the Engine

To begin building the engine, it was necessary to get up to speed on how to use SDL. *SDL Game Development*, a book by Shaun Mitchell was used to quickly learn about how to render images to the screen, how to get input, etc. in SDL. In addition, the engine would need to handle multiple states such as gameplay, game over, main menu, and so on and so forth. To that end a state machine tutorial mentioned by Professor Charles Rich in the course Technical Game Development II was used. The tutorial, by Mat Buckland, was intended for state-driven artificial intelligence, but an adapted version made it perfect for game state management.

However, scripting and audio were still needed to fully flesh out the engine. First there was scripting. Lua was the obvious choice for this, as it is frequently used in games. However, it was quickly discovered that Lua does not deal well with C++ features such as classes. Essentially, trying to access member variables or functions was impossible. Due to time constraints it seemed unrealistic, and unnecessary, to create a wrapper for this since there were so many wrappers available in the first place. Unfortunately, while there were many wrappers available, only one of them worked with Visual C++ 2012 at the time. This wrapper, LuaBridge, provided all the needed functionality.

Lua scripts are used in the engine for initializing game objects and other important values in each game state's initialization function. While Lua scripts can be used in real-time, it is not fast enough for a game where everything is so time sensitive such as this, otherwise Lua could be used for other things as well.



```
-- STRIKE NOTE
Gameplay.init.strike_note = StaticUI();
Gameplay.init.strike_note.filename = "Assets/Art/Gameplay/note_strike.png";
Gameplay.init.strike_note.texture = "Strike Note";
Gameplay.init.strike_note.frames = 1;
Gameplay.init.strike_note.rows = 1;
Gameplay.init.strike_note.frame = 0;
Gameplay.init.strike_note.row = 0;
Gameplay.init.strike_note.x = 5;
Gameplay.init.strike_note.y = 476;
Gameplay.init.strike_note.width = 64;
Gameplay.init.strike_note.height = 64;
Gameplay.init.strike_note.speed = 1;
Gameplay.init.strike_note.flip = false;
Gameplay.init.strike_note.alpha = 128;

Gameplay.init.strike_note:init(Gameplay.init.strike_note.x, Gameplay.init.strike_note.y,
  Gameplay.init.strike_note.width, Gameplay.init.strike_note.height,
  Gameplay.init.strike_note.filename, Gameplay.init.strike_note.texture, Gameplay.init.strike_note.flip);
```

Figure 3: An example snippet from a Lua script.

For audio, there was only one real clear choice – FMOD Studio, the latest version of the FMOD Audio library. FMOD was chosen because it was not only free for anyone building a game with a gross income less than one hundred thousand dollars, but also due to prior exposure. FMOD is used in the engine for loading and playing music and sound effects as well as performing some digital signal processing.

However, FMOD Studio didn’t appear to have functionality that would allow for MIDI files to be parsed into a format suitable for the gameplay. To that end, there were two options – write a custom MIDI event parser, or find a library that would do it. Once again, time was of the essence so it was decided to use a library that already existed. After searching, Jule’s Utility Class Extensions, otherwise known as J.U.C.E., was chosen. J.U.C.E has many features, but allows users to compile only what they needed into a library. For that reason, the MIDI functionality is the only part of J.U.C.E used in the *Hyper Groove* engine.

There are three main components to the engine – the managers, game objects, and game states. The managers are designed following the singleton design pattern. This means that there can only be one instance of the manager at a time. This was extremely important because if there were multiple instantiations of a manager at any one time the whole engine would cease to function as intended. There are a total of six managers.

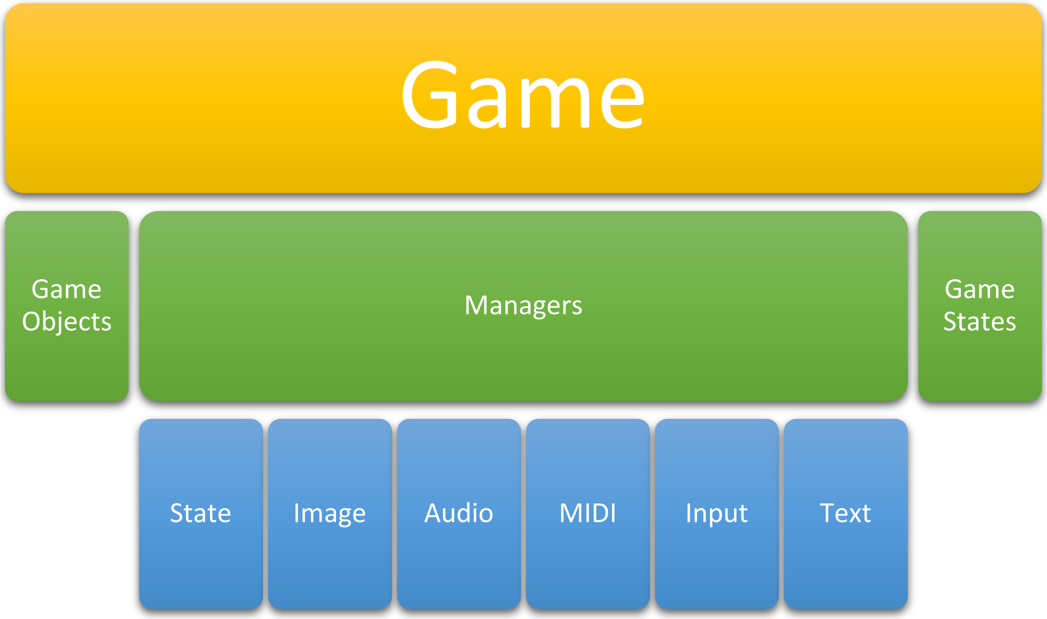


Figure 4: The general structure of the game engine.

4.2.3 The Managers

The first manager is the game state manager. Its duty is to provide easy access to a state machine that is responsible for updating and rendering the current game state. The state machine also handles state transitions.

The image manager handles the loading and rendering of images. Rendering an image is done via the draw function, of which there are several overloads. Each overload has some different functionality – one can draw a frame of a sprite sheet, while another can change the alpha value of the image, the final overload can also tint the image based upon an SDL_Color value. In all version of the function the following procedure is followed:

- Create a source rectangle.
- Create a destination rectangle.
- Copy the image of size *source rectangle* to the destination rectangle.
- Using an SDL_Renderer, which represents a rendering context, render the image to the destination rectangle.

Before an image can be drawn however, it must be loaded. Loading involves providing a label and filename for the image the user wishes to load. The image is loaded and stored in a temporary SDL_Surface. As long as this surface is not NULL, an SDL_Texture is created from the surface. The surface is than freed from memory and the texture is added to a std::map where the key is a string called label and the value is the texture that has been loaded.

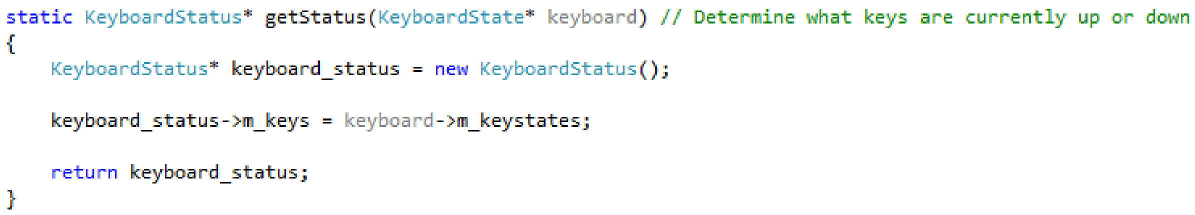
Similar to the image manager is the text manager. Originally it used the SDL add-on library SDL_TTF for rendering text. However, it proved to be too resource intensive – as the text would need to update in real-time, it would eventually disappear or cause the game to crash. So an alternative was needed. A user on an SDL mailing list posted a solution back in 2003 for an older version of SDL. This snippet of code was modified for use with SDL_Image, and the newer SDL library the engine was using and allowed for a simple bitmap font rendering manager.

Next, there is the audio manager. As previously mentioned, the audio manager uses FMOD Studio to handle the loading of music and sound effects along with a very basic digital signal processing function. A std::map is used for storing audio similar to the image manager. There are two separate functions (and std::maps) for sound effects and music. The FMOD sound system creates either a sound or a stream depending on whether a sound or song is being loaded. If the resulting FMOD_RESULT is not "okay", then a null value is returned. If the result *is* okay, the sound is returned.

Then there is actually playing the music and sounds. First, FMOD checks if the music or sound's channel is playing. If it is not then the audio is played. Additionally, there is a function for performing Fast Fourier Transform for specific user interface elements – updating them based upon the audio spectrum data.

The audio manager doesn't deal with MIDI at all. Each song's note chart and special effect list is authored in a MIDI file with different MIDI events. For the note chart, each track is looking for MIDI note-on/off events while the special effect list is looking for a MIDI text event that contains the name of the special effect to be used at the time in the file. The MIDI manager, when parsing the note track, opens up a MIDI file via J.U.C.E then loops through to find all the note-on and corresponding note-off events. It stores these in a std::vector, which is then used in another loop to create the game-ready note data based upon whichever note (ranging from C4 to D4 on a piano roll) is on. For special effects, the second loop creates a temporary "FX" object that is eventually used to create the actual FX object.

The final manager is the input manager. This manager includes an update loop that acts as a polling mechanism, implemented as an SDL Event loop. The input manager operates by checking for key presses on the keyboard. It also has some basic gamepad functionality that allow it to detect button presses. The method for accessing these was inspired by Microsoft's XNA. There is a keyboard state and a gamepad state - through these the various key/button press states can be checked.



```
static KeyboardStatus* getStatus(KeyboardState* keyboard) // Determine what keys are currently up or down
{
    KeyboardStatus* keyboard_status = new KeyboardStatus();

    keyboard_status->m_keys = keyboard->m_keystates;

    return keyboard_status;
}
```

Figure 5: A small piece of code that returns the current status of the keyboard.

Gameplay is made possible through the interaction of various game objects. All game objects inherit from a base game object class. All game objects have their own update and rendering loop, along with their own initialization function. There are many different types of game objects, including the notes that come down the screen and must be hit by the player, the UI, the special effects, and objects that can be animated. Finally, there are the different states of the game. Each state has its own render, update, and initialization function. All the different states inherit from a base state class.

4.2.4 Gameplay Programming and its Challenges

There were two major issues encountered while programming the gameplay. The first issue was synchronizing the note objects to the audio. The problem was first noticed early on – notes would come nearly a second after the corresponding note in the song was actually played. There were two main reasons this was happening. First, and most obvious, was that there was no precise way at first to detect if the time had been reached for a note to appear. Second, getting the time into the song using FMOD was too expensive a call – it could take more time in milliseconds than there was between notes. These two problems made it obvious that a different approach was necessary. To solve this, it was necessary to implement a callback timer that was kicked off right as the song started, and upon reaching a pre-defined amount of time prior to the note needing to be hit it would add the note to a std::vector of active notes. The timer would then kick off again until the next note needed to be displayed. This would continue until all notes were pushed off an inactive note vector to the active note vector. Additionally, SDL's built-in function SDL_GetTicks() was used to determine how far into a song the player was as it was a significantly cheaper function than the option provided by FMOD.

The second major issue was creating the song selection state. At first Lua scripts were used to get information about playable songs. However, this combined with file I/O caused a large amount of instability. Sadly, it never became clear what exactly was at the root of this problem, other than the fact that something was corrupting random values in each song on the list of playable songs. The solution to the problem was to introduce tinyXML into the engine and convert the song Lua scripts into XML files. This solution was also used for bringing in the special effect objects that the game uses.

5. Art Direction

5. 1 User Interface

The UI artwork started out as a bland, rather generic and clean interface. It was quickly decided this would be a detriment because it failed to make the game stand out. As a result, the user interface was rekindled to look more bizarre. The original intention was to model it after glitch art, specifically the style of data-bending done by artists such as Rob Sheridan. While there are certainly elements of this in the final interface, the actual art direction could be described as a nightmarish amalgamation of glitch art, indie game *Hotline Miami*'s art direction, and bright colors. The art was created by using some shape tools in Photoshop, applying some layer style to them, then exporting the resulting image as a RAW file. The RAW file would then be brought into an audio program – namely Audacity – and had audio effects applied to it. After that it would be brought into Photoshop again. This data-bent image would then be overlaid on the original and constantly distorted in Photoshop CS6 until the desired style was reached.

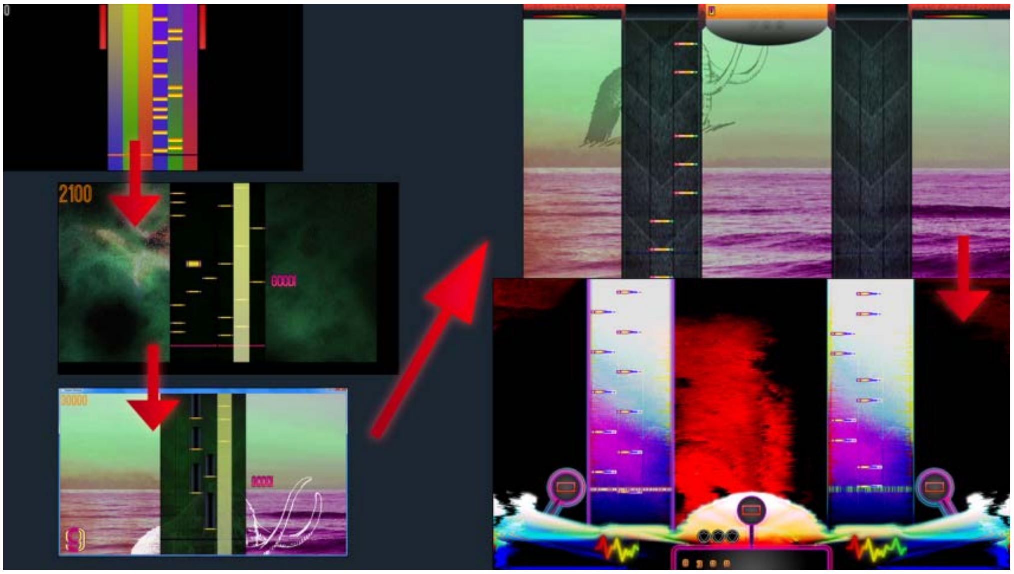


Figure 6: The evolution of the game's interface.

5.2 Backgrounds and Special Effects

The majority of the special effects and backgrounds in the game were created by Daniel Cherkassky, a senior at Mount Ida College in Newton, Massachusetts. The backgrounds and effects he created were inspired by surrealist artist Tim Biksup's music video for the Mastodon song "Dry Bone Valley". The art is, once again, bizarre and lacks cohesion – but this helps it stand out among its peers.

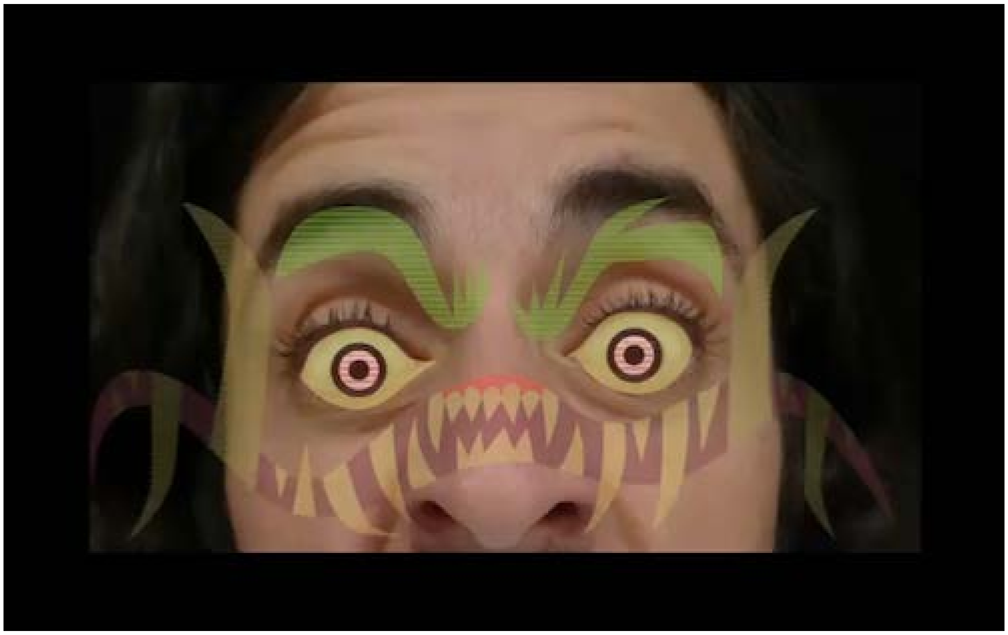


Figure 7: A shot from the "Dry Bone Valley" music video.
http://www.metalsucks.net/wp-content/uploads/2012/01/Screen-shot-2012-01-24-at-1.47.52-PM.png

5.3 Icons

Dan was also responsible for a set of icons that would be used in the game. These symbols were inspired by the instrument images featured in *Rock Band*. The icons in the game are:

1. A skull
2. A calendar
3. A microphone
4. A shield
5. A fast-forward symbol
6. A rewind symbol
7. A record symbol
8. A musical note

Unfortunately, the rewind and fast-forward icons were never implemented due to time constraints.

There were also keyboard pictures used in the game. These icons were made by Nicolae Berbece, with a license that permits them to be used for either non-commercial or commercial work. No credit is required either, however his name does appear in the final game's credits.

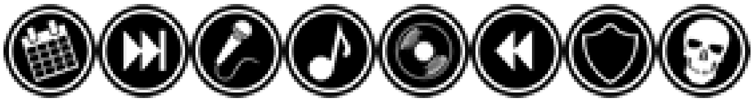


Figure 8: Icons made by Dan Cherkassky.

6. Music

Most music games have a feature where missing a note in the song silences the active instrument until another note is successfully hit. Early on in development, it was decided that this would not be the case in *Hyper Groove* due to a few constraints. First there was the constraint of time. It was unclear how laborious of a task implementing this feature would be, and it was decided it may not be 100% necessary. Second, and perhaps most important, the lack of access to master tracks was an issue. Third, it was unclear at the time what direction the game would head in. Would it be like *Elite Beat Agents,* where hitting a note accentuates the song, a sort of positive reinforcement, rather than muting a track, which is negative reinforcement?

Issue two was actually an uncertainty at first. Originally, a Berklee alumnus agreed to provide two or three original songs for the game. Eventually communication with the musician ceased. For this reason, production moved forward using various popular music tracks for testing purposes. Eventually, when the prospect of showing the game at Penny Arcade Expo (PAX) East 2014 arose, some form of music that could be shown at a public event was needed. OverClocked Remix, a popular video game remix site, grants people permission to use their songs as long as they are not profiting from their use. In addition, a Nine Inch Nails album, 2008's *The Slip,* features a Creative Commons Attribution – Noncommercial - ShareAlike license which enabled a track from it to be used. What this means is that as long as the track is attributed, used in a non-commercial way, and, if remixed or modified, shared under the same license as the original track then there are no issues. The tracks used for the PAX East demo were:

1. "Lights in the Sky" – Nine Inch Nails
2. "Welcome to the Human Race" – John Revoredo and Paul Weaps
3. "A'Kid-pella" – Squarelaw

The final game also features some original music. At Made in MA, a pre-PAX East event, *Hyper Groove* was shown to many people, including a few current Berklee students. One of these students offered to write an original track for the game. His name is Vinicius Pippa, and his track was inspired by the battle theme from

Final Fantasy XIII, “Blinded by Light.” It features a dueling guitar and violin track. Additionally, another Berklee student, Jonathan Reed, wrote an original track for the game inspired by Scottish synth-pop up and comers CHVRCHES.

7. Playtesting and Evaluation

7.1 Made in MA Party

Because of difficulties during development and the amount of time it took for the game to be fully up and running, the first time the general public got their hands on the title was at the Made in MA Pre-PAX East party. Many people stopped and played the game and opinion was generally positive. Most people found that the learning curve for the game was rather steep, doubly so since the tutorial wasn’t audible in the loud environment of the event.

During this showing of the game, two senior designers at Harmonix Music Systems, located in Cambridge, stopped and played the game. They were impressed with the game, especially upon learning that it was essentially a solo project. One of the designers suggested adding in more feedback when a player hits a note in the game, this way players would be more likely to feel like they were playing along with the music.

Figure 9: Players trying out the game at the Made in MA party. https://fbcdn-sphotos-h-a.akamaihd.net/hphotos-ak-prn1/t1.0-9/1014306_663490353720751_7430438720721852382_n.jpg

7.2 PAX East 2014

The three days that comprised PAX East 2014 saw many people play the game. There were two major points of critique. The first was the tutorial. It was hard to understand, it didn't explain things clearly, and it wasn't interactive in any way. These were valid criticisms. Given this feedback, changes to the tutorial were made, including new voice work by Sasha Abdurazak who was able to better articulate the script. In addition the script was revised to better focus on what the actual controls of the game were. Unfortunately, due to time constraints the tutorial could not be made fully interactive.

The second critique was that it was difficult to know when to switch tracks due to the placement of the timer on the screen. To that end, duplicate switch timers were added. These new switch timers were placed in such a way that they would be easier to see while matching beats on each track.

A lesser complaint was the control scheme. In general people thought that the split controls – with one hand pressing a key, and the other hand pressing the other two keys – was odd. They preferred a configuration in which all three buttons were next to each other. To remedy this solution, the final game has two control schemes that players may choose from.

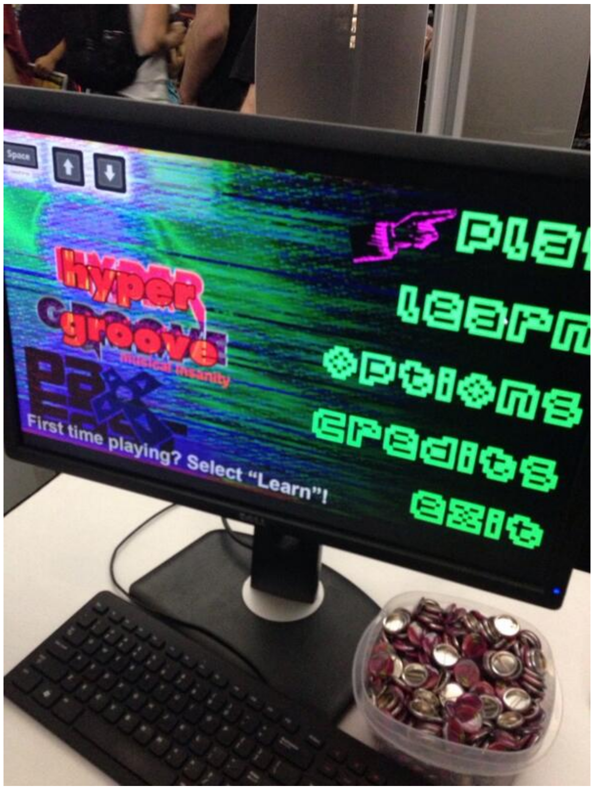


Figure 10: The game at the WPI Booth during PAX East.

8. Post-Mortem and Conclusions

Overall, developing *Hyper Groove* was an excellent learning experience. Much about the development process was learnt during the course of its creation. With that in mind, there are a few things that would stay the same while others would need to change.

8.1 Things to Keep

1. First, if this were to be done over again, it would still be done in C++. Doing this project in C++ proved to be the most important and valuable experience.
2. Furthermore, going it alone allowed for more control and the ability to follow through on a very specific vision.
3. Finally, the bizarre art direction was an excellent decision. It made the game stand out and players really found it quite charming.

8.2 Things to Change

1. While SDL was not a terrible choice for building the core of the engine with, it did consume more time than it probably should have. With this in mind, a library such as SFML would probably have been a better choice as it streamlines certain aspects of development that SDL does not.
2. The tutorial needed to be a much higher priority than it was. As a music game heavily inspired by other music games that weren't as well-known as behemoths like *Guitar Hero*, it should've been more apparent that a fully fleshed out tutorial would be important.
3. Play testing was equally important, and should have happened earlier in the development process. Very valid points made by play testers could not all be implemented due to lack of time.

8.3 Conclusion

In the end, *Hyper Groove* fulfilled all three goals of the MQP. More experience programming in C++ was gained. Additionally, *Hyper Groove* can easily stand as a centerpiece to a portfolio – both in terms of gameplay and technical features. Finally, the game is an extremely challenging music game that allows players to strive to earn the highest possible score they can in a song – to that end the goal of making a music game that allows for players to feel a certain level of mastery was successfully met.

Future MQPs can look to *Hyper Groove* as an example of how a game does not need to look like a AAA, big-budget blockbuster to be a successful capstone project – it just needs to be highly polished and focus in on a few mechanics that really work.

9. Credits

9.1 Images

Berbece, N. *Keyboard Icons.*

Cherkassky, D. *Black Mammoth.*

Cherkassky, D. *Black Narwhal.*

Cherkassky, D. *Calendar Icon.*

Cherkassky, D. *Fast-Forward Icon.*

Cherkassky, D. *Grizzly Bear Face.*

Cherkassky, D. *Koala Bear Face.*

Cherkassky, D. *Microphone Icon.*

Cherkassky, D. *Musical Note Icon.*

Cherkassky, D. *Panda Bear Face.*

Cherkassky, D. *Record Icon.*

Cherkassky, D. *Rewind Icon.*

Cherkassky, D. *Shield Icon.*

Cherkassky, D. *Skull.*

Cherkassky, D. *Skull Icon.*

Cherkassky, D. *White Mammoth.*

Cherkassky, D. *White Narwhal.*

Cherkassky, D. *White Octopus.*

Devlin, J. *Eyes.*

Kipnis, V. *Balloons.*

Kipnis, V. *Cat.*

Kipnis, V. *Eye.*

Kipnis, V. *Full Moon.*

Kipnis, V. *Lake.*

Kipnis, V. *Lens.*

Kipnis, V. *Mountains.*

Kipnis, V. *Rooftops.*

Kipnis, V. *Tree.*

Smith, H. *Girl By a Wall.*

9.2 Music and sound effects

funkymuskrat (Composer). (2007). chil.wav.

Hamazau, M. (Composer). (2011). Final Fantasy XIII 'Blue Skies' OC ReMix. [A. C. bLind, Performer]

Hulick, S. (Composer). (2011). Mass Effect 'Uncharted Depths' OC ReMix. [H. Bound, Performer]

Kondo, K. (Composer). (2011). Legend of Zelda: A Link to the Past 'Labyrinth of Dance Floors' OC ReMix. [some1namedjeff, Performer]

Korb, D. (Composer). (2013). Bastion 'A 'Kid-pella' OC ReMix. [S. Law, A. McLaren, D. Lane, D. Hayden, & R. Billington, Performers]

Mitsuda, Y. (Composer). (2009). Welcome to the Human Race (Stage of Death). [J. Revoredo, & P. Weaps, Performers]

Pippa, V. (Composer). (2014). Arroz. [V. Pippa, Performer] Boston, Massachusetts, United States of America.

Reed, J. (Composer). (2014). Pop Can. [J. Reed, Performer] Boston, Massachusetts, United States of America.

Reznor, T. (Composer). (2008). Lights in the Sky. [N. I. Nails, Performer]

Sakuraba, M. (Composer). (2012). Golden Sun 'Rebirth of Venus' OC ReMix. [S. Battle, & M. Rasmussen, Performers]

10. Works Cited

Bednar, J. (2003, January 29). *[SDL] Custom BMP Font Engine*. Retrieved from Lib SDL: http://lists.libsdl.org/pipermail/sdl-libsdl.org/2003-January/033369.html

Buckland, M. (2010). *Programming Game AI by Example.* Plano: Jones & Bartlett Learning.

Mitchell, S. (2013). *SDL Game Development.* Birmingham: Packt Publishing.

Appendix 1: Design Document

Summary

Elevator Pitch

Hyper Groove is best described as "*DJ Max Portable Fever* meets *Rock Band Unplugged*."

Detailed Pitch

Hyper Groove combines the fast paced beat matching gameplay of both *DJ Max Portable Fever* and *Rock Band Unplugged* to create a wholly unique music game. The game plays similarly to *Rock Band Blitz* or *Rock Band Unplugged* in that players switch between tracks while playing the song with each track representing a different part of the song (vocals, drums, etc.) but is presented on a 2D Plane similar to *DJ Max*. Each track of the song has its own "life" meter that once depleted deducts a life. Once a player loses all their lives they reach a game over screen.

Target Platforms

Hyper Groove is targeting Windows based PCs.

Vision Statement

Core Mechanic

The core mechanic behind *Hyper Groove* is pressing buttons in time with the music.

Look and Feel

Hyper Groove will aim to have a mixture of glitch art and pop-surrealist inspired art to dictate its feel. Particular artists that have inspired the art direction for the game are Rob Sheridan, Paul Romano, and Tim Biksup (whose music video for Mastodon's "Dry Bone Valley" was possibly the largest source of inspiration for the art).

Featured Music

The intent is to have music written specifically for the game (2 or 3 pieces). However, failing that, popular music will be used. Examples of potential songs follow.

“Song Name” Artist (*Album Name* Year) [track one / track two]

1. “Came Back Haunted” Nine Inch Nails (*Hesitation Marks* 2013) [vocals / synth - guitar]
2. “Stargasm” Mastodon (*The Hunter* 2011) [vocals / guitar]
3. “Plug-In Baby” Muse (*Origin of Symmetry* 2001) [bass / guitar]

Gameplay Description

Playthrough Narrative

The first thing players are greeted by is the *Hyper Groove* title screen. From there they press either the confirm button to proceed to the main menu. The main menu presents players with the option to choose a song, learn how to play the game, go to an options screen, or quit the game. If players choose the “learn to play” option they will be given a short tutorial on how to play the game. The options screen provides players with an opportunity to disable failing, change the control scheme, or activate a mode where points are deducted each time a note is missed. When players choose the “song select” option they will be greeted by a screen that shows a list of each of the playable songs, with basic song information such as genre, title, year of release, and album art. Once a song is chosen the player proceeds to play the game.

Players are then tasked with playing the song until they meet one of two conditions:

1. They fail the song. This is caused if the player manages to deplete the life meter with zero lives left.
2. They successfully complete the song.

The score screen shows players their final score alongside their overall accuracy and highest note streak.

Affordances

Players’ lack of action (aka missing a note) will cause a sound to play each time a note is missed.

In terms of actual affordances, the player has the following types of interactions at their disposable:

- Tap
  - Simply tap the button as the note hits the “strike zone” – a red line that indicates when the note should be played.
- Hold
  - Press and hold the button as the note hits the “strike zone” and hold it until the tail passes through the strike zone completely.

Scoring and End Conditions

Scoring

Players gain points for hitting notes in time with the music. This means that when the note hits the “strike zone” players press the correct button at the correct time. Depending how accurate their timing is, there will be a UI element that indicates how well the player did. Different levels of accuracy follow, along with how many points they are worth:

- Great! (2X)
- Nice! (1.5X)
- Good! (X)
- Poor! (.5X)
- Ouch! (0)

Once players complete the song they are given a star rating from one to five. The higher the score / percentage of notes hit the more stars the player will earn.

Failing

Players will reach the fail state by depleting their one of their life meters four times. Each time a life meter is depleted, a life is lost. Once all three lives are lost the player has one last opportunity to complete the song without depleting the life m

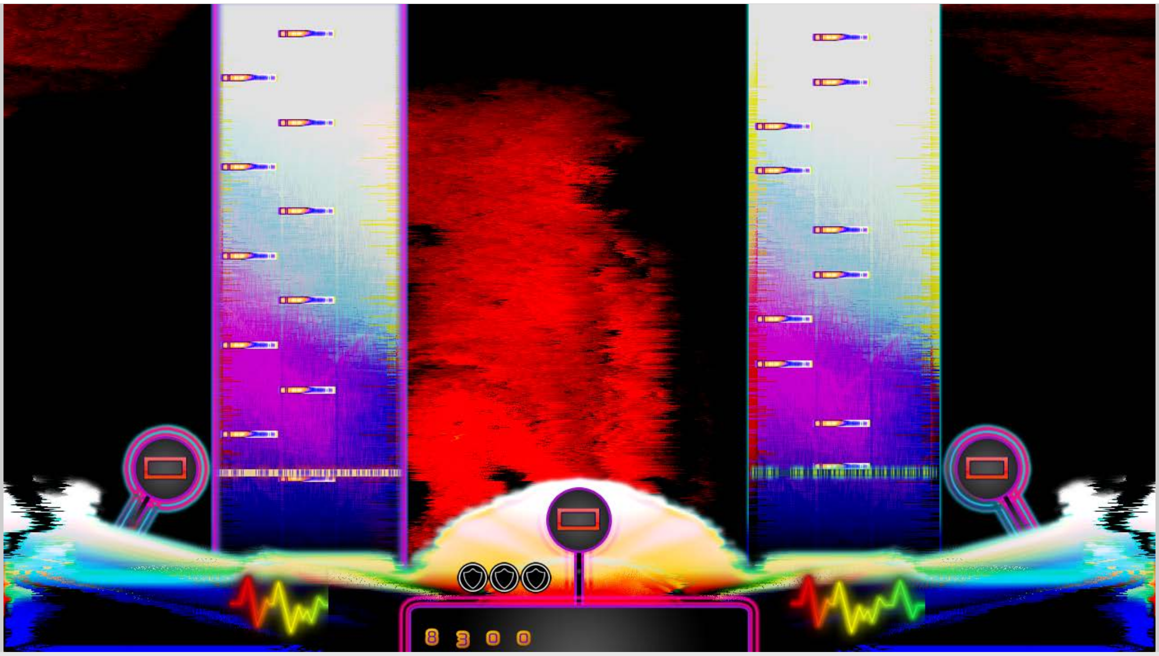
User Interface

The game will be controlled with a keyboard. The controls are as follows:

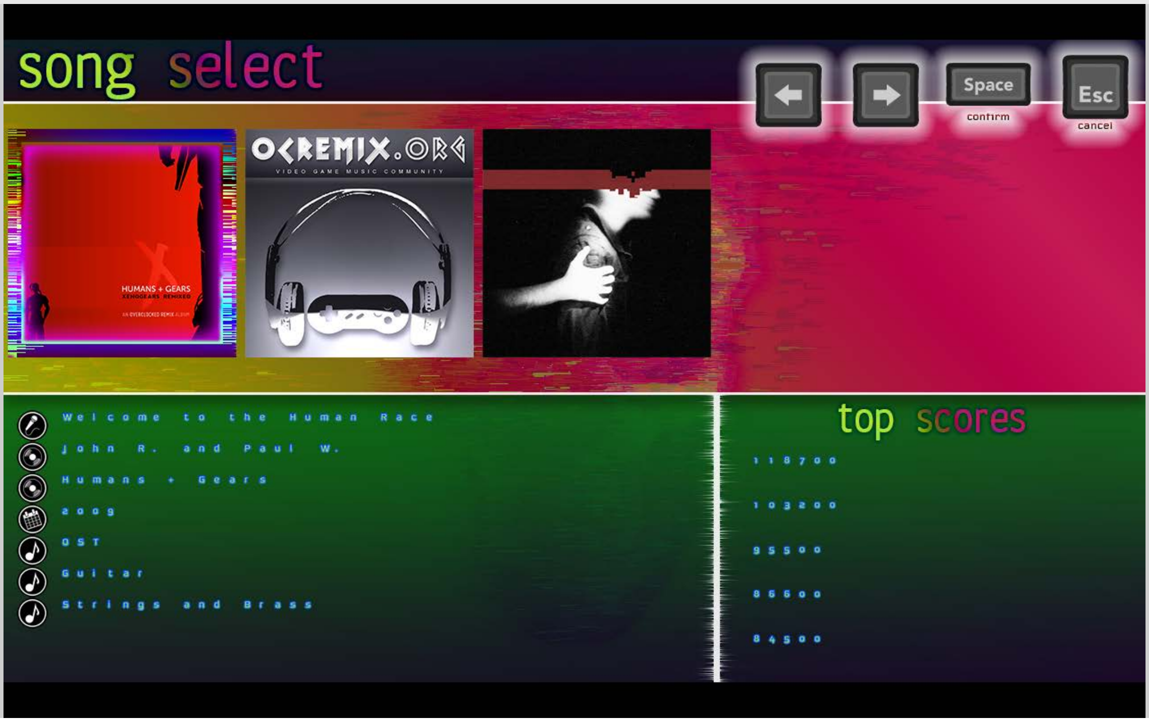
- Spacebar: Pause / Confirm
- Z or A: Left Note
- X or K: Middle Note
- C or L: Right Note
- Arrow Keys: Navigation
- Spacebar: Switch tracks

Please see the next-page for a screen-by-screen break down of the graphical user interface.

User Interface Explanation by Screen



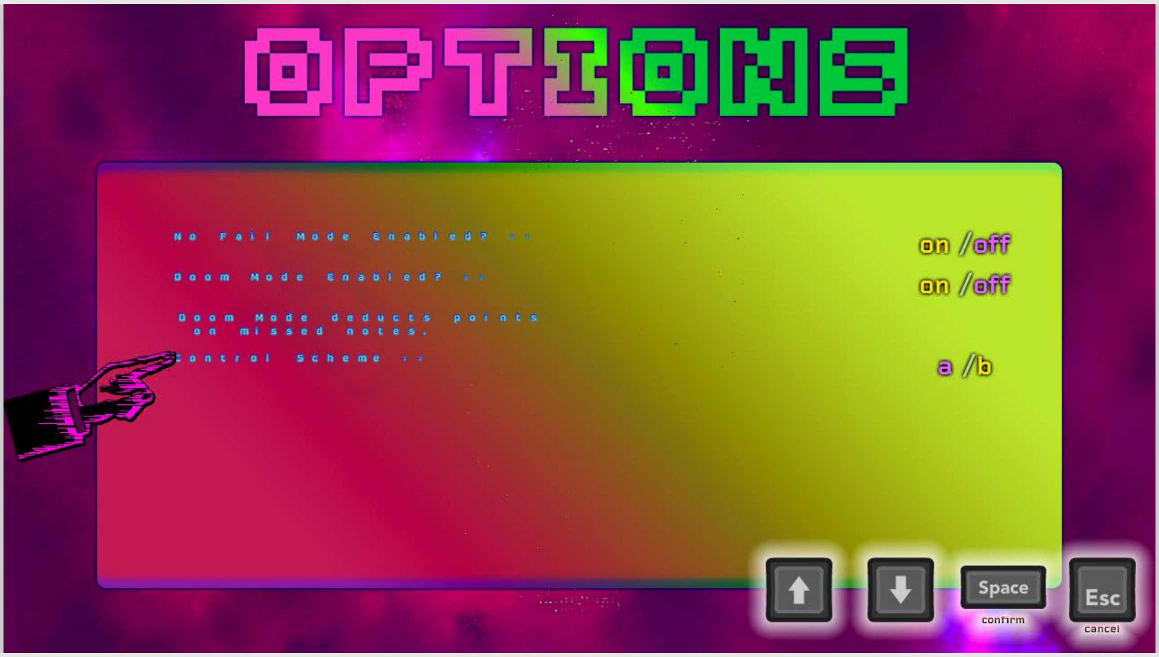
Main gameplay screen. The three red zeroes spread across the screen represent the switch meter. The score is displayed at the very bottom of the screen. The colorful heart-rate monitors represent each track’s life. In the middle are three shields – these represent the amount of lives the player has currently. The two rectangles in the screen, with the bars coming down them represent the note tracks and the notes that need to be played. The purple border around the left hand track indicates that it is the currently active track.



Welcome to the Human Race
John R. and Paul W.
Humans + Gears
2009
OST
Guitar
Strings and Brass

118700
103200
95500
86600
84500

Song select screen. The top half of the screen displays album art of all the songs. Title, artist, album, year of release, genre (not in picture), and instruments played are highlighted on the left hand side of the divider on the bottom of the screen. On the right hand side is the top five scores.



OPTIONS

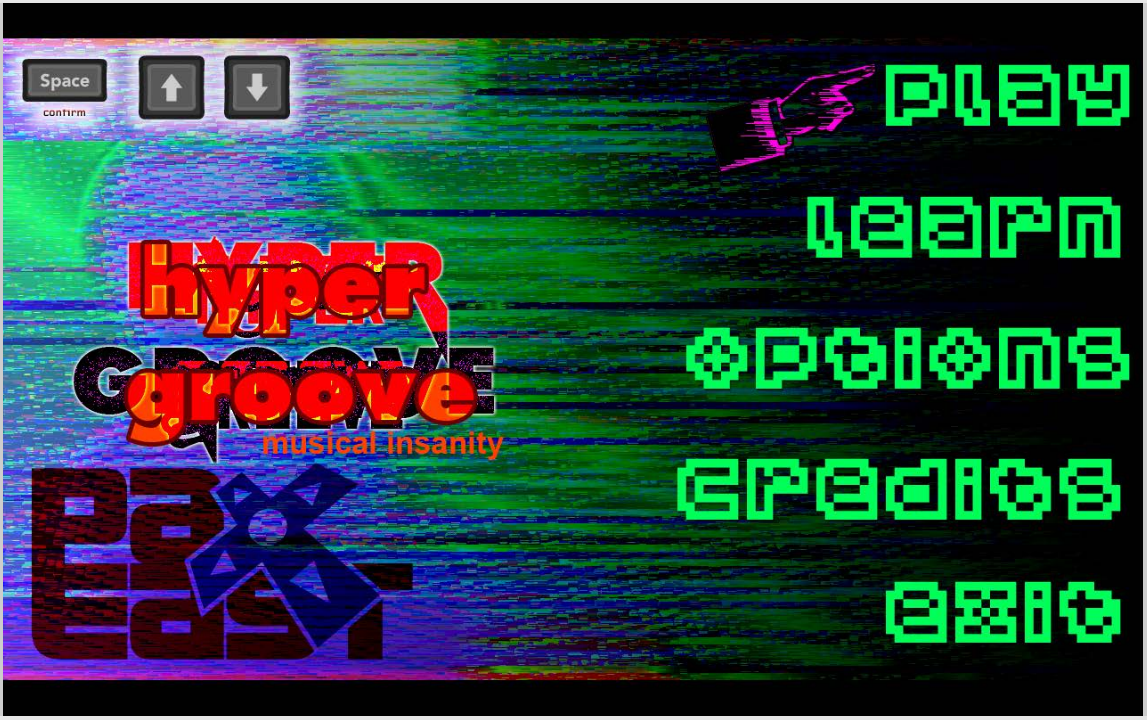
No Fail Mode Enabled? :: on /off

Doom Mode Enabled? :: on /off

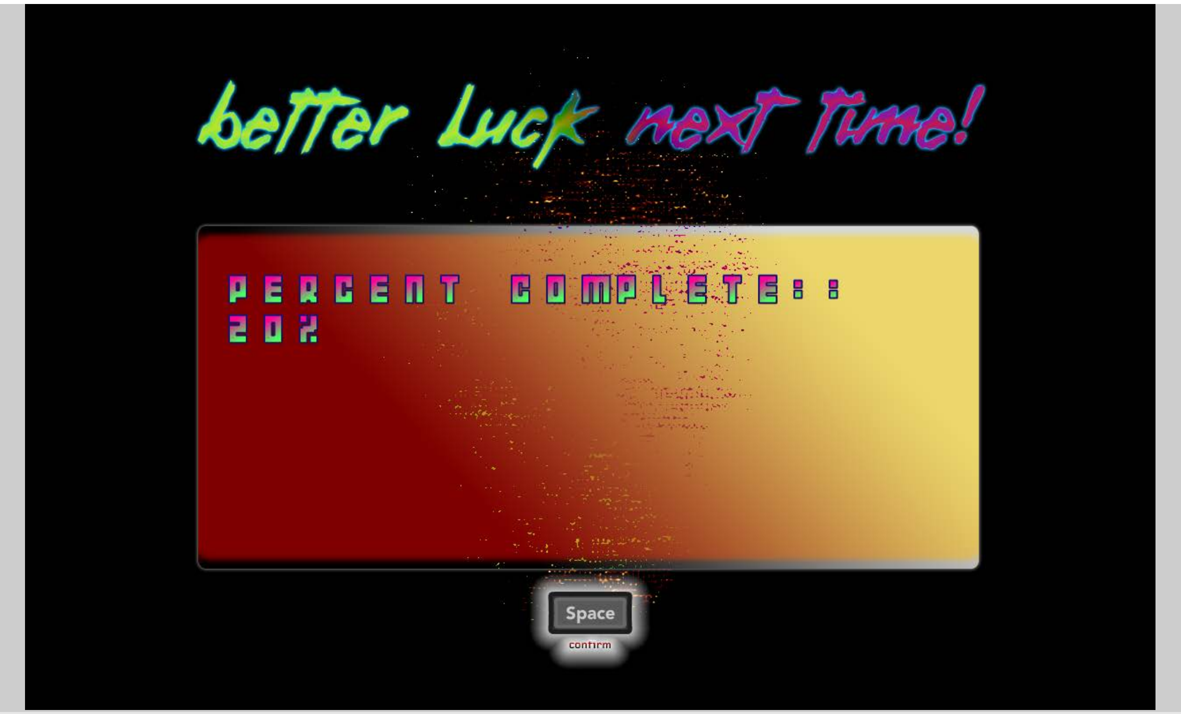
Doom Mode deducts points on missed notes.

Control Scheme :: a /b

Options screen. It allows players to choose their control scheme, turn on the ability to play a song without failing, or to activate "doom mode", which makes the game harder.



Main menu. It acts as a hub that allows players to get to the different screens.

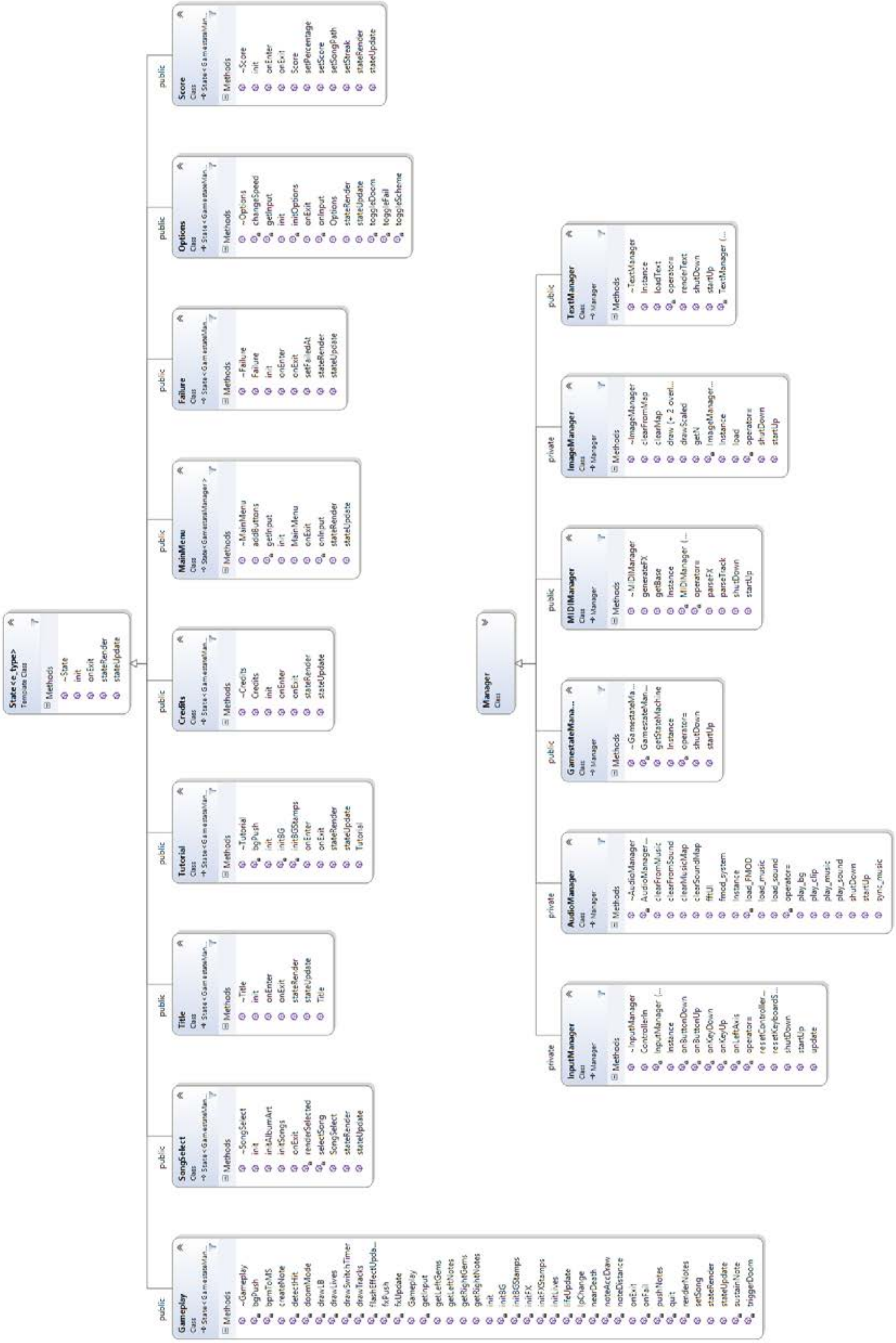


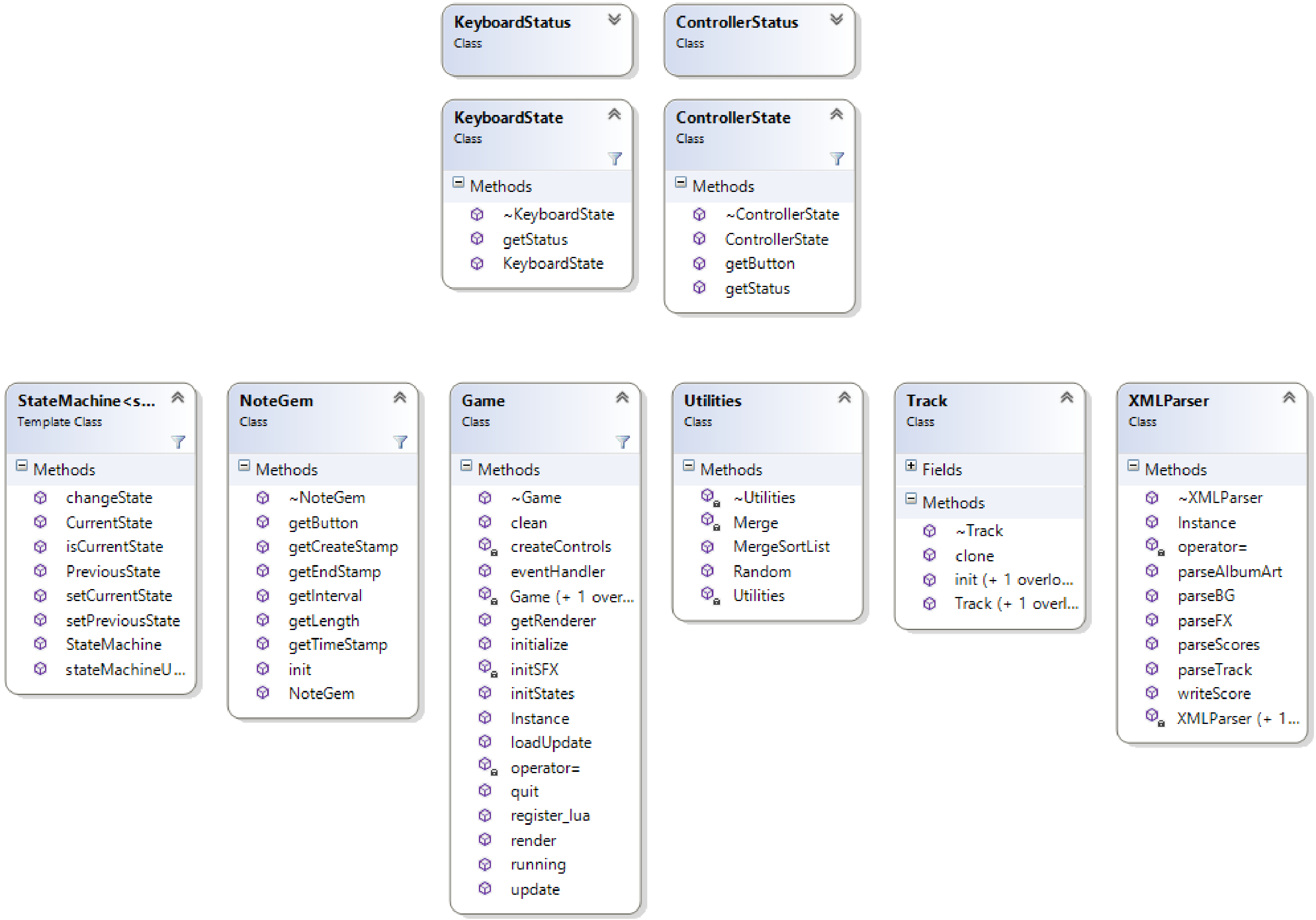
better Luck next Time!

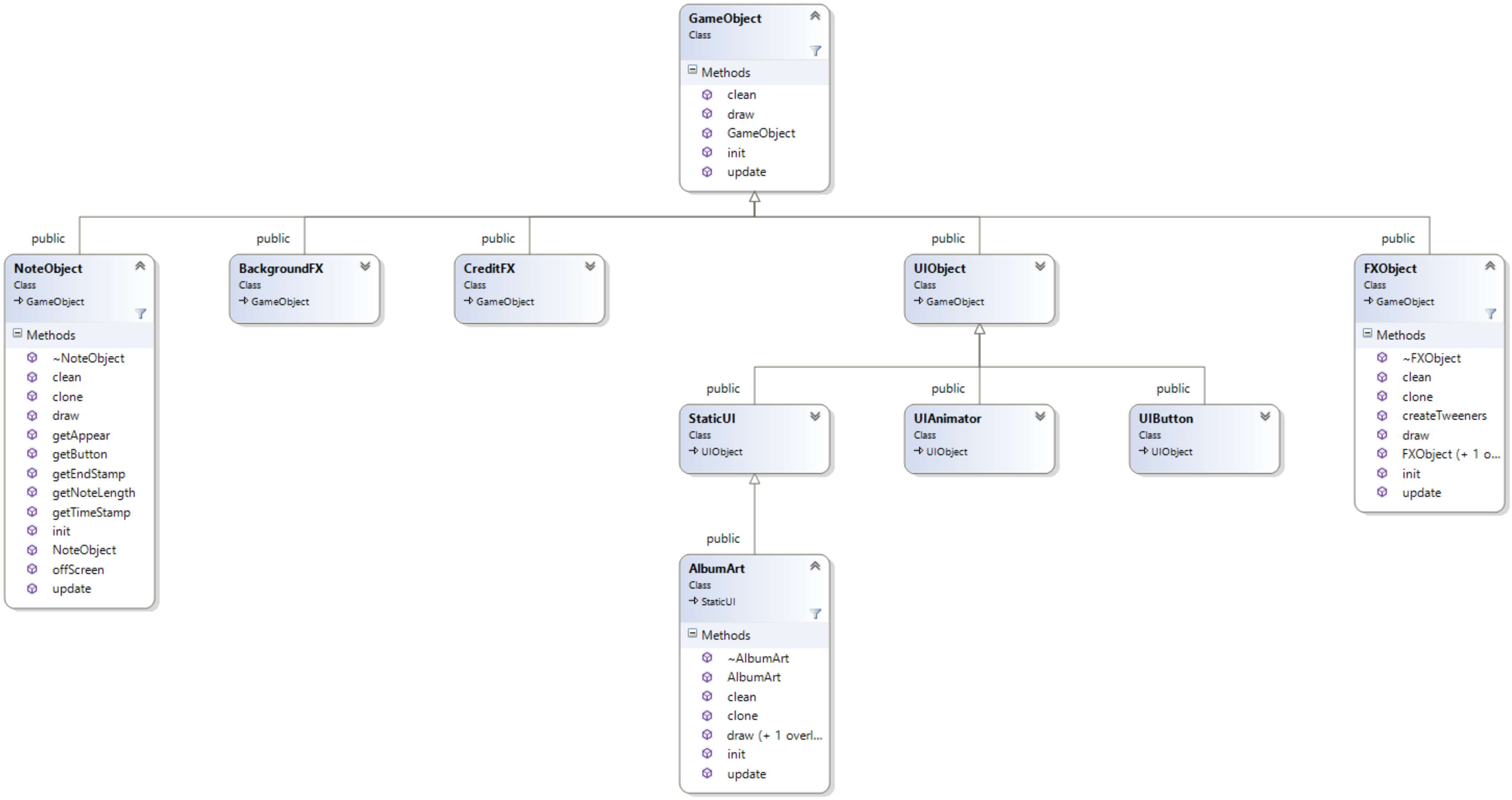
PERCENT COMPLETE::
20%

Song failure screen. From here players can return to the main menu. It tells them how far they got into the song.

Appendix 2: C++ Class Diagrams




KeyboardStatus
Class
ControllerStatus
Class
KeyboardState
Class
Methods
~KeyboardState
getStatus
KeyboardState
ControllerState
Class
Methods
~ControllerState
ControllerState
getButton
getStatus
StateMachine<s...
Template Class
Methods
changeState
CurrentState
isCurrentState
PreviousState
setCurrentState
setPreviousState
StateMachine
stateMachineU...
NoteGem
Class
Methods
~NoteGem
getButton
getCreateStamp
getEndStamp
getInterval
getLength
getTimeStamp
init
NoteGem
Game
Class
Methods
~Game
clean
createControls
eventHandler
Game (+ 1 over...
getRenderer
initialize
initSFX
initStates
Instance
loadUpdate
operator=
quit
register_lua
render
running
update
Utilities
Class
Methods
~Utilities
Merge
MergeSortList
Random
Utilities
Track
Class
Fields
Methods
~Track
clone
init (+ 1 overlo...
Track (+ 1 overl...
XMLParser
Class
Methods
~XMLParser
Instance
operator=
parseAlbumArt
parseBG
parseFX
parseScores
parseTrack
writeScore
XMLParser (+ 1...


GameObject
Class
Methods
clean
draw
GameObject
init
update
public
NoteObject
Class
GameObject
Methods
~NoteObject
clean
clone
draw
getAppear
getButton
getEndStamp
getNoteLength
getTimeStamp
init
NoteObject
offScreen
update
public
BackgroundFX
Class
GameObject
public
CreditFX
Class
GameObject
public
UIObject
Class
GameObject
public
StaticUI
Class
UIObject
public
UIAnimator
Class
UIObject
public
UIButton
Class
UIObject
public
AlbumArt
Class
StaticUI
Methods
~AlbumArt
AlbumArt
clean
clone
draw (+ 1 overl...
init
update
public
FXObject
Class
GameObject
Methods
~FXObject
clean
clone
createTweeners
draw
FXObject (+ 1 o...
init
update