

ボカ踊る - Rhythm Gaming and Artificial Cinematography in Virtual Reality*

DAVID GIANGRAVE, AIDAN O'KEEFE, and PATRICK CRITZ, Worcester Polytechnic Institute, USA
ADVISED BY: DR. GILLIAN SMITH, Worcester Polytechnic Institute, USA

Vocaodoru is a virtual reality rhythm game centered around two novel components. The gameplay of Vocaodoru is a never-before-seen pose-based gameplay system that uses a player's measurements to adapt gameplay to their needs. Tied to the gameplay is a human-in-the-loop utility AI that controls a cinematographic camera to allow streamers to broadcast a more interesting, dynamic view of the player. We discuss our efforts to develop and connect these components and how we plan to continue development after the conclusion of the WPI MQP.

CCS Concepts: • **Human-centered computing** → **Virtual reality**; *Heuristic evaluations*; • **Computing methodologies** → **Computational photography**;

Additional Key Words and Phrases: Virtual Reality, Artificial Intelligence, Cinematography, Reinforcement Learning, Unreal Engine

*Vocaodoru

Authors' addresses: David Giangrave, dlgiangrave@wpi.edu; Aidan O'keefe, ajokeefe@wpi.edu; Patrick Critz, picritz@wpi.edu, Worcester Polytechnic Institute, 100 Institute Rd, Worcester, MA, 01609, USA; Advised by: Dr. Gillian Smith, gmsmith@wpi.edu, Worcester Polytechnic Institute, 100 Institute Rd, Worcester, MA, 01609, USA.

A Major Qualifying Project Report submitted to the faculty of WORCESTER POLYTECHNIC INSTITUTE In partial fulfillment of the requirements for the Degree of Bachelor of Science

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).

ACKNOWLEDGEMENTS

The Vocaodoru Team would like to acknowledge Dr. Gillian Smith of Worcester Polytechnic Institute and Dr. Ruck Thawonmas of Ritsumeikan University for their advice and guidance throughout the project. Without them, none of this could have happened.

We would also like to thank the students at Ritsumeikan University who were gracious hosts and worked with us on our joint project. Their welcoming demeanor and great attitudes set an example that we hoped to follow.

The team would also like to thank our fellow Japan MQP students, especially Natalie Bloniarz and Dan Chao, for all their help and support throughout the project. We couldn't have created such a successful project without their backing and their input.

We would finally like to thank our family and friends for putting up with our long hours, complaints, and overspending while in Japan. We truly owe them the world for all they've done for us.

CONTENTS

Abstract	1
Contents	3
List of Figures	5
List of Tables	5
1 Introduction	6
2 Background	7
2.1 Virtual Reality in Japan	7
2.2 Vocaloids	8
2.3 Games As a Product, Platform, and Service	9
2.4 Twitter Memes	10
2.5 The Unreal Engine Game Structure	12
3 Design and Gameplay	13
3.1 Content Sourcing and Art Design	13
3.2 Environment Design	14
3.3 Gameplay Design	14
3.4 User Interface Design	16
4 Gameplay Implementation	18
4.1 Technical Overview	18
4.2 3D Art	18
4.3 Song File Format (.vdg)	21
4.4 Gameplay	23
4.5 Pose Data Generation	25
4.6 Judgement System	27
5 Utilities	27
5.1 The AI Idol Camera System	27
5.2 File IO	31
5.3 Twitch Integration	33
5.4 Data Persistence	35
6 Cinematic Artificial Intelligence	36
6.1 Utility-Based Artificial Intelligence	37
6.2 Defining a Shot	38
6.3 Selecting a Shot	43
6.4 Creating the Framing	44
6.5 Positioning the Camera	45
6.6 The Camera Queue	46
6.7 Reinforcement Learning and the Multi-armed Bandit	47

4 • D. Giangrave, A. O’Keefe, P. Critz.

7	Testing Process	50
7.1	Paper Prototype	50
7.2	Gameplay Testing	51
8	Post-Mortem	53
8.1	What Did We Learn?	53
8.2	What Went Right?	53
8.3	What Went Wrong?	54
8.4	Future Developments	54
A	IRB Consent Form and Playtest Survey	56
B	Song File Example	65
C	Puzzle-Solving Robot	66
D	Authorship	67
	References	67

LIST OF FIGURES

1	Tweet from Hoshino_Rico	11
2	The final stage implemented in the game world.	14
3	Mockups of the character and song selection menus.	17
4	In-game user interface mockup, displaying sequencer bar and performance bar.	18
5	a: Untextured model directly after import. b: Hand-retextured model used in trailers	20
6	Diagram demonstrating change in frame of reference for player (World-space -> Actor-space)	24
7	Pose image comparison: pre- and post-processing	26
8	Validation for a FIdolCameraKeyframe	30
9	AIdolCamera Tick Function	31
10	Load all poses from folder to array.	32
11	UTwitchGameInstance::InitConnection	33
12	Twitch setup menu.	34
13	Vote Registration	35
14	a: Options menu with HMD disabled. b: Options menu with HMD enabled.	36
15	The general utility function.	38
16	The 7 utility-based components that create a cinematic shot.	39
17	Shot type hierarchical classifications	39
18	Standard shots location sphere range: left is theta range, right is phi range.	43
19	Utility function equation	44
20	Vertical field of view conversion equations	46
21	Representation of location sphere subsection determined using spherical coordinate ranges.	47
22	The typical full reinforcement learning loop.[49]	48
23	The epsilon-greedy decision tree. [4]	49
24	Context Bandit Learning Loop	50

LIST OF TABLES

1	The fields of the FIdolCameraKeyframe structure.	28
2	The valid values for the EFocusPointType enumeration.	29
3	The list of values as well as base and scaled equations tied to EKeyframeTransitionType	29
4	The utility parameters for AIdolCamera	30
5	Definition for the FShotRatingData.	34
6	Framing type definitions and vertical field of view ranges in unreal engine units.	40
7	Shot Length	42
8	Focal length and vertical field of view classifications.	45

1 INTRODUCTION

Coming into this project, our team knew that we wanted to create a truly next-gen virtual reality rhythm game experience, and the result is *Vocaodoru*, a gaming and live-streaming experience built from the ground up for VR. *Vocaodoru*, as the name implies, is tied to the concept of Vocaloids- virtual voice synthesizers frequently tied to virtual idol characters. We wanted to target a primarily Japanese market while retaining an American flair, which we believe we have captured with our unique blend of Japanese and Western influences. The end goal of our game was to make the players feel as if they’re an idol on stage performing for fans whether they really are (on a live stream) or are not, and for viewers to feel like they’ve been treated to a concert stream and be just as engaged as if they were seeing their favorite band on Youtube Live.

This report will delve into the background of our project and our influences, as well as the design of the game and our influences on both visual and gameplay design. We will then transition into an overview of the implementation of three major areas of our project: gameplay, utilities and bridging functions, and the cinematographic AI. The gameplay implementation will cover our unique file format, the novel pose-based dance gameplay system, and our automatic pose generation system for creating gameplay from a series of timing data. The utilities implementation will cover file input and output, integration into the livestreaming service Twitch, and other functions that were necessary to connect the gameplay to the AI. Finally, the cinematic AI implementation will cover our new human-in-the-loop learning utility AI that powers our dynamic camera system, presenting stream viewers with a new method to watch and enjoy VR gameplay. After the implementation, we will explain our testing process, the information we’ve gleaned from it, and a post-mortem detailing the project experience and our future plans for *Vocaodoru*.

2 BACKGROUND

2.1 Virtual Reality in Japan

Virtual reality technologies have only recently become widely accepted in the United States, but Japan has found itself well ahead of the curve. Virtual reality, or VR for short, is a highly complicated system that has been compacted into a consumer-ready product. VR systems consist of a set of interweaved parts, including the VR engine and software that provide the immersive display, and a set of input and output devices for the user. The most common output device, and the one that has achieved the most widespread acceptance, is the Head-Mounted Display, or HMD [3]. Input devices have also changed significantly over the years, with both major players in the commercial desktop VR space (Oculus and HTC/Valve, respectively) moving to motion-based, handheld controllers with physical buttons attached.

In Japan, the biggest player in commercial and personal spaces for VR has been the HTC Vive, an inexpensive head-mounted display. HTC has partnered with many large VR spaces in Japan, including the highly successful Shinjuku VR Zone run by Bandai Namco [30]. The VR Zone is not alone, however, with plans to open at least 20 more VR Zone locations across Japan by the end of the current year, and with its first location abroad in London having opened in August 2017. The expected financial impact of commercial VR centers is currently a very tantalizing prospect for arcade owners- Bandai Namco expect attendance of over 1 million in just two years, or ~ 50,000 users per month. With an initial entry fee of ¥800 and a price per play of ¥1000, it would not be unlikely for sales to reach over ¥2 billion over the course of its operations [30].

VR Zones in Japan have also helped to reduce the stigma associated with VR- that gamers or people associated with virtual reality tend to be male, nerdy, and solitary. According to Kunihisa Yagashita, the manager of Bandai Namco's VR operations, the VR Zone was specifically designed to attract "non geek people" [30]. Other VR projects, such as the VR planetarium offered in Tokyo's Koto Ward, have targeted and attracted a primarily female audience. While even the planetarium experience had a game attached, future efforts, such as the VR Art Museum project being prepared by the JTB group, may move away from games altogether [35]. This is in stark contrast to attitudes about VR in the United States, where female users tend to harbor more skepticism towards using VR in public or are wary about relinquishing agency.

It must be noted here that attitudes towards arcades in the US and Japan are very different as well. Many Japanese people treat arcades as a social place- somewhere to hang out with friends or go on dates, and to make new friends. Arcades in the US stand in harsh contrast to this, with video games and gaming being treated as a solitary hobby rather than a social activity outside of some isolated communities [27]. An exception to this is in the rhythm and dancing game communities, which have a greater sense of camaraderie and a much larger social aspect due to their nature. Many games, such as *Dance Dance Revolution*, have released home counterparts to their arcade cabinets which can help shyer players to enjoy the game, without the prying eyes of other arcadegoers [27]. Despite the resurgence of rhythm games in arcades in the US, Japan's massive, \$ 6 billion USD arcade market [43] for a country of just 127 million far outstrips the US's \$ 2 billion revenue with almost three times the population [16]. We believe that our product, as an experience that would fit in a living room just as well as an arcade, we are in a prime position to tackle both the Japanese and American market.

Japan is also a country that is on the forefront of emerging entertainment media in VR, through new concepts in games and film. Vaio Corporation, a Sony spinoff company, has begun showing movies in virtual reality in its theater in Japan's Shinjuku district in July 2018. The first efforts will feature short films, the first of their kind to be shown in VR in theaters [36]. In terms of home entertainment, manga and anime adaptations are starting to move towards VR for new and unique experiences. *Spice and Wolf VR*, a visual novel that takes place entirely in virtual reality, was announced as one of the first original VR experiences adapting an Anime property [13]. It is safe to conclude that the Japanese have a significant interest in VR, and that they are not skeptical of

VR experiences like some Americans may be, which has helped us narrow our target audience to the Japanese population.

2.2 Vocaloids

Vocaloids are a phenomenon that is uniquely Japanese, but that has achieved widespread appeal both domestically and internationally. Vocaloids are a singing synthesizer technology first developed by Yamaha and released in 2004. Research began in 2000 as a joint effort between Yamaha and Pompeu Fabra that transformed recorded phonemes into a natural sounding singing voice [32]. Vocaloid software were not initially successful, as the first releases did not feature the now-famous characters. It wasn’t until the release of the critically acclaimed Hatsune Miku in the second release- giving users a character that was essentially a blank slate for them to characterize with their work. Immediately after release, Hatsune Miku was flying off the shelves. Thus was born the Vocaloid phenomenon.

Music that used Hatsune Miku grew popular quickly thanks to emerging sites like Nico Nico Douga in Japan. Artists like Livetune, kz, and supercell all trace their roots and their popularity back to some of their very early releases featuring Hatsune Miku [32]. Only 7 years after her release, Hatsune Miku had her first show on American television, performing on the Late Show with David Letterman [34]. Miku’s appeal as an idol is apparent in her touring productions, including her headlining tour Miku Expo and her outing as an opener for Lady Gaga [39]. While Vocaloids are unique, the culture surrounding them is not- many characteristics that American fans associate with Vocaloids, such as the waving of penlights or the highly choreographed dance numbers borrow more directly from Japanese pop and idol culture.

Groups such as Dwango, the owners of the popular Japanese video sharing site Nico Nico Douga, and Domino’s Pizza Japan have used Hatsune Miku to promote their own brands. Dwango host an annual party called the Nico Nico Choparty, a massive concert event held each year in Tokyo. A full third of the show is dedicated to Vocaloid performances, starring Hatsune Miku and other Yamaha and Crypton Media characters such as Kagamine Len & Rin or Megurine Luka [32]. Hatsune Miku has also been used extensively to promote corporate brands in Japan, including a famous cross-promotion with Domino’s Pizza Japan. The disappearance of the famous “Domino’s App feat. Hatsune Miku” commercial from Youtube’s streaming platform caused an uproar in the international Vocaloid community, resulting in a short documentary being produced covering the phenomenon [41]. The primary reason for this is the power of recognition that Hatsune Miku has with a Japanese audience; a 2013 survey run by Tokyo Polytechnic Institute showed that 95% of all respondents could recognize Hatsune Miku, up from 60% in 2010 [26].

Vocaloid producers have used the technology and its creative nature as a springboard to propel their careers to new heights, thanks to the popularity of their original songs. A concept popular in Vocaloid culture is the idea of “Halls of Fame” , which contain songs that have reached a certain number of views on Niconico. The highest of these (so far) is the “Hall of Myths” , which at the time of this writing contains 6 songs [55]. As of October 2010, over 100,000 Vocaloid-related videos had been published on Nico Nico Douga [19]. VocaDB, a crowdsourced database attempting to collect all Vocaloid works by every artist into one place with translated titles, contains information about a large portion of Vocaloid works available on Niconico and Youtube. Over 153,000 songs featuring Vocaloids have been catalogued, produced by over 35,000 artists [54]. Other sources report Hatsune Miku’s output at over 100,000 songs, making her the most credited artist in history [34].

The collaborative and creative nature of Vocaloid products and personalities has influenced our decision to create a robust set of tools and a content pipeline for our project. Hatsune Miku and related characters are closely tied to the concept of “doujin culture” , a phenomenon consisting of fan-made works and communities based around these fan works [26]. Collaboration through programs like MikuMikuDance (MMD), a program used to create 3D animations of Vocaloid characters, and UTAU, a freeware program that allows users to create their own

Vocaloid-like synthesizers, have helped to bolster this phenomenon even further. MikuMikuDance in particular is a target for our project; we believe that MMD files, existing animations, and the robust model library will work as a fantastic source for our content pipeline and allow us to have a dearth of potential content from the very start.

While Japan has already embraced Hatsune Miku and the Vocaloid phenomenon wholeheartedly, acceptance overseas is quickly ramping up to the same level. Concerts for the “digital pop idol” in Mexico and the US have sold out large venues such as the Hammerstein Ballroom and Microsoft Theater [25]. The software for Hatsune Miku is also incredibly popular in its own right, with the first version alone selling 40,000 copies in a single year [25]. However, it hasn’t been until recently that English language versions of Hatsune Miku have been released. The first Miku English module was released in 2013, a full 6 years after her original release [28]. Miku English also has not caught on to the same level as the Japanese version- the most popular Miku English song, “Miku” by Anamanaguchi has achieved a mere 3.3 million views on Youtube [12]. By contrast, a popular recent release on Youtube, “Suna no Wakusei feat. Hatsune Miku” or “DUNE” has amassed over 25.3 million views in less time [56]. As a result, we propose that our game will target a primarily Japanese audience, with the international audience being an added bonus- similar to how official Vocaloid products are released.

2.3 Games As a Product, Platform, and Service

Rhythm games have slowly begun to move away from the traditional, early-2000s business model, where games are released with a flat cost and never touched again. Rhythm games began this shift in the home market with the advent of downloadable content in the commercial sphere, and with the growing popularity of freeware. Games as a service have been a common model for arcade games since the advent of coin-op arcades, with recent additions of networking for arcade cabinets and persistence of data beyond the traditional “high scores” page. We contend that the best way to ensure growth and stability as a rhythm game is to provide either a service that extends the gameplay beyond the initial release, or to open up the game as a platform that other users, companies, and partners can add content to.

The concept of “games as a service” is relatively new, but has been a prominent model across all platforms. For the purpose of this paper, games as a service (GaaS) refers to any game with a continuing revenue stream beyond the initial purchase of the game or cabinet, and any content releases that serve to bolster these revenue streams. In the home market, GaaS exist primarily in the form of microtransactions and downloadable content (DLC) for games short of a new release, or in the form of a subscription service. Games as a service, however, requires that revenue be generated. DLC that is released exclusively for free does not constitute a game being used as a service, but content that is released for free with an option to pay for content early is. Subscription model games, which are the progenitor of the modern GaaS ecosystem, are also included under the games as a service banner.

Home console rhythm games have a clear cut path to monetizing after their release, typically through the sale of cosmetic items and new songs to play. The most ubiquitous example of post-release DLC is perhaps the *Rock Band* series, which has seen a total of 2,631 officially released songs, with 2,328 of them being available for purchase separately of any game in the series [52]. This figure does not include the over 1,000 songs released through the Rock Band Network by fans of the game and independent content creators. While Rock Band is certainly the most ubiquitous of all GaaS home console rhythm games, other titles like *Hatsune Miku: Project Diva Future Tone* take a similar approach- the base game (with a limited track list) is free, and players can buy content packs containing songs to play in the game to extend their experience [47].

Games as a service as an ideology does not solely contain the release of DLC as the primary model for monetizing the game, however. Subscription services and pay-as-you-play games also fall under the GaaS model. Arcade titles in particular are notable as one must pay for each credit they intend to use, with each game offering a set amount of plays for the money that is used on the machine. This model is extremely prevalent in Japan, and has grown with the advent of contactless cards that are used to save data about a player to allow them to continue

their progress across play sessions. Subscription services are popular in both the US and Japan, however, with games like *Just Dance* offering subscriptions for extra DLC content [8] and games like *Sound Voltex III: GRAVITY WARS* offering a subscription to allow consumers to play this typically arcade-only title at home instead [22].

Games as a platform (GaaP), by contrast, refers to the type of primarily freeware games that offer content creation tools for interested players to craft their own experiences. Games like *Stepmania* [53] and *osu!* [38] offer in-game content creation tools that are simple to use and ubiquitous. *Stepmania*, thanks to its long history, has a dearth of content available in its custom “simfile” repositories (a simfile is a special filetype that contains the song and chart data for any given piece of content). A quick search for the acronym “DDR” (*Dance Dance Revolution*) on *stepmaniaonline.net*, the largest *Stepmania* “simfile” repository, results in being able to find 192 simfile packs, containing up to 237 songs each. All of these simfiles are based on songs from official games as well as anything the charter may have found interesting, released for free for any interested players to download. The sheer amount of content available has kept *Stepmania* popular and relevant to rhythm game players since 2001 and resulted in over 6 million downloads over the same period [53].

The success of GaaP for rhythm games, especially those without a commercial license to speak of, has influenced our design decisions greatly. Our project will be released under an open-source license to encourage further development and to open up the platform to a larger audience than would otherwise be possible. While content creation tools of our own have proven to be too complicated to implement currently, the existence of a well-documented pipeline from programs like *MikuMikuDance* into our game software will allow sufficiently technically skilled users to import their own songs and release compiled content for other users to add to their copies of the game. By releasing our project under an open-source license as we finish, we will also allow other interested developers to continue our work and help to create a lasting impression on the rhythm game community.

2.4 Twitter Memes

The most important justification for our project emerged from the burgeoning Japanese Twitter community based around an unlikely source and a similar intellectual property. The popular *Idolm@ster* series has spawned many iterations over its decade-long history, with *The Idolm@ster Cinderella Girls* being its most successful current iteration. A recent update to the *Cinderella Girls* mobile game, *Starlight Stage* or more colloquially “Deresute”, added the ability to use character models and animations from the game with an AR camera app built into the product [5]. This feature’s addition and the subsequent reaction by the Japanese and overseas communities via Twitter show beyond a shadow of a doubt that the world is ready for VR and AR dance games based around popular intellectual properties.

Announced on September 2nd on the official Deresute Twitter account and later translated by the unofficial English account, the AR photo studio was released later that week [6]. Immediately upon release, this feature and its associate hashtag (# テレ ステAR) went viral in Japan, with some tweets reaching over 50,000 total engagements within 24 hours of their posting [24]. While many of these tweets used the feature for its intended purpose, showing the characters in a new, real world environment, many made use of the feature in a transformative way. Tweets from users @ky_gtr and @HoshinoRlco show the idols dancing along to the songs being played by the user in real life [15].

While tweets showing users interacting in a new way are one thing, dance covers and penlight routines that were enacted next to the characters are the perfect demonstration of virtual idol culture and the openness of the Japanese community towards alternate and mixed reality experiences. First and foremost amongst these tweets are the idol fans using the new AR photo studio feature to cheer on their favorite idols with penlights. These interactions range from simple performances, where the user simply shakes their penlight at the idol [48] to highly choreographed wotagei sequences that are based heavily on the song being performed [50]. These



Fig. 1. Tweet from Hoshino_Rlco, showing *The Idolm@ster: Cinderella Girls* character Miria Akagi dancing alongside a real person playing a song on a piano.

interactions mirror those seen at expensive virtual idol concerts like Magical Mirai and Miku Expo, despite the AR photo studio being at a much smaller scale than these massive events. We believe that this shows the enthusiasm of these users towards virtual idols, such as Vocaloids or other characters.

Beyond penlight performances, however, some users have begun to use these AR modes to perform dance covers alongside the CG idol character, matching very closely the proposed gameplay of our project. The first tweet of this kind (and most popular) came from Twitter user @mmm_fujiko, who posted a dance cover next to the Deresute character Kimura Natsuki on September 7th shortly after the feature was released [33]. This tweet has accumulated over 25,000 interactions since it was posted and has been an inspiration to other users and cosplayers, who have posted their own versions of this interaction on Twitter after seeing this tweet [1]. This set of interactions has attracted a primarily female audience, which we believe may help expand our target market to include women as a primary audience in the future. This explosion of popularity in dance covers has led us to

believe that our project, if released as a commercial or freeware product, would have the potential of becoming a viral hit in Japan. We believe that VR idol games and opportunities to interact with virtual idol characters match popular cultural trends in Japan and provide us a fantastic way to connect with our audience.

2.5 The Unreal Engine Game Structure

In order to effectively understand the reasoning behind why the utilities are created and placed where they are, we must first understand the UE4 game structure. Unreal Engine 4 has a very defined structure and it expects game developers to expand upon the parts that they need. While a project can be created using the base versions of these classes, expanding on the base classes to suit the projects own needs make the game feel smoother and more polished. In addition it allows us to take advantage of the systems that the engine has been built around. There are four major base systems in unreal that we will discuss as their role in the lifecycle of the play session are extremely important to the creation of *Vocaodoru*’s custom utilities. These are:

- The Game Instance
- The UE4 Level System
- The Game Mode
- Actors

2.5.1 The Game Instance. The first important class is the game instance base class `UGameInstance`. Every project uses a single game instance class. This class is the first thing to start up when the game is initiated and the last thing to shut down when the game is closed. In addition, the game instance base class follows a singleton pattern; after being created it cannot be copied or replaced. This means that all packaged games will ship with a single `UGameInstance` subclass that holds all of the logic necessary for selecting how the game should start up and how the game should shut down. The game instance is also the only class that will not be re-initialized when moving between levels.

2.5.2 The Level System. Almost all game engines have a system that allows for different sections of the game to be broken up into smaller pieces in order to reduce the amount of data that must be loaded at once. Each of these pieces is referred to as a level in Unreal Engine 4. Each level contains within it information about all of the objects that are should be loaded when the player enters that section such as where they are to be placed in 3D space, whether or not they have collision, and what materials to apply to them. When a level is started, it will make sure that all of the objects in it are correctly initialized and placed before the user is loaded in. This enables games to run smoothly during standard gameplay.

However, levels are themselves objects that can be loaded and unloaded. This means that, by default, when a player loads into a new section of the game the level that they were previously in is unloaded, meaning that all of object instances that were in it are destroyed. If the player were to re enter a previous section, that section’s objects would need to be reinitialized. In order to have data that persists across multiple levels, the data can not just be store in the given level.

There are a number of ways to accommodate for data that needs to be passed between levels. For most circumstances, where levels actually represent different phases of the game playing experience, such as moving from the main menu to the game, the data that must persist across levels is either put in the game instance or in an external file. The second option is called level streaming. Level streaming allows for a level to be loaded before the user gets to that section of the game. This method is mainly used in open world games where it would be jarring to the player to need to go to a loading screen every time the want to go to a different section of the map, but the map is also too large to load the entire thing to memory.

In *Vocaodoru*, there are two levels. The first is the menu level where the user can change settings, set up twitch integration, or choose a song to play. The second is the actual gameplay level.

2.5.3 *The Game Mode.* The next class that the utilities deals with is the game mode (AGameMode). In contrast to the game instance which is global to the instance of the game, each level has a game mode class that is in charge of the handling the actual gameplay mechanics for that given level. In addition to the base logic, each game mode subclass also holds information about the default classes such as the player controller, player character, and AI controllers. In *Vocaodoru*, both the menu level and the gameplay level each have their own game mode that is designed for the experience, and actions that the player needs for the specific level.

2.5.4 *Actors.* In Unreal Engine 4, actors are any objects that can be placed in levels. Actors are a subset of base objects, that include added functionality such as network replication capabilities for online games, and support transformations in 3D space allowing them to be directly placed in a level which is not allowed for base object. The common standard is to prefix all actors (subclasses of AActor) with the letter “A” before their class name while subclasses of the base object class (UObject) that are not actors are prefixed with the letter “U” .

3 DESIGN AND GAMEPLAY

3.1 Content Sourcing and Art Design

Art in games is an important characteristic that defines the style and presentation of a title to the masses. Despite our team not having an artist, we have been able to work around this through the use of user-generated content and art asset repurposing to create the majority of our 3D assets for characters. We were also able to use our limited 3D art skills to design a detailed, expressive environment for gameplay to occur in. A combination of these ideas, along with the restrictions of being a VR game with multiple cameras that must render at the same time, resulted in our overall design philosophy- reuse and reduce.

The first tenet of our design philosophy, reuse, refers to the use of premade MikuMikuDance assets for the majority of our content. MikuMikuDance, as described in the Background, is a freeware animation program that has a dedicated content creation community releasing assets daily. We have decided to use default MMD models, namely the included Metal Miku mesh, as the basis for much of our character art. One of the nice features of MMD is the ability to port any animation, also known as a “motion” or “motion data” , onto any skeleton, provided they share the same basic characteristics. As a result, we have a model with the simplest possible skeleton that can be reused with any motion data that we wish to use. By reusing the same skeleton and same mesh, we have cut down on time that is being spent on non-essential portions of the project and also ensured that our visual aesthetic for characters remains consistent throughout.

Reduce, the second pillar of our design philosophy, refers to the process of ensuring that all art assets are as simple as possible to ensure the game runs smoothly. Part of our concern during design was that the character models, when combined with the crowd and the stage, would feature way too many polygons and would slow down machines to the point that gameplay was impossible. As a result, we began researching into methods to shave polygons from models while maintaining the highest image fidelity and ensuring our game looked similar to other freeware titles on the market. This led us to the recently developed Octahedral Imposter feature used on *Fortnite*'s trees during the skydiving sequences. This technology allows us to reduce high-poly models down to an octahedral, or hemi-octahedral texture and low-poly mesh while maintaining a high level of detail through texturing tricks. Our design leverages this technology to ensure we have a fully featured crowd populating our arena while also reducing the poly count to an acceptable level.

Reduce also ties into the amount of extraneous assets that would typically populate a scene to provide extra detail. We believe that some extra assets, such as those that would be seen from the stage by the player, are very important to ensure the player remains immersed. However, elements that will never be seen by the crowd or the spectator camera can be ignored entirely in order to save on poly count and time. As a result, we have decided to create a sparse stage, featuring rafters and stage lights as well as speaker cabinets, but have left out any extraneous assets from our design. The backstage area and parts of the stage that are inaccessible to the

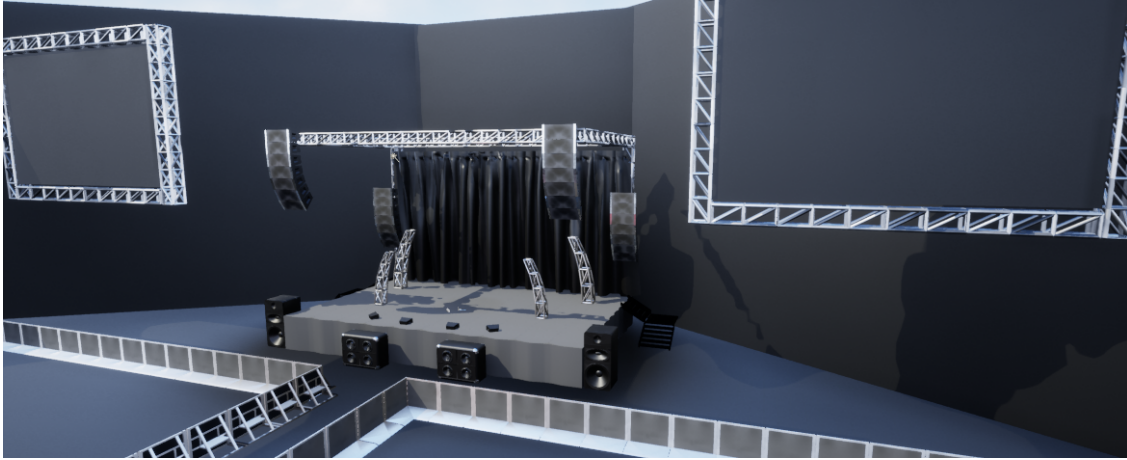


Fig. 2. The final stage implemented in the game world.

player will not feature any models or assets that could potentially slow down the game, and the stage will be devoid of instruments (which may otherwise be present at an idol concert) to save time.

3.2 Environment Design

The aim of the game is to make the player feel as if they are a Vocaloid performing in concert. To create this illusion, the environment must replicate that of an actual concert hall. Due to the nature of the Vocaloid projection and the glowing penlights in the audience all Vocaloid concerts are in dark indoor venues. The remainder of the setting is similar to most any concert, but in order to keep the player in the confined space of the play area, the stage design must remain flat and rectangular. Many MMD stages contain complex structures that allow the character to traverse the environment, however our players should be focused on dancing rather than moving around the stage. As a result of this, the stage environment is fairly simplistic in design, containing typically scaffolding, lights and stage materials.

The player will have a fixed point of view from where they are standing, but the audience or stream will be seeing the stage from all different perspectives including from a distance. As a result, the stage must account for angles of view and account for audience actors to be present in the concert hall. The audience will all be using penlights as an additive effect on the view. The primary source of RGB lighting in the scene will be cast down on the player and audience from the scaffolding above the stage with rotating lighting rigs.

3.3 Gameplay Design

Dancing games are not new to the market, and as with all established markets, differentiation is the key to succeeding. We will identify our three competitors in the dancing game market in Japan, namely *Just Dance* [8], *DANCERUSH: STARDOM* [23], and *Dance Dance Revolution* [20]. We will discuss their core gameplay loops, gameplay mechanics, peripherals, input methods, song selection, and cost to consumer. Finally, we will discuss how our game has been influenced by these games and how we hope to improve on their design elements. We will also discuss how our choice of medium, namely VR, has imposed restrictions on what we are capable of doing, and how future developments in the VR space may allow us to expand our game.

Our primary competitor in the dance game space is the popular series *Just Dance*, created by Ubisoft and released for Wii, Wii U, Nintendo Switch, Xbox 360, and Xbox One. *Just Dance* is a veritable juggernaut in terms

of dance games, designed around a very simple system. The gameplay loop consists of performing to songs that track exactly one point- your hand, or the controller in it. Each dancer you must follow has a single glowing hand which you must follow in order to score points, with more accurate moves being worth more. *Just Dance* is compatible with peripherals like the Wiimote and the Microsoft Kinect, and potentially with any motion controller. *Just Dance* is an annualized franchise, with games starting at \$ 60 USD and DLC being sold. As a result, the song library available for *Just Dance* is staggering, with over 300 songs and dances being available for “Just Dance Unlimited” subscribers for \$ 5/month [8].

DANCERUSH: STARDOM (DRS) is the newest game of the three and features gameplay that combines Kinect-style input with a physical dance mat. Released in Japanese arcades and select Round 1 Bowling and Amusement (a popular Japanese arcade chain that has expanded to the US) locations in the US in March 2018 by Konami, DRS features a small but respectable lineup of Japanese pop music and EDM songs, as well as some crossovers from other Bemani series titles. Gameplay consists of vertically scrolling instructions that tell you to perform one of four actions in a certain spot: step with a specific foot, step and hold/slide your foot along the ground, jump, or “down”. Premium plays allow a player to record their gameplay and automatically upload it to the web for others viewing pleasure, though this feature is only available in the US. DRS is reasonably priced as arcade games go, with STANDARD and LIGHT starts (2 songs each, with Standard having an unlockable “Extra Stage) listed at a price of 100 to 110 yen each, and PREMIUM (1 song only, gameplay is recorded) costing ~ 200 yen [23].

Dance Dance Revolution (DDR) is a classic series that needs no introduction, with over 20 years of releases in arcades and on home consoles thanks to Konami. DDR is an early example of a Vertically-Scrolling Rhythm Game (VSRG), where arrows corresponding to four directional pads below you scroll into catchers located at the top of the screen. Players score points by stepping on the pads at the correct time, with more accurate inputs being worth more points. Outside of simulators like *Stepmania*, the only input method used with DDR is a dance pad or dance mat, consisting of four pads that represent the cardinal directions. The song selection available in *Dance Dance Revolution* is heavily dependent on which version you are using, with most newer titles offering over 100 songs focused primarily on American and Japanese pop music and some hand-crafted titles for Konami’s Bemani series of games. The cost to consumer also varies wildly, with arcade cabinets charging 100 yen for 3 songs, and home console releases going for 6,000 yen plus the cost of your dance mat peripheral (anywhere from 2,000 to 30,000 yen) [20].

Our project has incorporated elements of all of these games in order to create the most complete dancing game experience possible. Our core gameplay loop takes elements from all three games in order to provide an extensive gameplay loop. Using HTC Vive controllers, we use Lighthouse tracking to replicate *Just Dance*’s gameplay on both hands instead of just one, and plan for positional tracking on a grid system (like DRS meets DDR) to make the player move around the play area. Our peripheral system is similar to DRS and *Just Dance*; we utilize motion controllers and planned for positional tracking in tandem to track accuracy. The gameplay loop, however, is where the similarities end.

Song selection and content availability is a key component of rhythm games and is a major motivator of what games people play. Limited song selection and being forced to use only official songs limits gameplay opportunities. Games like *Dance Dance Revolution* and *Just Dance* suffer from a lack of content creation opportunities, resulting in small song libraries. Games like *osu!* [38] and *Stepmania* [53], freeware titles, have song libraries that dwarf these games by orders of magnitude, resulting in players having significantly more choice to play the game the way *they* want to experience it. Our content creation pipeline and planned dynamic song creation tools will allow for a significantly larger song library driven by players, for players.

Our game also differs significantly in pricing model. Our game is planned to be released as a freeware title, open to anyone who wants to play the game for free. This results in a much lower operating cost than arcade titles or subscription services, as our competitors are. However, the upfront costs are significantly larger due to our project being a VR game. A VR headset and related equipment costs well over \$500 for the peripherals

needed to play our game, which can be a steep barrier to entry. However, as any experienced arcadegoer can tell you, this will save you money in the long run.

Being a VR game, we have some practical limitations as to how our game can be played vs these other popular titles. Current VR solutions that have widespread adoption require a tether to be attached to the head-mounted display, meaning that moves like turns or jumps that are found in other games are untenable. This limits us in two ways. The first and most significant way is a limit in terms of what dance moves we can incorporate into our game. This restricts charts based on existing dances to only those that face in a 180 degree cone towards the camera at all times. This tether also prevents the burgeoning freestyle community that exists for games like DDR and DRS to come to our title, as a tether severely limits effective range of motion. In addition, having a VR headset on means that some high-intensity moves or choreography may cause motion sickness or make players trip, which means that we will focus on low-intensity charts for the time being.

3.3.1 Judgement System Design. The judgement system of any rhythm game is its most key component; without an easily readable, fair judgement system, your game will likely never see play. Many Japanese games use English words that one would not always associate with accuracy for their judgements, with *Chunithm*’s Justice Critical and Justice [44], *Ongeki*’s Critical Break [45], and *Sound Voltex*’s Near [22] being the most prominent examples. In keeping with this trend, we decided to move away from standard names for our best and worst judgements. Our four judgement levels in our project are as follows: Fantastic, the best possible judgement; Great, the second best possible judgement; Good, the worst judgement that will not break your combo; and Abysmal, the worst judgement in the game and that will immediately break your combo. While these names may seem as if they were chosen at random, there is actually a significant amount of thought put into these names.

When it comes to signifying how players are doing at a rhythm game, it is important that players are able to identify how they are doing without looking away from the upcoming notes and without needing to read the words that pop up. There are two methods that games commonly employ to provide this information to players: color and silhouette. The easiest method to notice is the color provided for many judgement systems. Different levels of judgement will be colored differently to let players know without having to read exactly how well they are doing. This system is included in our game, with a Fantastic judgement being rainbow, Great being yellow (in keeping with the themes of other popular rhythm games like *Chunithm* and *Ongeki*), Good being green, and Miss being red. Colors are a great signifier for players, but don’t help players who suffer from colorblindness.

The second method that games use to provide this information to players is through the silhouettes or shapes of the words used for judgements. Players can quickly identify what judgement they have received based on the length of the word that pops up, its size, how many lines it takes up, and moreover the shape of the word in and of itself. As many people have seen in the old chain letter, readers can identify and read words where all letters but the first and last are placed randomly inside the word. This means that the most important letters are easily the first and last letters of the word and serve as quick indicators of what word you are reading. Our word choices serve this purpose fantastically- the most important ratings, Fantastic and Abysmal have much different shapes and distinctive first and last letters, while the most similar ratings, Great and Good, share similar characteristics. By combining multiple methods to show players what grades they are getting, we have determined by our analysis of playtesting that we have crafted an unobtrusive system that provides players with important information without getting in the way of their gameplay.

3.4 User Interface Design

The mockups of the user interface were inspired by many previous games, specifically those in the rhythm game genre. Once the player has progressed past the main menu, they are first placed into character selection. There the player selection displays the current player in an idling animation on a circular floating stage. This helps to support the fact that when in concert, the Vocaloids themselves are actually just projections rather than actual

humans. The selection system takes a spinner approach around the player, this is added to give more of an effect to the desired futuristic feeling of the character selection menu. We hope this creates a better effect on the player than simply selecting a name from a drop-down list.

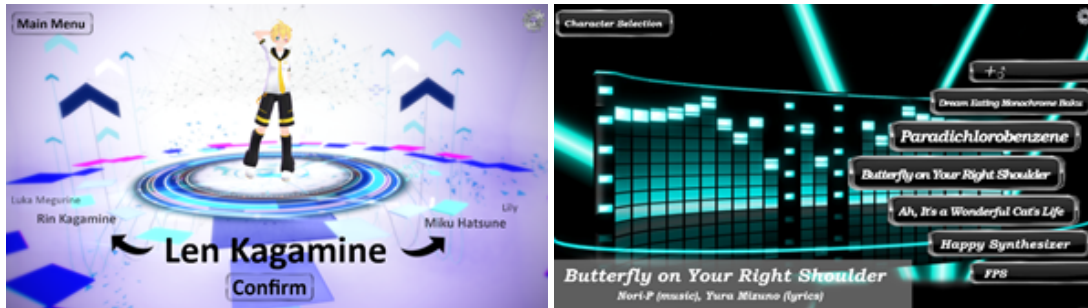


Fig. 3. Mockups of the character and song selection menus.

After selecting a character is the song selection menu. This menu is the one that is most inspired by other rhythm games, due to its radial song section design seen in the figure above. This style of song selection is extremely popular, and we hope to use this as our ode to rhythm games. Behind the radial menu will be a virtual equalizer. This equalizer will be playing a sample of the song the user currently has selected. Like the character selection menu, the hope of the equalizer is to keep with futuristic style of Vocaloids.

The in-game user interface focuses primarily on the pose sequencer found at the bottom of the screen. During gameplay, the user will be shown where to move in 2 different forms. The sequencer bar is the 2D version of this, however they will also have a 3D guide in front of them. At the center of the sequencer bar is the target frame. As each pose moves across the bottom of the screen, when it is dead center within the target frame, the player should be exactly mimicking that pose. It is at this point in time when it is judged as seen by the miss and perfect of the previous two poses. Several minor frames that will not be scored may be displayed in between to help fill in the motion of the player. This is classified as a secondary pose which is displayed as a partially transparent pose icon. Above the poses is the waveform of the song helping to show the players progression through the song.

Above the player guide is the performance bar. The performance bar builds as the player performs well, typically as they maintain a long streak, however it will drop should they begin to miss poses. When the performance bar empties out, the song is stopped, and the player has failed. This stops players from attempting songs they are incapable of as well as adding a level of difficulty to the game.

The player guide will remain in the area between the progress bar and the sequencer bar. The player guide is the 3D pose guide, however their motions will be perfectly synced up with the song. To show the user where to place their hands in 3D, rings will close in on the location of where the hands should be next. This helps the player see how they should move in a 3D space in the event that the 2D silhouette from the sequencer bar isn't enough to decipher the location. The grid beneath the player will highlight where to move their feet to next.

As we continued designing the rest of the UI elements as well as the main menu level itself, we began by looking at other games for inspiration on how they created their systems. While games like *Beat Saber* [29] had a good menu setup in terms of usability in VR; the harsh red and blue colors in the nearly black scene made for a very intense feeling in the level as a whole. While this works for beat saber, it didn't feel right for designing a game whose predecessors had color palettes made of bright but not extreme colors. We felt that the colors in the *Project Diva* series as well as the visual design created for *Magical Mirai 2018* [31] were a better fit for the game overall. We also designed the space around the player using the same wireframe style sphere that was used in



Fig. 4. In-game user interface mockup, displaying sequencer bar and performance bar.

Project Diva X [46]. Using a basic sphere provided an easy way to surround the player, creating a world without requiring complex art assets.

Additionally we had to adapt for the user being the 3D space. We needed to make sure that the player was oriented towards the North of the play area (as defined during setup). In addition to the menu we placed arrows to the left and right of the player to help them orient themselves in the correct direction. Placing the arrows directly to the sides ensures that they don’t become a visual distraction in the players peripherals when they are navigating the menus as they sit outside the Vive’s 120 degree view area.

4 GAMEPLAY IMPLEMENTATION

4.1 Technical Overview

The gameplay of *Vocaodoru* is powered by several interlocking systems, drawing from the unique properties of our art pipeline and based on other popular freeware titles. We will first detail our 3D character art pipeline and how our planned reuse of MikuMikuDance assets has caused issues related to systems introduced later, but has also been a major time-saving measure for developers and content creators alike. We will also detail the custom file format, the “.vdg” (Vocaloid Dance Game) file, that allows us to store detailed information about a song and the timing of each beat. By combining our models and animations with the file format and a parser, we are then able to dynamically generate the pose data used for the core gameplay loop using a series of utility functions and animation blueprints. Finally, all of this leads into the core judgement system that determines if a player has matched a pose sufficiently or not, based on a relative coordinate system designed to allow players of all shape, size, and level of ability to play the game.

4.2 3D Art

MikuMikuDance and its limitations have been the primary driver of the content we are aiming to include in our game. MikuMikuDance is a freeware program used to animate models, typically of Vocaloid characters like the titular Hatsune Miku, and sync them to music. Content designed for MMD is what we will be using as our proof-of-concept art assets and gameplay files. We have several motivations for doing this. First and foremost, MMD models all use a similar rigging structure to each other, allowing us to use a variety of animations across

multiple models. On top of this, MMD has a dearth of content available for download for non-commercial use, eliminating much of the need for new 3D art assets. Finally, the realistic rigs on each character makes them perfect for playback of recorded dance moves on a model that closely resembles an actual human.

The content pipeline from MikuMikuDance to Unreal Engine 4 is somewhat complicated, and we used several iterations before finding a single pipeline that worked for all but one of our test cases. A lack of documentation for many tools (especially English documentation) contributed heavily to our struggles, but it must also be noted that using MikuMikuDance files in Unreal Engine 4 is a relatively unexplored field. Our original plan was to use the “blender_mmd_tools” plugin by sugiany on Github to import our test models into editing software, bake in the animations, and finally export them to an FBX for use with Unreal Engine. We ran into several issues with this approach, however. Importing models worked fine, though their textures would be stripped. Exporting led to several issues, including invalid file formats and problems with bone structures having no root for our skeletal meshes. In the end, we moved on from Blender without ever having imported a model into Unreal Engine 4.20 properly.

Our next attempt for the content pipeline involved a series of plugins directly into MikuMikuDance, namely MMDBridge and MikuMikuMotion. The pipeline was designed to work directly in MikuMikuDance, with larger files being exported to the more advanced MikuMikuMotion before exporting. MMDBridge would use a script alongside the AVI renderer included with MikuMikuDance to export various file formats (alembic, FBX namely) ready for use in modelling programs and Unreal. This would have been fantastic, had the FBX renderer not crashed every time we attempted to bake in animations.

Left with few options, we then tried exporting to an intermediate format (alembic for Blender) and exporting FBX files through Blender. This was the first method that allowed us to import files to Unreal Engine 4. We imported several static meshes that rendered properly, sans textures, in Unreal, but found that the skeletal structure did not carry over properly through the multiple file format changes. Undeterred, we tried several other formats that Blender could read, only to be stumped with the same issues with our skeletal meshes. We then decided to pivot to a new program entirely.

Maya 2017 became the program of choice for our MikuMikuDance to Unreal Engine 4 pipeline and serves as the basis for the content production system for our game. Thanks to a plugin titled “MMD4Maya” by Github user gameboy12615 (based on the plugin “pmx2fbx” by Stereoarts), we found an easy way to import PMX files (the models themselves) and VMD files (motion and animation files) to Maya 2017. These imports worked immediately, and export directly to FBX files with only minor issues (textures are stripped from models, or are packaged improperly). This process has some quirks, however. A particularly tricky motion, set to the song “Yoiyoi Kokon” by REOL, had keyframes parented to bones above ones moving in the hierarchy, as well as an absurd rate of keyframing (over 600,000 keyframes for a 90 second animation). As a result, exports made using this motion file would crash Unreal after about 2 hours of the importing process. In order to avoid complications, however, we have limited ourselves to songs that are approximately 90-110 seconds in length, and approximately 2-5,000 keyframes.

MikuMikuDance assets have a few quirks that caused their implementation in-game to be a more involved process than standard UE4 assets. This can primarily be attributed to their texture format and non-standard skeletal meshes, along with the need to map these skeletons onto each other for skeletal retargeting. As a result, the art pipeline has been more complex than other games of this scale. In addition, the animation files associated with MikuMikuDance motion imports have some quirks that complicate the gameplay later on, which will be covered in subsection 4.4.

Textures for MMD assets exist as a single png file containing the wrap for the entire model at once in many cases, which can make it difficult for Unreal to import these textures properly. As a result, models must be retextured inside of Unreal in order to appear the same as their MMD models. Texture region data is preserved and auto-generated materials are created for each region of the model, but with some issues. Textures tend to

appear more pastel than their originals when imported, with some textures being replaced with a standard grey material that does not match the MMD texture at all.

Retexturing each model must be done by hand as there currently exists no method for exporting MMD texture data into Unreal Engine properly at the time of this writing. While creating a tool to remake these textures would be useful in the future, it is unfortunately outside of the scope of this project and considered a low priority based on our initial goals for this project. Each content creator will have to create their own new materials inside the Unreal Engine material editor, which will make additional work, but will also allow for more opportunities for interesting texturing.

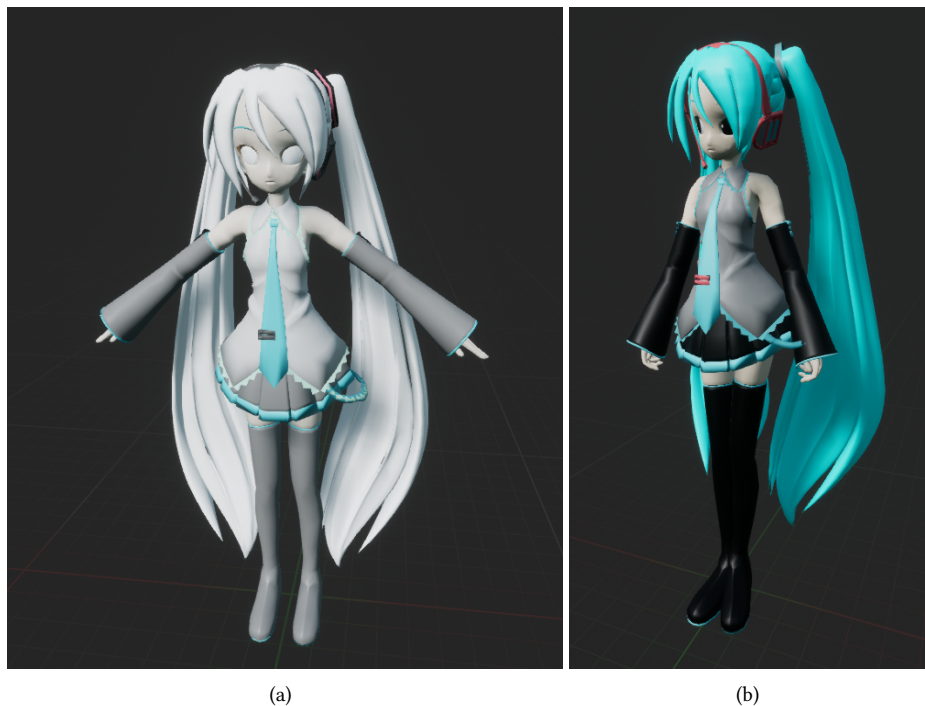


Fig. 5. a: Untextured model directly after import. b: Hand-retextured model used in trailers

Beyond the texture issues, models also have a skeletal hierarchy that differs from model to model due to the MMD4Maya plugin’s renaming scheme. In order for a skeleton to be valid in both Maya and Unreal Engine, bone names may not include Unicode characters and must instead be written entirely in the standard ASCII character set. This can be an issue for some models created by overseas modelers and animators, as many times the bones are named in the native language of the creator. The MikuMikuDance community is based primarily in Japan, which means that bones must be procedurally renamed before importing them. Bones are renamed by the plugin to a standard format: the number of the bone in the hierarchy, a flag to indicate if it is used with inverse kinematics, and then the bone name separated by underscores (example: No_51_i_RightArm).

Another quirk of MikuMikuDance skeletons is the skeletal structure and the varied number of bones in each skeleton. Physics objects like hair and clothing are given bones in the same manner as normal bones and are given an improper amount of emphasis compared to the skeleton. Some models have more bones dedicated to

hair and clothing than to the character skeleton itself! However, because these bones are included in the bone hierarchy as if they are essential skeletal bones, it means that bones that serve the same purpose in multiple skeletons are named slightly differently and must be specified on a per-skeleton basis. This means that we must either store the names of bones that will be used for gameplay or use partial string matching (a fairly expensive operation) to find the bones at run-time.

The varied skeletons and non-standard bone names also make animation retargeting more difficult and reduce the quality of the results. Because skeletons follow only an identical basic structure, it is hard if not impossible to replicate the movement of clothing and hair across models. This also means that skeletal retargeting must be done by hand for each skeleton that we import, resulting in an increased workload as each character is imported. We have reduced the workload by limiting animation retargeting to partial skeletal retargeting such that only the lower body is replicated. This means that we cannot use new skeletons in place of existing dance guides but will allow us to use any character model as the player character during dances.

Once a model and animation pair has been imported into the game, we begin creating the gameplay assets that go along with them. We generate an idle animation based off the reference pose and create a set of three animation blueprints for each model. We require two of these animation blueprints for gameplay, and one for the image preprocessing used to create pose data. For gameplay, an A pose is used as the default position for the actor to ensure that calibration occurs from a standard position, and a second blueprint is used for animation synchronization and playback during gameplay. Our preprocessing blueprint converts the gameplay animation to a single frame animation that we can provide a time in order to get a picture of the pose at a given tick. This system will be detailed in subsection 4.5.

Each animation blueprint is then used in an actor based off of the model that was imported. Actors store information about the model and the song that is associated with them, including the names of important bones for the judgement system. This information is read from a set of text files that describe the basic information about each song including tempo mapping, associated files, timing point information, and judgement data. These text files are currently written by hand based off of inspection of the file in an audio editor such as the free program Audacity. Each set of model, animation, text file, and song data is stored in its own folder in the game's directory for easy access. This sorting system also allows users to quickly and easily explore the game content for themselves and see what songs they have installed.

4.3 Song File Format (.vdg)

The song file format is a new development for this game based off of the file format used extensively in the freeware game *osu!* (developed by Dean “Peppy” Herbert, also known as the creator of internet file-sharing service puu.sh) [38]. Song data is stored in a single text file given its own extension to avoid players accidentally editing the data stored inside. Files are given a .vdg extension, which stands for “Vocaloid Dance Game”, the working title of our project. These files are standard text files separated into a standard template scheme. An example of a .vdg file generated for the song “No title” by Reol based on the motion data by YouTube user ureshiiiiii (<https://www.youtube.com/user/Steve007Fox>) can be found in Appendix B.

4.3.1 Header and Metadata. The first line of every song file will indicate the version number of the file format template that was used. This is to ensure that songs will remain backwards compatible even if the song format changes over time and will tell the engine what data will or will not be included in this file. The first section, tagged with the line “[General]”, stores information about the song's metadata and associated files. The song's title and artist are stored in romaji characters as well as Unicode characters to be displayed on the song select screen. This section also stores the file names of the associated song file (in .wav, .ogg, or .mp3 format) as well as the associated actor and animation blueprint.

Past the important metadata information lies information that enhances the presentation of the game and allows for an online database of songs to be created at some point in the future. The first piece of data included is a “preview time” in milliseconds that tells the game where to begin playing from when the song is hovered over on the song select screen. Following the preview time is the list of tags associated with the song to allow players with a large song library to quickly search for the song. Information about the song’s creator(s), difficulty name, and a unique song ID for web hosting is stored after this but is currently not used in any capacity. These features exist solely to allow the file format and game to be extended after this project has concluded.

4.3.2 Timing Points. The second header in the file format, [Timing], stores information about the metronome for each song based off of single lines that give all the information a song needs. Timing information is stored in segments of the song separated by “ticks” . Ticks are specific dividing lines where the tempo, time signature, scroll velocity, or gameplay information changes relative to the previous section. To quote from the template file, “[Each tick] gives information about where the metronome should start in the following format, comma separated:

```
[Time], [BPM], [TimeSignatureBeatsPerMeasure], [TimeSignatureLengthPerBeat], { SVTick } ,
{ InheritsPreviousTiming } , { DanceStart } , { DanceEnd } "
```

Despite how intimidating this format looks, it is simple to understand and can be written by hand with few, if any, problems. The first two tags are straightforward: [Time] represents the time this tick occurs at in milliseconds, while [BPM] is a floating point representation of the tempo of the song. The two [TimeSignature] tags allow for fine control of rhythm, and represent the two parts of the time signature on standard sheet music. [BeatsPerMeasure] is the top number and represents how many beats fall into a single measure, and [LengthPerBeat] determines the time signature’s “denominator” , or what length of note makes up a beat. This information is used to help determine which beats will make up the basis of the judgement system, if songs will get mid-beat guideline silhouettes on the scrolling UI, and where to place measure dividers on the timeline.

The next four tags are actually flags and store boolean information about the current tick and how it functions in-game. { SVTick} adapts a rhythm game concept used primarily in vertical scrolling rhythm games (VSRGs) like *Beatmania* [21] and *Dance Dance Revolution* [20]. SV stands for “scroll velocity” or “slider velocity” depending on which game you are referring to, but it accomplishes the same thing. In our game, if the { SVTick} flag is checked, the scrolling silhouette display will not display any guides until the next tick is encountered and will suspend judgement until that time. This is used for sections of dances where the character holds a single position for a long or unusual amount of time until the next pose is snapped to, such as in the choreography to songs like “Francium” by Chiitan.

{ InheritsPreviousTiming } , { DanceStart } , and { DanceEnd} are used to as one would expect them to be. { InheritsPreviousTiming} allows someone charting the song to change the BPM while preserving the current position in a measure. This is useful if a tempo change occurs mid-measure, such as on the second or third beat and you don’t want to reset the measure. This technique will be used primarily for songs with variable BPM, such as the classic Dance Dance Revolution song “deltaMAX” by DM Ashura that goes from 100 BPM to 573 BPM over a period of two minutes. { DanceStart} and { DanceEnd} are flagged once per song and determine where judgements begin and end for a song. Both flags must be present in the song for it to be valid, as without these flags set, the song will either never begin to judge the player or will be stuck in game endlessly.

4.3.3 Color Data. The final section of the song format, [Colors] gives information about what color lighting effects should appear in the game. Light colors can appear in one of two areas, either in the stage lighting or in the penlights that appear in the crowd. Each light is given a set of RGB values to determine its color. Finally, the

color is given an integer weight from 1 to 100 that determines how likely the color is to show up in the given area. The sum of weights for all light colors for the stage and penlights must equal 100 to be valid. This color information is not currently used, but is planned as a future feature for the game.

4.4 Gameplay

The core gameplay loop is implemented in blueprints in Unreal Engine, and allows players to quickly verify that their play area is properly set, calibrate to their body type, and begin the song they have selected. The gameplay system is designed to calibrate to any rectangular play area (provided the user is playing with an HTC Vive) and players of any height. This was part of our experience goals- we wanted to ensure that our game could be played by the largest audience possible, and work with players of any body type. This section will detail the calibration process, the judgement system, and the issues we had to work through in order to develop this system.

Because this system works with any room size and body type, it is important that we first get a set of baseline measurements of the room size and the player's arms. We begin by using the SteamVR Chaperone attached to the player to get the corners of the play area. We store these corners for later and mark them during playtesting with particle emitters. Once the play area has been determined, the length of each side is measured and stored to be used with the grid based system, which is part of our future work plans. By not basing our grid on set measurements and instead determining the grid size relative to the player's available space, we have made our game playable in even the smallest of room-scale setups.

Another part of the pre-calibration setup is ensuring that the player is moved to the proper starting location before the gameplay starts. Players should be facing the dance guide actor inside the play area on the "stage" before calibrating. One challenge is preserving the player's position in head-mounted display (HMD) space such that they are still able to use the full play area. Unreal Engine works with several coordinate spaces; we will be talking about world space (position in the play area), HMD space (position of the HMD in the play area), and actor space (position relative to the actor's origin) in this paper. Conventional thinking would lead us to move the player directly to the center of the play area, but this can lead to the player not being centered in both HMD and world space if their HMD is not centered during startup. Instead, we offset the player's position in the world based off their HMD position and rotation in order to ensure that we have a one-to-one correlation between world-space coordinates and HMD space coordinates.

Each song can have a different character model that is used as the dance guide, which means we have to generate new calibration data each time we start a new game. We begin by getting the actor that is used for the dance guide and using partial string matching to find important bones (namely, the shoulders or clavicles and the wrists). We can then use these bones to determine the arm length of the model for use in the judgement system. It is important that arm length is found using the reference pose rather than during the animation playback. The reference pose for UE4 models is an A-pose, which ensures that arms will be fully extended during calibration. We then operate under the assumption that neither the model nor player will be able to extend their arms further than this length, which comes into play during the judgement phase.

After the gamemode is set up and the model is calibrated, player calibration then begins. Before the song can be started, the player needs to provide us with a set of key values that will be used to determine their arm length. The player calibrates with their arms in four different positions: arms hanging low at their sides, arms raised directly above their heads, arms outstretched to the front, arms outstretched to the side. This exhaustive calibration process allows us to determine if the player has a limited range of motion in any specific direction and will allow us to exert more fine control over the judgement system. Arm length is measured in each direction to allow us to account for players with limited range of motion in some respects. Arm length is stored by arm, so players who have limbs of uneven length or with uneven movement ability will not suffer severe gameplay issues due to this problem.

The character we use in *Vocaodoru* is based on a conglomerate of plugins and tutorials, each working hand in hand to provide the exact features we need. Several iterations on the character design were used before settling on an implementation of the `VRSimpleCharacter` from the popular `VRExpansion` plugin for Unreal Engine 4. The `VRExpansion` plugin handles much of the intricacies of managing movement and HMD interaction by default, saving time and fixing many of the issues we encountered with previous implementations. A skeletal mesh is attached to the parent scene component and has its position adjusted with the head-mounted display. Each model is also hooked up to an Inverse Kinematics (IK) rig, as shown in Marco Ghislanzoni’s tutorial on easy VR IK setups, with slight changes for our MMD models [9]. The `VRExpansion` plugin also makes the entire character follow the HMD position and move around the game world as the player moves, solving the player movement problem.

The skeletal mesh is, for gameplay purposes, entirely cosmetic, but solving the problem of making it move with the player was integral for the streaming aspect of our project. Most VR games use separate meshes for the head and motion controllers as it is simpler to set up and does not require complex inverse kinematics to be implemented. Many solutions for VR characters in Unreal Engine do not even attempt to use a skeletal mesh as this is a complicated and time-consuming process. However, since we use a camera that exists outside the player and captures their movements in time with a dance, it was important to us that we have the ability to capture the player’s movements properly and blend them with pre-rendered partial dance animations. The end result is a character whose root movement and upper body movement are controlled by the player, while the lower body is controlled by an animation blueprint, and both are seamlessly blended together.

The core gameplay loop follows similar patterns to other rhythm games, but requires some math to shift all coordinates to the same point of reference. Judgement is given at “ticks” or specific millisecond points throughout a song. At these ticks, the position of the player in the world is compared to the position of the guide in the world, and the position of the player’s hands is compared to the position of the guide’s hands. We can’t check positions in world space as the characters face each other and are offset, so instead we must find a constant point of reference relative to each character. For the player character, this is simple enough. The scene root of the character is position $(0, 0, 0)$ and is placed directly where the character is standing in the world. We must also ensure that the player is always facing in the positive X-axis direction, which we achieve by rotating the model upon calibration and replacing its rotation once the values are captured. We then get the world positions of the hands, subtract the scene root position, and scale it based on the player’s arm length to get our final values.

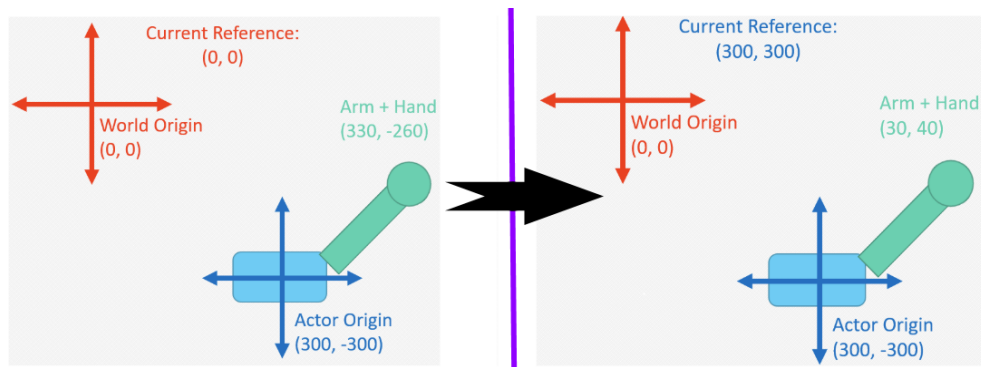


Fig. 6. Diagram demonstrating change in frame of reference for player (World-space -> Actor-space)

For the guide, things get more complicated. MikuMikuDance models have a strange trait in their skeletal structure that means we must do a bit more math to find their position. A characteristic of most animations is

something called “root motion”, movement that can drive the actor’s position in the world when it occurs. MMD models and animations are incapable of root motion; they all feature a single, static root bone that is locked to the position (0, 0, 0) in actor space. Instead, another bone called the “CenterTip” acts as the position of the actor in space outside of their root, locked to the same Z-plane as the root. When we get the position of the dance guide in space, then, we can’t just get the position of the actor but instead must find the position of the CenterTip relative to the position of the actor. Finding the hand positions will then use this adjusted value as its reference point and generate the same relative values as the player model.

Once the relative positions of the player’s limbs have been determined, checking the player’s accuracy becomes trivial. Our current approach is to check the player’s relative position in each axis, allowing for more fine-grain precision over our judgement levels. We find the absolute difference between each relative axis and compare it to our judgement table. If any position is off by more than 70% of the player’s arm length, they are considered to have “missed” entirely. Players who are less than 40% off in any direction are considered “perfect”; less than 60% off, “great”, and less than 70% off, “good”. After the grid-based movement system is implemented, players also must be standing in the proper grid square for this move to count, with grid position being based on HMD position. This process is repeated throughout the song, with a final score being determined based on how many of each grade a player received. For more details about the implementation of the Judgement system, please see subsection 4.6.

4.5 Pose Data Generation

One of the most important affordances provided to a player in a dance game are the pose or dance move indicators provided around the FMV or model used to show the player the dance move. Games such as *Dance Central* use a vertically scrolling “card” system to show players upcoming moves [11], while *Just Dance* uses a horizontally scrolling track that provides information about upcoming moves [8]. We use a segmented horizontal UI with a “catcher” similar to more traditional vertically scrolling rhythm games. It became apparent to us that generating these poses during gameplay would be too taxing on the system, so we began to look into what pre-processing we could do during song generation to simplify this process. This became a multi-step process that can be easily replicated for each song that is added.

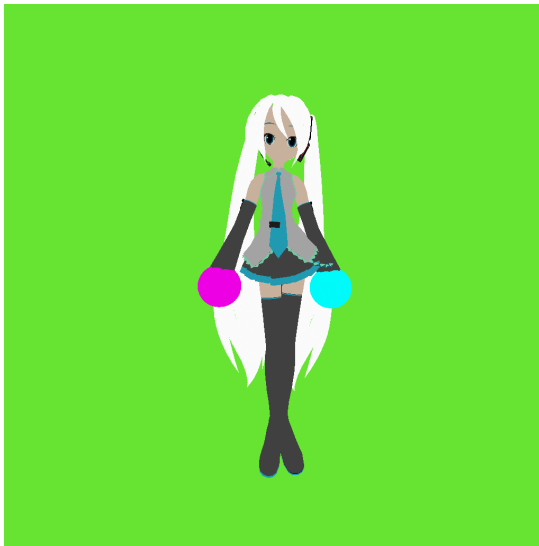
First, models and animations are loaded into the game and prepared into Animation Blueprints as detailed in subsection 4.2. From there, we must set up the scene to prepare for pose data capture. The actor related to the dance we are processing is placed into the world in front of a large wall that is given a “GreenScreen” texture that consists of a bright green base color and a high roughness value to prevent reflection from light sources in the scene. This area can then be lit with a bright, neutral color light if we plan to use lit or shaded models. A Scene Capture 2D actor is then brought into the scene and placed 200-300 units away from the actor, about 90 units above its base position. The Scene Capture 2D actor is set up to use a “Show-Only” list that adds only the green screen and the actor and is set to “RGB BaseColor” capture mode.

After this scene is set up, we attach colored spheres to the model’s hands in order to ensure that they are visible on the images generated. We can call a function from within the gamemode or the player to capture images at each pose. The song data is loaded from a text file into an array of “BeatData” objects, containing the time, BPM, and other info based on the .vdg file format. We then take the time from each beat in this array to set the animation to the pose at that tick and generate an HDR image from the scene. This image has flat colors and provides us with a very stylized, but clear image of the pose the player should be making at that point in the song (albeit with a green screen still placed around the model). This completes the first and most demanding step of the pose generation process.

From here, we can process these images using a utility called ImageMagick which provides us with command-line functions for altering images. Inside the folder containing the images that were generated, we run two commands in sequence to provide us with the final .png images. These commands are as follows:

magick convert * .hdr * .png - Converts the images from HDR format to PNG

magick mogrify -fuzz 10% -alpha set -channel RGBA - fill none -opaque "# 00fd00" -dispose 2 * .png - Removes the background (base color # 00FD00) from the image and disposes of the leftover colors in those areas, resulting in an image with transparency surrounding it.



(a)



(b)

Fig. 7. a: Auto-generated pose image. b: Same image after the two commands have been run (contrast increased for usability).

This set of commands allows us to quickly and easily convert these images to a format that saves a significant amount of space and allows us to reimport them easily. These images are all named in sequence with a trailing number to allow us to easily load them in order during gameplay. These images represent every judgement pose and helper pose that is used in gameplay and are used in the gameplay UI.

One caveat of this generation method is that it is directly tied to Unreal Engine’s actor tick rate. You cannot get an image from a scene of an actor in a pose until that actor has had a tick event so its pose can update. This requires us to put the bulk of our work here inside of a tick-based function, which means that the length of time that each song takes to generate its pose set is tied directly to song length and BPM. The number of poses generated also affects the runtime of ImageMagick’s processing. On our test computer using our test song, the full pose generation process for the song “No title” by Reol (200BPM, ~ 60 seconds, 225 images) took about 10 minutes from start to finish. For longer songs or songs at very high BPMs, this process may take significantly longer even on powerful hardware. This is an area that should be focused on for optimization in future versions.

4.6 Judgement System

The design of the judgement system for our game requires us to think outside of the box with regards to the typical judgement systems found in other rhythm games. By virtue of our judgement tick system and how input is received, it is impossible for a player to fail to provide an input at the proper time. As a result, the traditional timing + input-based judgement system is impossible to implement inside of our game. We must instead approach judgement as a function of the player position at a given time and explore spatial judgement systems in 3D space. This differs even from similar dancing games, which instead use gesture-based judgement or position in 2D space to determine accuracy. We also took steps to ensure that the judgement values line up with expectations of Japanese rhythm game players and made sure that players can tell, at a glance, how their previous move was graded.

First, we will tackle the method by which your judgement is determined in our project. Similar to many modern Japanese arcade titles, our project features 4 distinct judgement levels- three that count as a “hit” and a single “miss” judgement. In many games, your current combo is measured based on how many hits you have had since your last miss judgement. Our game is no different and follows convention up to this point. However, other games will check a player’s timing as their primary method of determining judgement; instead, we check the player’s hand positions to determine their judgement.

The naïve approach to determining judgement would be to treat each hand as a single point and use this two-point approach (left and right) to give the player a rating. However, each hand can instead be placed in any arbitrary position across three axes (X-, Y-, and Z-axis). As a result, we instead check each component offset from the scene root of the player against the component offset of the dance guide, providing us with a set of 6 comparisons to make for maximum accuracy and granularity of judgement. We considered including rotation of each hand as an element of the judgement system, but have determined that current VR systems do not provide players with enough visual feedback to make this a fair system of judging accuracy. Instead, we have stuck with just position checking at the moment, but may include rotation-based judgement as part of the planned future work, for use with more advanced VR hardware.

The judgement system starts each tick by getting the player’s hand positions and converting them to the relative coordinate system that we use, as well as getting the guide’s hand positions in relative space. This ensures that a player can never be “late” or “early” to a judgement, the player is always checked at the precise time that the tick occurs. As a result, our judgement system is based entirely on accuracy. A player, to score a “perfect” judgement (referred to in our game as “Fantastic”), must not be off by 40% or more of their arm length in a given component direction on any arm. The following two levels of judgement allow a player to be up to 60 and 70% of an arm length away before they are given a miss (called “Abysmal” in our game’s lexicon). This system is simple enough to implement and does not cause the game to slow down while checking judgements.

5 UTILITIES

Vocaodoru, as a package, has a number of different smaller systems that are required to make the game function smoothly. This section will attempt to explore each one and its specific importance to the greater project. These pieces are being called “utilities” because they are not a large standalone piece of the project, but instead, more equivalent to the glue that holds the larger systems to each other allowing the game as a whole to function smoothly. These utilities include the AIdolCamera system which is responsible for moving the camera using an array of designated keyframes, the Twitch integration, data persistence, and file IO

5.1 The AIdolCamera System

In order to capture the gameplay for twitch in a cinematic way, we needed to create an object that was modular enough that the AI could have control over setting up the path and display settings for the camera and then

tell the camera to move on it’s own. We designed a camera that would travel between keyframes that the `ACameraDirector` could create and pass into the camera. Each of these keyframes would hold information about the settings for the camera at a give time.

5.1.1 The FIdolCameraKeyframe Structure. To store the information we created a structure called `FIdolCameraKeyframe`. This structure holds the information of a camera at a specific point in time as shown in Table 1. In addition to information about location, the camera stores information about where it should be looking and what the settings on the camera are.

FIdolCameraKeyframe Structure		
Variable Name	Type	Description
Location	FVector	The position the camera should be in when it reaches this keyframe
ActorLocation	Pointer to AActor object	The object that the camera should be looking at when it reaches this keyframe
BoneName	FName	The name of the bone the camera should be looking at when it reaches this keyframe
FocusPoint	FVector (float 3)	The location the camera should be looking at when it reaches this keyframe
FocusPointType	EFocusPointType	Determines which field the camera should use to determine where it is looking (Table 2)
TimeToNextKeyframe	Floating point number	The amount of time the camera should take to get the the next keyframe
TransitionToNextFrame	EKeyframeTransitionType	The type of movement curve the camera should use to get from this keyframe to the next (Table 3)
bIsLastKeyframe	boolean	True if the camera should stop moving when it reaches this keyframe
FocalLength	Floating point number	The focal length of the CineCamera when it reaches this keyframe
FocusDistanceOffset	Floating point number	The focus distance offset of the CineCamera when it reaches this keyframe
Aperture	Floating point number	The aperture of the CineCamera when it reaches this keyframe

Table 1. The fields of the `FIdolCameraKeyframe` structure.

When the camera gets to a certain keyframe it will be focusing on one of three different locations. The first option is a point in space. This is mainly used if we want a scrolling shot that doesn’t track the user specifically but instead just moves from point to point without worrying about the actor. These are commonly used shots in traditional concerts for things like shots over the crowd or when they are looking from the very back of the venue looking towards the stage where zooming in on and locking to a single target doesn’t make much sense.

The second option the camera has for focusing, is to pass in a reference to a specific actor in the world. This functionality, while not complicated, is extremely important because it allows the camera to continue to follow the player even if they are moving around inside the level. This is very similar to the third option, which just allows the camera to focus on a specific bone on a specified actor, adding an even greater level of precision.

Which method is in use is determined by the value in the keyframe's FocusPointType variable. Possible values for this variable are shown in Table 2.

EFocusPointTypeValue	Focus Point
FPT_Point	A float3 location in the world
FPT_Actor	A referenced actor
FPT_Bone	A specifically named bone on a referenced actor

Table 2. The valid values for the EFocusPointType enumeration.

In addition to the focus parameters and the camera parameters, the FIdolCameraKeyframe also has information about how the camera should move from this keyframe to the next. A keyframe has information about how long it should take until the camera reaches the next keyframe, as well as the kind of transition the camera should follow along that path. This transition type is called the EKeyframeTransitionType.

The EKeyframeTransitionType is an enumeration that is used by the AIdolCamera to determine at what rate the camera will move between the keyframes (Table 3). These equations were selected because we felt they gave a good amount of base functionality. Constant and linear equations allow for jump cuts and single speed interpolations (such as shots where the camera only rotates). The ease equations were adapted from *Robert Penner's Easing Functions*[37]. We chose to just stick to the quadratic and quartic curves as their parabolic equations were easy to tell apart during testing.

Transition Type	Enum Value	Fit curve	Scalable Equation
Constant	ICKTT_Constant	$y = 0, 1$	$y = 0, 1$
Linear	ICKTT_Linear	$y = x$	$y = \frac{x}{T}$
Ease Out	ICKTT_EaseOut	$y = -(x - 1)^2 + 1$	$y = -\frac{(x-T)^2}{T^2} + 1$
Sharpened Ease Out	ICKTT_SharpEaseOut	$y = -(x - 1)^4 + 1$	$y = -\frac{(x-T)^4}{T^4} + 1$
Ease In	ICKTT_EaseIn	$y = x^2$	$y = \frac{x^2}{T^2}$
Sharpened Ease In	ICKTT_SharpEaseIn	$y = x^4$	$y = \frac{x^4}{T^4}$

Table 3. The list of values as well as base and scaled equations tied to EKeyframeTransitionType

To prevent error that that would cause the camera using these keyframes to malfunction and potentially cause a fatal crash, we added the ability to validate a single keyframe as well as a list of keyframes as a whole. In order for a single keyframe to be valid, it must meet two requirements. First, every keyframe except for the last one must have a *TimeToNextKeyframe* value greater than 0.0. This is in place to prevent obvious jump cuts over the life of one camera. The other requirement is that if the FocusPointType is not FPT_Point (Table 2). Then the values for the *ActorLocation* (if FPT_Actor or FPT_Bone) and potentially the bone name (if FPT_Bone) must both be valid or the validation will fail.

Additionally, a list of keyframes can be validated as a set. This validation will set *bIsLastKeyframe* to false for all keyframes except the last where it will be set to true. Then each keyframe will be validated individually.

5.1.2 The AIdolCamera Class. The AIdolCamera is a subclass of the Unreal Engine 4 cinematic camera base class ACineCamera. We chose to use the cinematic camera instead of just the base camera class, because it added extra features like focal length, f-stop, and aperture that cause it to function more like a real-world video camera. There are a number of fields in the AIdolCamera that are used directly by the AI. To learn more about those fields

```

bool validateKeyframe(FIdolCameraKeyframe keyframe)
//Validate Timing
IF keyframe.bIsLastKeyFrame is false AND
    keyframe.TimeToNextKeyframe is less than or equal to 0 THEN
    return false
END IF

//Validate Actor for FPT_Actor
IF keyframe.FocusPointType is FPT_Actor and keyframe.ActorLocation is nullptr THEN
    return false
END IF

//Validate Actor and Bone FPT_Bone
IF keyframe.FocusPointType is FPT_Bone THEN
    IF keyframe.ActorLocation is nullptr THEN
        return false
    END IF
    IF
        IF keyframe.ActorLocation does not have bone keyframe.BoneName THEN
            return false
        END IF
    END IF
END IF

return true
END
    
```

Fig. 8. Validation for a FIdolCameraKeyframe

and how they work please refer to the Cinematic Artificial Intelligence section. This section will cover the fields directly used by the utility system for the purposes of moving the camera.

Variable Name	Variable Type	Description
Keyframes	TArray of FIdolCameraKeyframe	The list of keyframes for this instance of the camera to travel through
CurrentKeyframe	integer	The index of the last index in Keyframes to be passed
TotalTimeOnCamera	floating point number	The total amount of time the camera has been active
TimeInKeyframe	floating point number	The amount of time the camera has spent moving in this keyframe
bIsPlaying	boolean	True if the camera is currently moving through its keyframes
bIsFinishedPlaying	boolean	True if the camera has reached its last keyframe

Table 4. The utility parameters for AldolCamera

The first section of variables held is for keyframe information. This contains a list of keyframes as well as the index of the current keyframe. When the AI creates a camera it will also give it a list of keyframes to follow for the duration of its life. The second set of variables holds timing information to help guide its path through the

keyframes that are passed in. In addition to the time in the keyframe, the total time on the camera is kept and updated in case the AI chooses to cut to a different camera once it has spent a certain amount of time on a single camera. It's also important to save this information because these cameras can be paused in the middle of their movement in the AI so desires. This ties into the camera status information variables because it allows for the camera AI to choose to move cameras as soon as the previous one gets to the last keyframe.

Before the camera is told to move, the AI will first request that it validate the keyframes that it has been given (Figure 8). If the validation on all of the keyframes clears then the camera can be told to move. Instead of using an animation, the camera will simply move a specified distance every tick.

```

void Tick(float DeltaTime)
    Update TotalTimeOnCamera
    //Make sure camera is not paused
    IF bIsPlaying THEN
        Update TimeInKeyframe
        set float movementAlpha is the progress through the transition
        Update all fields using movementAlpha as lerp alpha

        //Check for time to update keyframe
        IF TimeInKeyframe >= the current keyframe's TimeToNextKeyframe THEN
            Increment CurrentKeyframe
            Set TimeInKeyframe to 0
            //Check for last keyframe
            IF CurrentKeyframe is last keyframe
                Set bIsPlaying to false
                Set bIsFinishedPlaying to true
            END IF
        END IF
    END IF
END

```

Fig. 9. AIdolCamera Tick Function

The function in Figure 9 is templated depending on the kind of transition that the designer wishes for the keyframe pair. In moving the movement percentage equation to its own function we have opened up the system to be expanded upon. This expandability allows for the ability to add any number of different camera movement effects by simply adding another equation template.

As mentioned before, the AIdolCamera can also be asked to pause. This will set bIsPlaying to false, resulting in the time on the camera continuing to increment every tick while preventing the TimeInCurrentKeyframe from incrementing, stalling the camera's movement. This also means that when the camera is asked to continue playing, there will not be any jumps as the camera begins moving again.

5.2 File IO

Vocaodoru, is designed as a modular game platform where users can create songs and simply add their files to a directory where they can then be dynamically loaded into the game. Each song has two major file components, the Song Data file and the Pose Data folder.

The Song Data file is the .vdg file explained in its section, Song File Format (.vdg). This file houses the metadata of the song, as well as the timing points, and color information. The song loader is a simple file parser that when pointed to a .vdg file will read through line by line and create a UIdolSongData object that just stores all of the

information for the given song. This object will be returned so that information is accessible for other operations such as displaying a song selection menu of all of the songs currently in the user’s library.

The song data loader also will perform some sanity checks on the timing points and color data of the object being created from the file, to guarantee that the user has not made any potential errors that will result in potentially erratic behavior from the engine. For the timing points the check will just ensure that each timing point’s `TimeSignatureLengthPerBeat` is a valid power of two between one and sixty-four inclusive. For the color data points two different checks are performed. The first ensures that a valid color has been passed in by the file. Each of the different parts of the color (Red, Green, and Blue) are only valid if their values are between 0 and 255 inclusive. The validation will first check and if they are outside those boundaries and clamp them if they are. The second check is performed on all of the color points as a whole. The weights from each of the penlight and stage light color points are summed separately and checked to make sure they add up to 100. In the future, we would like to add extra functionality allowing for more creative uses of the `.vdg` file (see Future Work).

The other major part of the File IO system that *Vocaodoru* uses is the `PoseData` loader. Unreal Engine’s default object for displaying flat textures is a `UTexture2D`. In order to load a single file we adapted a method from Rama’s Victory Plugin, to work with a specific file path and file type (`.png` images) [40]. We then created a wrapper for this function that would be able to load all of the `png` images in a folder to an array of `UTexture2D` files that could then be sent to the UI to display.

```
TArray<UTexture2D> LoadSongPoseDataImagesFromFolder(FString FolderPath, int& NumFound)
    TArray<UTexture2D> TextureArray
    Sanitize FolderPath
    IF Sanitized path exists
        FOR EACH .png image in folder
            Load file to memory and convert for UTexture2D
            Add converted file to its place in TextureArray
        END FOR
    END IF
    Set NumFound to the maxium loaded index in TextureArray
    Load Error Texture into TextureArray at all locations that are still NULL
    RETURN TextureArray
END
```

Fig. 10. Load all poses from folder to array.

Some error handling is necessary in the loading process. If there is a problem loading a `png` file and converting it to a `UTexture2D`, attempting to then display the file could result in a crash. If a file is incorrectly loaded, it will not be placed into the final array of textures and the index will be instead left as `NULL`. The other piece of handling is in relation to the order in which files are loaded in. Unreal Engine alphabetizes numbers one character at a time. That mean when the engine returns an alphabetized list of file names, “`filename_10.png`” would be after “`filename_1.png`”, but before “`filename_2.png`”. For this reason the parser will ensure that the files names formatted to follow “`<filename>_<number>.png`” before loading. This means that no matter what order the files are loaded in, the parser can use the number to determine the loaded texture’s final position in the array. However, this may result in empty spaces being left over after all of files were loaded, which is why the error texture is added to all `NULL` indexes before the final array is returned.

5.3 Twitch Integration

One of the major components of *Vocaodoru* is the human-in-the-loop Cinematic Artificial Intelligence. This system is designed to take input in the form of votes from twitch users. In order to accomplish this, we used the a subclass of `UGameInstance` called `UTwitchGameInstance` that would be in charge of tallying and storing the votes from Twitch chat until the AI is ready to process them. The process begins when the game instance is started, where it will attempt to connect to the twitch internet relay chat (IRC) server (`irc.twitch.tv`) on the designated IRC port (6667). If the connection is successfully established using the Unreal Engine 4 `FSocket` and `ISocketSubsystem` modules, then a timer will get registered to monitor the socket and process any data that may be coming.

```
void InitConnection()
    Create a FSocket structure using ISocketSubsystem's default protocol
    Configure the socket to listen to data coming from irc.twitch.tv:6667
    Set the socket so the address is reusable by other sockets in case this one is closed

    Attempt to connect using the configuration in the FSocket structure
    IF connection was successful THEN
        Set bIsConnectedToTwitch to true
        Register a timer to call a function that will process the pending data on the socket
    ELSE
        Set bIsConnectedToTwitch to false
        Close the opened FSocket
    END IF
END
```

Fig. 11. `UTwitchGameInstance::InitConnection`

After a connection is successful the user has the ability to connect to their twitch channel. After entering their channel name in the “Twitch Setup” UI (Figure 12), the game will attempt to join the channel by sending a “JOIN” command to the IRC server. In the event that the game has already connected to a channel, trying to change channels will unfortunately require a restart in order to successfully connect to a new channel. However, this data is saved between instances of the game.

In order to register votes, the `UTwitchGameInstance` must receive data from the `ListenerSocket` set up in `InitConnection` (Figure 11). This is done by creating a function that will check if there is data on the socket. If so, the data will be converted from the default `uint8` transfer format to the UTF-8 character format. This new message is then sent to the parser. However, the `ListenerSocket` is configured such that it will begin refusing connections if there are more than eight currently waiting to be processed. This is why we set up a timer in `InitConnection` to call this function every 1/100 of a second. We believe that this is an adequate timer length because if there is no information on the socket then calling the function will do nothing, but at the same time there is virtually no risk of receiving 800 messages per second.

Once a payload is obtained from the socket and converted, it is then put into a parser. While we do not believe we will have eight messages waiting to be processes every time we call the parse function, there is the potential to have more than one. So first, the payload is broken up into its individual messages and then each message is converted from:

```
:<username>!<username>@<username>.tmi.twitch.tv PRIVMSG # <channel> :<message>
```

to a more easily readable and processable format: `<username>: <message>`

In the process of translating messages the parser looks for two different types of messages. The first are twitch PING messages. These are sent just to make sure the connection didn’t terminate and the parser will respond by

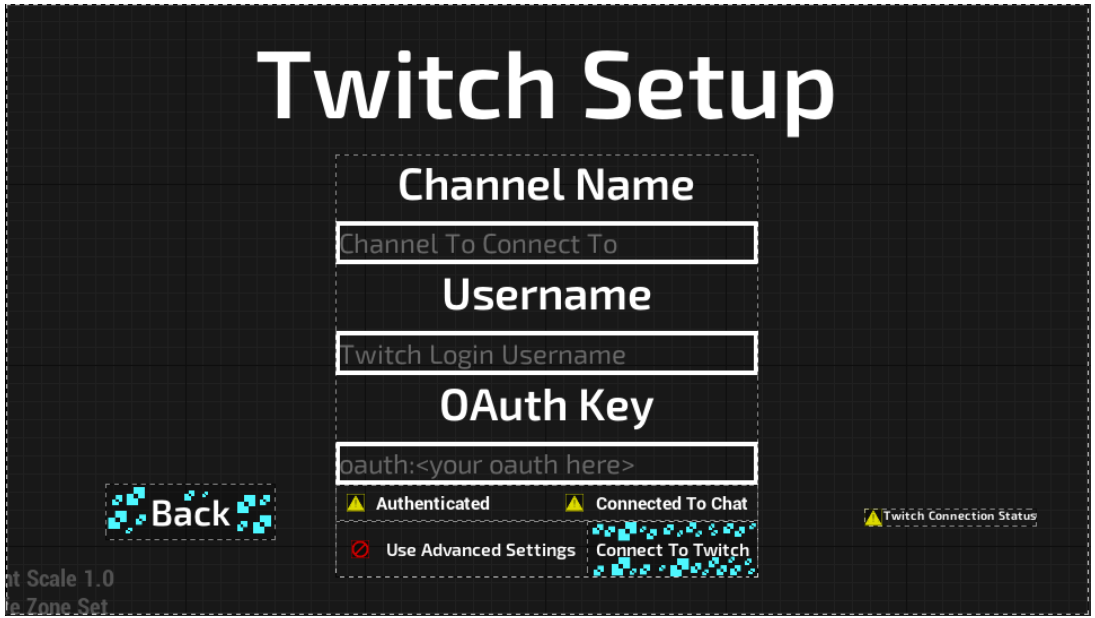


Fig. 12. Twitch setup menu.

sending back a PONG message. The other kind of message the parser is looking for, is one where the message follows the format: <VoteYea/VoteNah> <number>. This format is considered a valid vote by the parser and will send a request that the vote be registered shot specified by the number field.

5.3.1 *Registering a Vote.* In order to allow for fair voting as well as keeping the main thread clear, the UTwitchGame-Instance does not tell the AI everytime a vote comes for a shot. Instead, the AI and game instance work together using a system of locking and polling to lock and unlock different shots as valid and then poll the final results after a certain amount of time. To store the information about all of the logged shots, the game instance has a list of FShotRatingData structures that hold and update the vote information for a shot until the AI wishes to poll it (Table 5).

Variable Name	Variable Type	Variable Description
ShotNumber	integer	The number of the shot who’s voting data this structure holds
PositiveVotes	integer	The total number of positive votes for this shot
NegativeVotes	integer	The total number of negative votes for this shot
Voters	TArray of FString	A list of twitch usernames that have voted for this shot.
bIsVoteable	boolean	True if votes for this shot should be processed; false if they should be ignored

Table 5. Definition for the FShotRatingData.

When the game is initialized, the game instance will allocate an array containing 100 FShotRatingData structures. All of these structures are initialized to have a shot number from 1 to 100, 0 positive or negative voters,

and a clear voter list. In addition `bShotVoteable` is set to false. This is one of the two systems to prevent vote both intentional vote tampering as well as preventing users whose votes got lost in transit from being counted towards a shot they did not intend to vote on.

When a vote is attempted to be registered, the game instance will first look to see if `bShotVoteable` is true for the selected shot. If the shot is open, then the registration method will check to see if the voter's username is already in the list of voters for the shot. If not, then their vote will be registered and their username will be added to the list to prevent vote tampering. When the AI is ready to process the votes for a given shot, it will get the `FShotRatingData` structure for the requested shot from the game instance which will send the structure and reset it to the default state in preparation for the next time it is used.

```
void RegisterVote(FString Voter, bool bPositiveVote, int ShotNumber)
    FShotRatingData ShotStruct = the FShotRatingData struct for ShotNumber
    IF ShotStruct.bIsVoteable THEN
        IF ShotStruct.Voters does not contain Voter THEN
            Add Voter to ShotStruct.Voters
            IF bPositiveVote = true THEN
                Increment ShotStruct.PositiveVotes
            ELSE
                Increment ShotStruct.NegativeVotes
            END IF
        END IF
    END IF
END
```

Fig. 13. Vote Registration

We chose to go with a constant number of shots because even if not all of the shots are used in a song, it is less intensive to store a set number of structures and just reset the fields than to destroy and create new structures after each shot is gotten. However, this system has a few drawbacks. The first, is that in the current iteration, the `FShotRating` data does not store how each person voted for a shot. As a result, there is no way for a user to change their vote on a single shot. The second drawback is for users with extreme lag, as their votes may not be counted at all due to the AI polling and locking the shot structure before they see it and send a message on twitch.

5.4 Data Persistence

As with many games, *Vocaodoru* has information that must be persistent to some extent. This can mean data that is stored across levels, instances of the game, or even can be passed between installs. One of the challenges was coming up with the correct way to store information in all of these cases. The first case applies to data that needs to be stored across levels. This includes information like the Cinematic AI actor, and the twitch connection and authentication information. Because these pieces of information should never be recreated during the life cycle of a single play period, it made the most sense to store this information inside of the `UTwitchGameInstance`. This allowed for both storage across levels but also allowed for the ability to change their initialization depending on the specific launch parameters. As mentioned in `The FIdolCameraKeyframe` Structure, the game instance is the first object to be initialized when the game is opened and is also the last object to be destroyed when the game is closed. The main way this was utilized was during the initialization of the game. Since the user is expecting to be able to have a clean streaming solution when they booth the game, if they boot with a head mounted display (HMD) enabled, the viewport to be streamed will be show on the user's monitor. This spectator view is different than what the user sees when they are playing the game. One of the challenges of this method of loading and

displaying the game is that if user starts the game in HMD mode then the lose all ability to use the keyboard and thus the ability to set up their twitch integration.

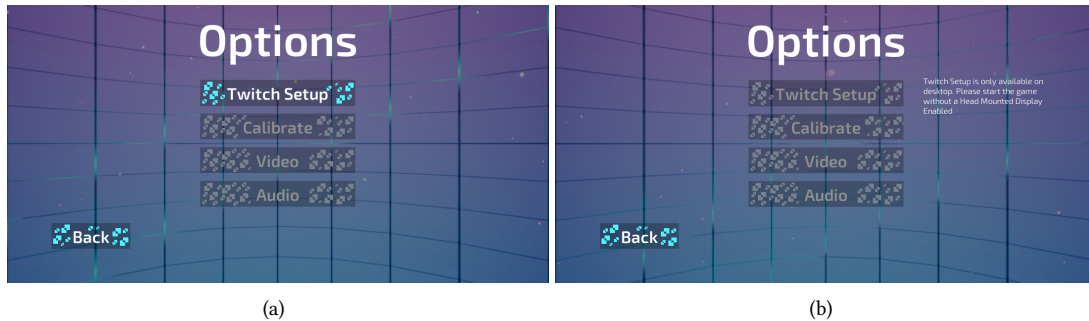


Fig. 14. a: Options menu with HMD disabled. b: Options menu with HMD enabled.

To deal with this, the game instance will first check if the user is booting with an HMD enabled. If so, then the display will try to use the Twitch connection data that was saved from the previous instance to try and connect directly to the user’s channel on load. In addition, the game mode in charge of the main menu level will spawn a 3D widget renderer object in the world and disable access to the Twitch Setup Menu. However, if the game is started without the HMD enabled, the game mode will instead display the menu UI directly to the monitor.

Another case where data persistence is needed, is for information that needs to be saved between instances or even installs of the game. Two great examples of this are saving the scores held in the cinematic AI for long term training, and each song’s data file and pose images. This group of information can be divided even further into data that needs to be human readable, and data that either does not need to be or shouldn’t be.

For data that must be human readable and editable, such as the song data files and pose images, the best way to store this information is as a set of .vdg and .png files. This builds on the games as a platform discussed previously. This method of storage allows for rapid iteration by content creators, allowing them to make content easier and faster than if they had to predefine it all in the engine. The other reason for storing this information separately is to avoid loading all of it at once. Whenever a user attempts to play a song, the song level will be loaded. This level is just a template the will change certain portions based on which song was picked. For this situation, attaching all of the songs to the level directly would result in extremely long load times for users with a large pool of songs. By not hardcoding the data to the level, we achieve shorter load times for users which results in a more seamless both gameplay and viewing experience.

The last kind of data *Vocaodoru* stores between game instances is information that either doesn’t need to or shouldn’t be human readable. This includes both the instance weight scores for the Cinematic AI as well as the login information for Twitch. Instead of store this data to a text file, all of these fields are added to an instance of `UIDolSaveGame` which is a subclass of Unreal Engine’s default save game class `USaveGame`. This object type already has serialization methods that save and load and set of variables as well as encrypting them. For sets of information like Twitch OAuth keys, the data should never be left in a human readable state for any longer than is possible as it presents a potential security risk to the user. Using the `USaveGame` means that the data will be encrypted, protecting the user’s sensitive data.

6 CINEMATIC ARTIFICIAL INTELLIGENCE

One goal of the proposed computational cinematography system is to allow the director to learn from each streamers chat and provide camera angles and direction that conform to their viewers’ taste. We decided to use a

utility-based AI as a result with learning directed by human-in-the-loop interaction with Twitch chat integration. A utility-based AI simplifies the learning system such that users have a direct effect on the weight or score of each camera component [10]. Users can vote on a shot-by-shot basis, with positive and negative votes resulting in a change to each part of the camera component's utility scores. Over time, favored camera shots will receive higher scores and appear more often than others. Limitations prevent any shot type from not having chance of appearing or from occurring multiple times in succession. Using these rules and methods of learning, the viewers are presented with a view that appears to be a professionally made concert music video.

6.1 Utility-Based Artificial Intelligence

Utility theory is used by an artificial intelligence agent to decide on an option by determining which best fits the current situation. The goal is to maximize the expected utility of the selected option [42]. To determine the utility, the agent uses a utility function, assigning a single number to represent the value of that option. The expected utility weights the raw utility scores by their respective probability of occurrences. Once every possible action has an associated utility score, the agent will likely select the highest rated option. Some agents will use different selection methods to appear more random or human-like.

In the event the agent is always selecting the action with the highest expected utility and two options switch back and forth as highest rated action, the agent may appear to stutter between the two actions, never committing to one. The first solution to this would be to allow for a set period of time to pass between action selection, forcing the agent to commit to an action at least temporarily. The second method is to set a cooldown on the utility score of the chosen action by temporarily decreasing its score, allowing for other options to have a higher appeal while not eliminating that actions possibility.

In our context, the camera director will want to select a variety of different shots and should not only consider the best possible option. Instead the camera director will use a weighted random function to select from the best possible shots, but this doesn't limit it to the best, providing opportunity to explore even the worst possible shot. A camera angle will get boring if repeated, which is still very much possible with the weighted random. The cooldown function will provide better diversity to all components of a shot, creating all sorts of permutations of the shot definition. Attributes like cooldown can be added to each action's respective utility function.

The utility function is the sum of all considerations involved. For example, an agent may consider two inputs, the price of an item and its delivery time [10]. Hypothetically, we can determine that a good price is worth twice as much as the deliver time, this is called preference elicitation. From there we can build a function that produces the utility based on the two inputs. Things like time and money are unrelated numbers, forcing all components to be normalized to a standard scale for the utility function, typically 0-1. To do so we need to lock the unscaled values into a range, in the example this could be 0 to 1000 dollars for the price range and 1 to 30 days for delivery time. These normalized values can be easily be added and/or scaled by the system in order to compare against other options. Here the price might be scaled by double that of the delivery time.

Smart utility calculations will do more than take the value of money as it stands. The value of money can be very subjective. To someone with 10 million dollars, spending \$ 100 on a purchase is nothing while a person with \$ 200 dollars would much rather wait longer for delivery on a cheap item [42]. This same concept can be seen when taking a bet or gamble, the wealthy person would be classified as risk seeking while the other is risk adverse and their preference for taking the gamble can be graphed proportional to a logarithm as suggested by Daniel Bernoulli. Each component of the utility score requires a different mapping and calculation, while risk to reward for money is logarithmic, other components may be quadratic or require multivariate input functions.

Multiattribute utility theory is based around this idea that multiple inputs and considerations are involved in the calculation of the utility score. Should each component be independent of one another, one does not affect the

other considerations, they are considered to have mutual preferential independence. In such a case the calculation of the utility score can be simplified down to the following function:

$$V(x_1, \dots, x_n) = \sum_i V_i(x_i)$$

Fig. 15. The general utility function.

Where V_i is the value function referring to the attribute X_i . In the earlier example we can say that $V(\text{Cost}, \text{Delivery Time}) = 2 \cdot \text{Cost} + 1 \cdot \text{Delivery Time}$. Should we want to consider the existing wealth of the customer, we can either add it as another attribute or replace the cost variable with a value function that determines the utility of the given cost based on the customers wealth. Using this basic formula most actions utilities can be determined by assuming independence and certainty of the input attributes.

Each component of the camera will have a combination of several cinematic attributes available to consider as well as the votes from twitch. First and foremost is the stand-alone cinematic score. A shot must be able to stand by itself without considerations of other components or past shots to succeed. After that consideration, we can consider what other components of a shot have already been selected and what the last few shots used. As in the earlier example of time and money, we can simplify our equation down to two attributes, cinematics and votes. The cinematics attribute should not be taken at face value like money and should accept other inputs that could change how the attribute is valued. The resulting utility function is described in further detail in subsection 6.3 and is seen in Figure 19. Before we can fully decide on a utility function however, we must first define what a shot is comprised of in the context of this game.

6.2 Defining a Shot

The first part of creating the cinematics for the stream consists of selecting a single shot style that describes the setup of a camera. A shot can be defined by 7 different utility-based components found in Figure 16. Each component defines an important aspect of the shot that could differ based on the viewers opinion or its cinematic strength. Different components can be combined to potentially improve the overall shot, creating a complex system of utility components. Each component specifies specific restraints that help conform the shot to a specific cinematic type while still providing the camera director class with freedom of control.

6.2.1 Core Components. The primary portion of the shot can be broken down into three layers: focus type, framing type, and shot type. Each layer’s scores are dependent on the selected type in layers above it creating the hierarchical structure outlined in Figure 17. Any missing components in the hierarchy mean that component had a zero score for its utility and was removed as it will never be selected. This structure allows for greater combinational learning of the AI. For example, a close-up combined with a player focus would have a high utility score while a close-up combined with a crowd focus should have a zero for its utility score as it shouldn’t exist in the context of this game. While having this combinational benefit exist over all of the 7 components would be nice, the number of specific utility scores would be astronomical, making the amount of time to learn go up exponentially. Instead we have only these core components use the system and everything else should fall into place around it.

Each component is broken up into an enumeration describing its type. Each layer has a class built around the enumerated type that contains its utility attributes, helper methods, default values and, if applicable, any

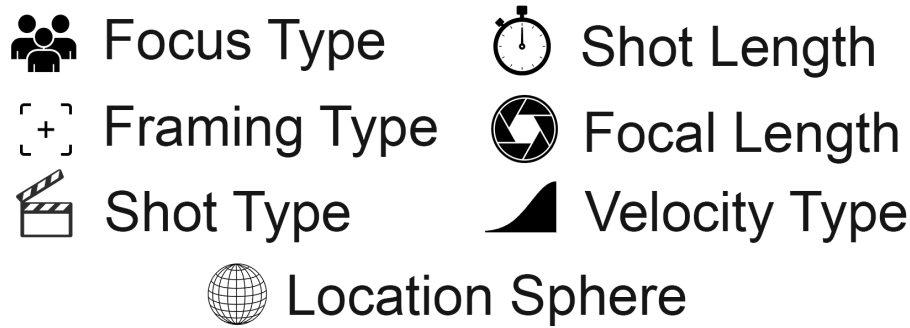


Fig. 16. The 7 utility-based components that create a cinematic shot.

components that fall below it in the hierarchy. This structure style allows the AI to keep the primary hierarchy of utilities in a single array traversing downward to obtain other shot components. The selection process selects from the array using a weighted random function and continues the process for each layer until all components of the shot has been selected.

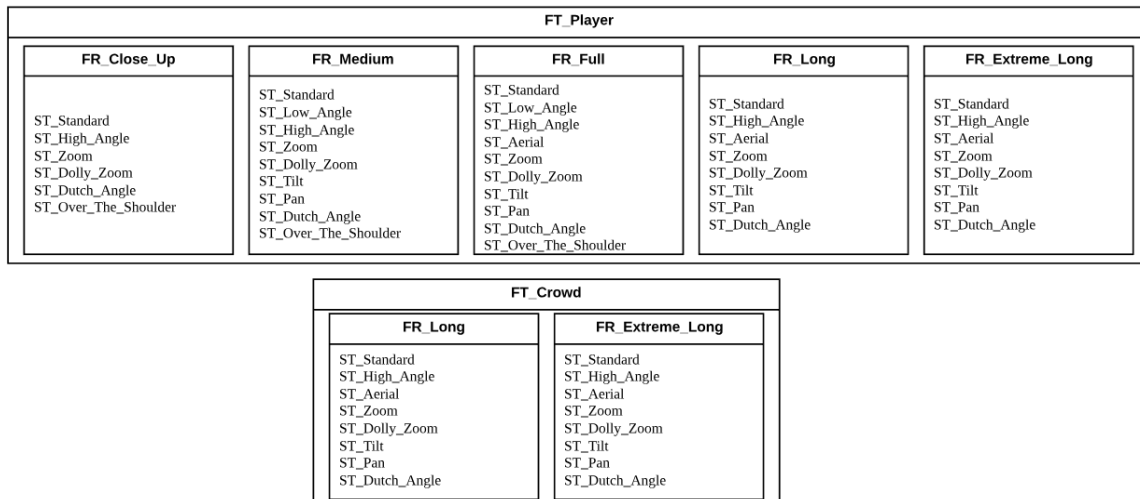


Fig. 17. Shot type hierarchical classifications. Each shot type is given a unique weight. (FT = Focus Type, FR = Framing Type, ST = Shot Type)

The focus type determines the main focus point of the camera. In the case of our game, this is constrained to the player or the crowd. These specifications are kept general allowing for later specifications of sub focal points within those categories that don't require the unique hierarchical structure that determines a shot type. Depending on the type of shot selected, the director will obtain a reference to a constant vector, actor transform, or bone reference. The constant transform is useful for maintaining a single location in space, typically used when no actor occupies the location of the focus. Because the director has no sense of the environment these

locations would have to be predefined or calculated based on the location of other actors, limiting its possibilities. The actor transform is useful for easily referencing a location; speaker, stage, crowd member or even player. This type of shot is very useful for pan shots that might move focus from one object to another. During gameplay an animated character’s skeleton will move away from the root transform, forcing the use of bone references for any dynamic objects. The bone reference is the most commonly used focus point as it allows the camera to track the players movement throughout play. The focus is typically in the head for closeups and around the chest to waist for most other shots. Bone tracking showed many jarring quick movements, especially any vertical tacking movement. To prevent this, we use varying lerp speeds depending on the framing distance helping to create a smooth follow motion. Additionally, vertical tracking was eliminated from anything beyond a closeup shot as it wasn’t typically necessary and only caused discomfort to the viewer.

Each focus type has a unique selection of framing types. The framing type defines the amount of the target is currently on the screen based on common cinematography terminology. Typically, the framing type is classified as a shot type similar to the layer below framing type. However, we decided to separate out framing types to be more precise when specifying location and focal length. Additionally, this allows for combinations of framing types and other shot types that otherwise wouldn’t be possible. Each framing type contains specifications for the range of vertical field of view distance used to specify how much of the target is on the screen. Rather than determine this amount at runtime, we measured specific ranges that correspond to the cinematic definition of a shot as seen in Table 6. In editor we measured our standard character model for those minimum and maximum sizes to create the vertical FOV range. These measurements are important when calculating a focal length and the distance combination that provides a specific framing.

Framing Type	VFOV Range	Definition
Close-up	15-40	Minimum: Top of eyes down to bottom of mouth Maximum: Top of head down to shirt pocket
Medium	40-130	Minimum: Slightly above head to chest Maximum: Above head to waist level
Full	130-225	Manimum: Above head down to knees Maximum: Above head to below feet
Long	225-500	Minimum: Full character maintaining 90% screen space Maximum: Full character maintaining 70% screen space
Extreme Long	500-2000	Minimum: Full character maintaining 70% screen space Maximum: Full character maintaining 20% screen space

Table 6. Framing type definitions and vertical field of view ranges in unreal engine units.

The shot type is the most unique component of a shot as it can describe many different aspects of a shot. Following historical cinematic shot types, we include 10 different possible types for the AI to select from as outlined below. Each shot type contains its own set of parameters that allow for the randomized creation of that shot through the use of custom keyframe components. Below is a brief outline of the unique components of each shot type:

Standard. The standard shot is the most common and easiest shot to implement. The possible locations for this type of shot is a 45° in any direction from directly in front of the target, giving a pretty basic area for positioning the shot. The movement will remain within the given framing type and the specified locations from the location sphere. We expect to see this type of shot perform very well among the audience given its simplistic design.

Low Angle. The low angle shot is commonly known for giving the target a sense of power. In respect to our game this might be useful for giving the player character a larger than life feel as well as making the audience feel like they are seeing the perspective of the crowd. This type of shot is locked to forward shots from a low angle. Movement for a low angle shot follows the same design as a standard shot does. Low angle shots of the player seem pretty desirable however, shots of the crowd would seem out of place, so the low angle shot will never appear for a crowd focus.

High Angle. The high angle shot replicates a low angle, except from above instead. Where in typical cinematography this depicts the character as being weak, we intend to use it to give the view a typical concert video camera would provide. Many shots that are seen in the high angle shot can be reduced to a standard shot and so it may prove to be less unique than intended. We expect to see high angle to be popular at longer distances where it provides more to view in scene, whether this is stage props or crowd members.

Aerial. The aerial shot takes the high angle to a further extent by providing a bird's eye view of the target. This again will do an excellent job of replicating an actual concert camera that is hanging from the ceiling. Because of the location of this type of shot, the location is kept constant to help avoid any weird tracking issues caused by spinning around the players vertical axis. The aerial shot only works effectively at a distance and as such will only be seen at a framing type of full shot and beyond.

Zoom. The zoom shot provides for interesting movement to occur, switching from standard movement to movement between framing types. A zoom shot will move from one framing type to at least one and at max 2 framing types above or below it. The effects of a zoom shot allow for a delightful change in movement patterns however if used excessively, their effects wear down, which is why a zoom shot might carry a high desirability rating while maintaining a lower cinematic probability. To keep simplicity of the system, a zoom shot operates within the same location range of a standard shot, however will always move in a straight line towards the target.

Dolly Zoom. Much like the standard zoom, the dolly zoom provides a drastically different cinematic perspective. The dolly zoom creates a sense of distortion for the viewer and when used rapidly can be quite jarring as often it is intended. In our game we want to use the dolly zoom to add an additional effect for viewers but in a smooth manner, forcing this shot to take a longer amount of time or occur over a shorter distance. When a dolly zoom occurs, the distance to the target changes while the focal length will change in the opposite direction. This effect makes the apparent distance of the foreground and background to change. The dolly zoom keeps all the same functionality of a standard zoom, except with the change in focal length.

Tilt. A tilt shot moves the camera either vertically up or vertically down, typically changing its target focus from beginning to end. In the context of our game the tilt shot provides a great way to switch to and from crowd and player focuses. Due to the nature of the shot, it must begin at a high angle directional position and end at a mid to low angle position or vice versa. Alternatively, the tilt can keep a constant position and shift its focus from one actor to another.

Pan. Very similar to the tilt is the pan, however the pan moves horizontally rather than vertically. As expected, instead of going from high angle to low angle, the pan shot will move its focus from left to right. The pan while extremely similar to the tilt shot has a smoother feel to it due to the limited vertical movement. Due to this, we expect the pan shot to have a higher cinematic rating than a typical tilt shot.

Dutch Angle. The Dutch angle shot is a fairly simple twist on the standard shot that has a powerful effect. By tilting the camera on its roll axis, this shot type conveys the message of tension. A common shot angle for horror films to depict unrest or psychological issues, this shot will have an interesting effect on the concert

style cinematography. While we don’t expect the Dutch angle shot to be extremely popular, we do see it to have potential for becoming more than a message of unrest.

Over the Shoulder. The over the shoulder shot is the only shot type that is deliberately placed behind the target. To provide this specific type of shot without hardcoding a location on every character, we constrain the location behind the player and offset it to the shoulder. First the shot must be a forward shot at a mid to high angle. Next the focus is set to the character so that it looks forward before the offset. Then finally the offset moves the camera to behind the player’s shoulder depending on the distance from the target. An over the shoulder shot should always maintain constant and never move due to the very small available locations that maintain the definition of over the shoulder.

6.2.2 Additional Attributes. A shot can be defined cinematically by the core components, however there are many different smaller details of a shot that need to be considered when attempting to learn what makes a shot good. While most of these additional components can easily be determined at random, we choose to place them in defined ranges in order to give the camera director better control over a shot composition and to increase some of the combinatorial diversity.

The first attribute to consider is the shot length, or the overall time of a single shot including all possible keyframes. This is a great example of an attribute that could be fixed or even selected at random, however we outline a number of categorical ranges to select from. By doing so we can later use the range to associate a length of a shot with another shot component. For example, a zoom shot looks far better when done slow and smooth rather than a drastic 2 second fly in zoom. Attributes like shot length with a provided range will have a fully random number produced within the selected range to determine the actual value for a given shot. Here we must balance between having fine grained control, and meaningful ranges. To do so we break down the shot length into 5 ranges seen in Table 7.

Classification	Shot Length (Seconds)
Extreme Short	2 - 4
Short	4 - 6
Medium	6- 8
Long	8 - 10
Extreme Long	10 - 12

Table 7. Shot Length

The next attribute is focal length. The focal length is extremely important to the cinematics of a shot and can easily distort a good shot to become bad. In many unreal engine cinematic approaches, the camera is left with its fixed 35mm lens, however that doesn’t fit the cinematic context of our game. We leave the focal length choices up to the camera director with the assumption that after some time the range containing 35mm will eventually have the highest utility score. If we look to real world examples however, movies will commonly see 35mm or less but for a concert scene we find the use of telephoto lens to be common due to the distance from the stage. The use of these ranges not only provides greater diversity but also an attempt at authenticity for a concert style video. The focal length follows the same mechanics as a shot length does in terms of selection. The 7 commonly used ranges of focal length can be seen in Table 8.

Velocity type is one of the simplest attributes, determining the style of movement a shot will use between keyframes. At the current state of the project, the system can choose between three possible movement styles. First, constant velocity will maintain a linear movement pattern between each keyframe, simple yet very effective.

The second approach will use a 3 keyframe approach, starting, midpoint, and endpoint. The velocity will ease in to the midpoint then out of the midpoint, creating a smooth movement from start to end. The final style combines the first two by using a 4 keyframe approach and easing into the second, holding velocity to the third the easing out the velocity to the final keyframe. In the event the shot shouldn't even move, this isn't determined by velocity type but rather in the movement calculations, so instead of including an option for constant movement, we define constant velocity. If a shot has the same start and end, then the velocity type will simply be disregarded as no movement will occur.

The weighted sphere is the most unique component of all the utility scores because it defines valid locations to place the camera rather than a type or category. When the system selects a distance, the possible locations can be defined on a sphere drawn around the player with the radius of the selected distance. Depending on the shot type, different cameras must be placed at different heights, angle, or positions and to do this we define locations on the sphere that are valid to a particular shot type. Each location sphere contains two ranges, the theta and phi of the spherical coordinates that define a valid range. The most common shot, a standard shot range can be seen below in Figure 18.

$$\theta = \left[-\frac{1}{4}\pi, \frac{1}{4}\pi \right] \quad \phi = \left[\frac{1}{4}\pi, \frac{3}{4}\pi \right]$$

Fig. 18. Standard shots location sphere range: left is theta range, right is phi range.

With a defined range on the sphere the camera director can determine where it can place a camera but has no information on where the best place is to place a camera. To provide the director with this information we divide the valid range of the location sphere into 9, 3 by 3, subsections. Each subsection contains its own range to select from as well as the typical attributes of a utility function. Each set of subsections is unique to the location sphere which itself is independent to a static shot type. Here we say static shot type because we use the same location sphere regardless of if the framing type or focus type is different. Keeping the location sphere static provides the learning functions more control over the system by providing less options to explore. Additionally, this will also help to save memory by not having duplicated location spheres scattered into the program and configuration files.

6.3 Selecting a Shot

Shot selection makes use of this utility theory to describe how valuable a component of the next shot is in the current situation. Each shot component can be described with a single number, its expected utility score, however this number is derived from several attributes of varying value. The most valuable of these attributes is the shot rating. Based on human voting, we update a count of positive and negative votes and later average it to describe the popularity of a component. By averaging this count, we aren't limiting or valuing the vote count of a shot in any way. This is extremely important because it gives each vote the same value rather than having each vote winner, positive or negative, have the same value. The difference is that if one-person votes in a single pole, it should have a very different effect than if 5000 people vote. Additionally, using an average will provide a number that clamps itself as it converges to its true utility, rather than having to use discounts to provide less value to votes over time.

The second most important attribute of a component's utility function is its cinematic base score. This is a score that does not move and is preset based on research and testing to approximate what a cinematic sequence

of this genre would expect to see. The score essentially provides our system with a seeding that it can grow from. Furthermore, this base line number will help to prevent user voting from going too far into the extremes and keeping the viewing experience grounded in some cinematic history.

Moving from a stateless model of cinematics, this has the potential to move into full state-based approach by basing the next actions based on the previous set of components, creating the full reinforcement learning loop. The first approach to this is to make a separate set of potential components based on the previous set but this would take far too many iterations to improve the learning model and tune the utility scores. The second, much faster approach, would be to add another attribute to the utility function of each component that changes based on the previous components. For example, the tilt shot, vertical movement, might have a low attribute score if the previous shot type was a pan, horizontal movement. This is simply because in cinematics you would not typically see this type of shot. While similar to a cooldown function this helps to bring out more of the historical cinematics into the sequence rather than just in a single shot.

In order to better understand the effects of the human in the loop interaction, we simplified the utility function down to the base line cinematic score and the voting average. Using these scores, we can better understand how the audience of the twitch stream chooses to control the cinematics and see if what they produce, does converge to accurate utility scores. To do so we give the voting system a heavy bias but keep a constant value given to the baseline score seen in Figure 19.

$$EU = 0.25C + 0.75V$$

Fig. 19. Utility function of any given component, where EU is the expected utility, C is the cinematic base score, and V is the voting average.

The first step to selecting a shot is to calculate the utility scores of all the possible components involved. Because of the hierarchical structure of the shot definition seen earlier in Figure 17 the calculations must begin from the top and move down as the components are selected. This means selecting the focus type followed by the framing type and shot type. Once the system has determined the utility scores of all components in consideration it will select the next selected action using a weighted random function. We use the weighted random approach due to the variety of shots it will provide. The Epsilon-Greedy way of selecting the next action works well for the situation of selecting the absolute best and still exploring the other options however our camera director needs to do more than look for the best. To provide some cinematic variety the weighted random can pick anything from the best shot to the worst shot and even the same style of shot in succession. The variety will blur together to appear to a viewer as a thought-out cinematic sequence.

6.4 Creating the Framing

Once the core components are selected, primarily the type of framing, the AI must decide on several other aspects involved in a shot composition. Most importantly it must find a valid combination of distance from the target and focal length that accurately fit the horizontal field of view constraints outlined by the framing type. Realistically, the camera can be positioned anywhere in the scene, so distance can be any real number. The focal length however can drastically alter the shot and needs to be properly selected. For this reason, the distance is calculated as a function of focal length and vertical field of view.

Due to the importance of proper focal length, we separated the possibilities into common ranges [2]. Each range will carry its own utility rate that functions in the same way as the shot selection, however it is unique only to a framing type. This means there will be five different sets of focal length weights, one for each framing type. The camera director will select a range from Table 8 then decide on a random integer within the range to

be the focal length of the camera. The focal length in engine will actually be classified as an angular field of view that can be assigned to the camera. The conversion from focal length to field of view is done using the 35mm equivalency [51].

Classification	Focal Length	Field of View
Ultra-Wide Angle	12-22 mm	122-89°
Wide-Angle	23-40 mm	88-57°
Standard	41-60 mm	56-40°
Short Telephoto	61-80 mm	39-30°
Medium Telephoto	81-135 mm	29-18°
Long Telephoto	135-300 mm	17-8°
Ultra-Telephoto	300-600 mm	7-4°

Table 8. Focal length and vertical field of view classifications.

With a focal length selected we can calculate the vertical field of view angle using equation A in Figure 20 [14]. The calculation is dependent on the height of the camera's sensor which we keep at a default height of 20.25 and width of 36 units. Based on the framing type we also can obtain the vertical field of view distance using Table 6. With both the vertical field of view angle and distance we can create a right triangle from the camera's sensor to the target and solve for the distance. The equation for working distance can be seen in equation B of Figure 20 and an example of the scene can be seen in part C. With all three of these components determined, we have successfully created the specified framing with a given focal length.

From there the field of view and distance can be passed forward to be used to setup the individual keyframes of the shot composition. Because some shots will be moving the camera, the distance will be constantly changing, possibly altering the framing of shot and producing undesired effects. All keyframes of the shot should check the distance from the given location to the starting target location to make sure it stays within the bounds of the horizontal field of view. Additionally, during this distance verification process, the camera director will also cast rays to the target and back to the camera to confirm it has the proper line of sight of the target. In some cases, the camera may have line of sight of the desired bone but not the rest of the body. At the moment there's no good way to account for this possibility and it may skew votes if it occurs, but it should be quite rare based on the environment.

In the event the camera fails any validation checks, obtaining line of sight, conforming to framing restraints, valid movement distance, etc., the camera director will jump back up to before the failed location or other data was selected and rerun the code hoping for better results. In some cases, no desired results can be found, in which case after a select number of failed attempts, it will move back an additional step in the selection process and reattempt it from there. This process continues until the entire camera selection process starts over again. At that point if the camera director can't find a valid camera, something is wrong on the programming side and the system will exit with an error. Typically, the camera manager will record 0-3 failed attempts per camera generation, mostly due to spawning the camera behind a wall and failing a line of sight check. This is normal and has no effect on the system's ability to process everything effectively.

6.5 Positioning the Camera

With a distance determined the possible locations of the camera can now be reduced to a sphere around the player with a radius of the given distance. This is the location sphere as described earlier in the additional attributes section. By referencing the static class of the selected shot type, we can obtain the proper location sphere needed

$$VerticalAngle = 2 \cdot \arctan\left(\frac{SensorDimension}{2 \cdot FocalLength}\right) \quad \mathbf{A}$$

$$WorkingDistance = \frac{\frac{1}{2}VerticalDistance}{\tan\left(\frac{1}{2}VerticalAngle\right)} \quad \mathbf{B}$$

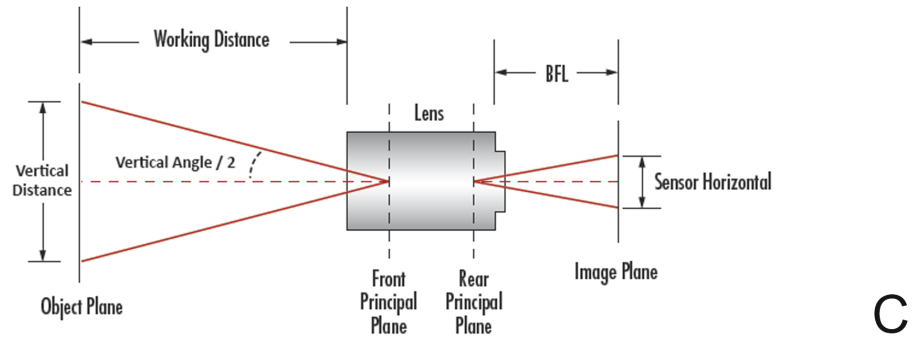


Fig. 20. A: Equation of vertical field of view angle given a sensor dimension (height) and a focal length. B: Equation of working distance given the vertical field of view distance and vertical field of view angle. C: Depiction of the scene, where the focused target would be on the right and the center is the camera’s sensor creating a solvable right triangle.

for a shot. From there we can select the subsection to use how we would any other utility scored component. From there we can obtain the specific range for theta and phi, the spherical coordinates used to determine the location on the location sphere creating an area to select the location from seen in Figure 21. Because the subsections range is already specific enough we can determine the final locations by random in the given ranges.

From spherical coordinates we can convert into cartesian coordinates which now will refer to the location of the camera with the target as the origin, however a camera is not a child of the target. One final conversion is required from target space to world space and we have a finalized location for the camera using a look-at function towards the focus point as the rotation. This location can be assigned to the starting keyframe and the process can begin again for the next keyframe in the shot. To make things easy, we only calculate the starting and ending positions of a shot and do a linear interpolation between the two points. The velocity of this transition is calculated by the velocity type to keep things looking smooth. The ending keyframes location must be 15% or more the distance to the target in distance from the starting keyframe. This allows for a baseline amount of movement to occur while keeping things randomized.

6.6 The Camera Queue

Once a camera has been successfully created it can be added to the camera queue. The camera queue is an array of cameras waiting to be used in the cinematic sequence. Because cameras can take a moment to create or may need to fade from one to another, there should always be more cameras to switch to. The one issue with precomputing the cameras and adding them to a queue is that all locations and calculations will be based on the time they were

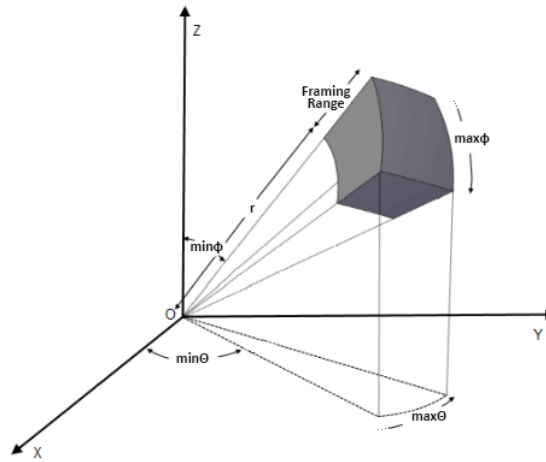


Fig. 21. Representation of location sphere subsection determined using spherical coordinate ranges.

created. This issue isn't a huge problem seeing as the player in this context doesn't move very far and the focus point is updated and tracked in real time. The camera queue will always maintain 3 cameras ready for use. This means when viewing a camera, its location was made based on the environment 3 shots or about 20 seconds prior.

When a camera has reached its final keyframe, the director will remove it from the active camera slot and move it to the post processing queue. Whenever the active camera slot is empty, the director will pop a camera of the camera queue and place it into the active slot, continuing the loop and adding to the post processing queue. The post processing queue is where a finished camera waits for its scoring information. Because streams are delayed, and people need time to make decisions, we allow for a delay in time between camera completion and utility reward calculation. Once the timer on a camera's vote delay has ended it will call the twitch instance and request the vote counts and use it to update its utility scores.

The camera director will always start by adding 3 cameras to fill up the camera queue then sit idle until a camera has finished or needs votes counted. Due to the delay, some voting will not be accounted for until the song is over, and the player has left the scene. To account for this, the camera director will remain as part of a game instance, persisting between scenes whereas normal actors will be purged between scenes. Cameras will continue to wait for votes and can count votes even during the next song. Once votes are counted the camera director itself will update the utility scores of each camera using a reinforcement learning methodology.

6.7 Reinforcement Learning and the Multi-armed Bandit

Based on human in the loop interaction, we use chat messages from a live stream view of the camera cinematics to rate the performance of each shot. Based on stream viewer input, a single shot can be rated as either good or bad, but all votes are counted and stored separately. Using reinforcement learning, after a slight delay, we can give rewards to the camera and its used components allowing for refinement of the utility score and better selection of cinematic shots.

Reinforcement learning is the process of teaching an intelligent agent what to do or what action to take when there is no information or example for it to learn by [42]. This type of learning is often compared to trial and error. Once an agent has completed an action and the state of the environment has changed, we can evaluate the

decision and inform the agent if it is positive or negative. In some cases, this is not known after each action but only after all actions have been completed.

The simplest form of learning is value iteration. In this case there isn’t even an agent but rather a series of sequential decisions to be made [7]. Each state is given a random initial utility and using the Bellman equation, the states expected utility can be recalculated based on the utility of its neighbors. Because the reward of each state is known, later states selected can affect the current states utility, providing this relationship. By doing multiple sweeps throughout the set of states, the utility of each state will eventually converge on the actual expected utility. In this situation, the rewards of each state must be known in order for the system to properly update the utility function. Because our camera director has no knowledge of the rewards at the time of action selection, value iteration is not a sufficient model however it is very similar to our approach.

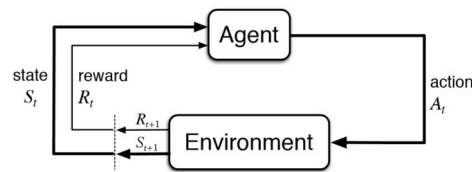


Fig. 22. The typical full reinforcement learning loop.[49]

Without prior knowledge of the reward in each state or from each action, the algorithm must use the reinforcement learning loop seen in Figure 22. Here the agent must select an action then update its state based on the selected action and resulting reward. The two general types of reinforcement learning are passive and active learning [42]. Passive learning occurs when the actions of each state are predetermined or in other words, the policy is fixed. This can be useful for discovering the environment and the given utility function of a state, however is not useful for learning about its actions or what to do. Active learning helps an agent to know what to do in this unknown environment, by altering its actions and policy for each state, learning from each attempt. By using value iteration or policy iteration, we can obtain the utility function of a state or action and determine what the next best action is. Rather than exploiting the best possible action on every iteration, we may want to select a different action with a lesser known utility providing for better exploration. Creating the balance between exploration and exploitation is a fundamental concept of reinforcement learning. Active learning will typically use Q-learning to maximize state action pairs while maintaining this balance.

The camera system in our game can be modeled extremely close to active learning. We can define a state to be the current set of utility scores, both cinematic and voting attributes. The action is represented by the selection process, which shot components to select next. The reward is found after each shot, so there’s no delay caused by scoring occurring only at the end of a song, however there is a given time delay due to streaming. The important part that makes this model into reinforcement learning is if the action directly effects the state. In our case, the action only partial effects the state, but more the reward updates the state. An action that updates the state would be like moving from one room to another where you now have a fully updates set of actions. In our case we have a single state with varying values creating a special case of reinforcement learning.

In the general reinforcement learning case, there are states and actions, however some special cases only have one state with multiple actions that is repeated with different results [17]. This case is the multi-armed bandit, named after the one-armed bandit referring to slot machines. The proposed issue is if you have multiple slot levers to pull, each with their own unknown payout, which one do you pull and how many times. In reference to learning, we would want to start by exploring the different levers to see which is good and then begin to exploit the one with the best overall payout. The most common way to find the best paying lever, is to use

Epsilon-Greedy. Epsilon-Greedy will assign a probability of $1-\epsilon$ to the current best paying arm and ϵ to any other arm as seen in Figure 23. Epsilon now correlates positively to exploration and negatively to exploitation, however in most cases is assigned to 0.1.

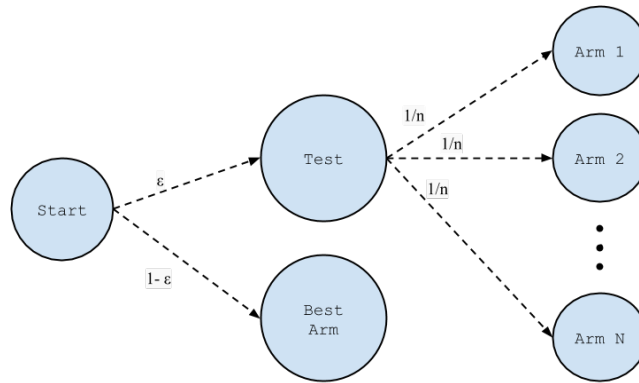


Fig. 23. The epsilon-greedy decision tree. [4]

While most bandit situations are based around a stateless environment, the contextual bandit uses states to define preexisting information that isn't directly related to the state before it as seen in Figure 24 [18]. In the example of slots, this could be information on the number of times a machine has been played and how many times it has payed out that day. An unknowing agent may assume if it's been played a lot and payed out very little, it must be a bad machine, however after later exploration, this is actually often associated with being primed to pay out next. In a contextual bandit this information is typically different every time and unrelated to the previous state. In this example, it is more of the full reinforcement learning problem, to be a contextual bandit, the state should be independent of whatever action is taken. The information might be randomized after each poll rather than updated based on the last pull creating the contextual bandit.

The contextual bandit is the best description of our camera system model as it has only one state and one set of actions, however varying contexts that can be updated by actions. Here our camera manager can select the same shot components each time but only has the score as the given context information. After each shot has completed, the selected shot components have any twitch votes added to their counters, altering their vote average and effecting their overall utility score. If we add a new attribute that selects the next shot based on the previous, this would move the system to the full reinforcement learning loop. For example, if the last shot was a closeup so the next shot can only select from long and extreme long shots, then we have the full loop.

Each shot is scored collectively although there are multiple components creating a shot. We have already described how this can be summarized as combinational variety, however another possibility is the exact opposite, concluding what component was the main factor in the results. We know what all other shot components scores are when we select a component so if we add those scores together we can determine what effect the currently observed attribute has. If when a component is selected, all other shot components have low scores, say 17 out of 100 average, and the shot is rated well, we can likely conclude that the current shot component was a large contributing factor in the shots performance. This is another attribute that can be added to the utility score or can be an input variable into the voting attributes value function.

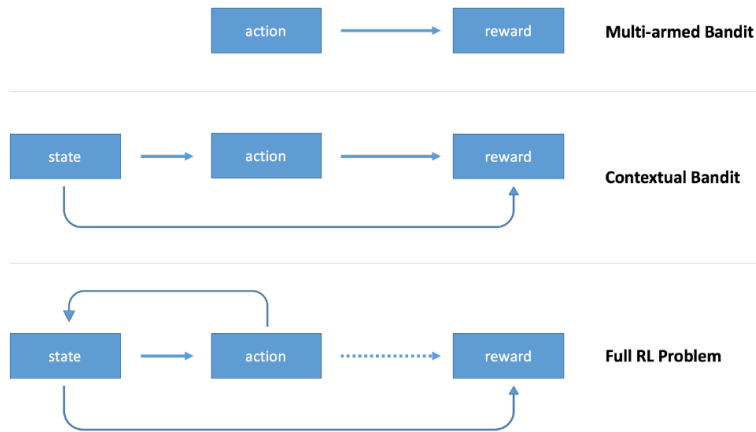


Fig. 24. Above: Multi-armed bandit problem, where only action effect reward. Middle: Contextual bandit problem, where state and action effect reward. Bottom: Full RL problem, where action effects state, and rewards may be delayed in time. [18]

7 TESTING PROCESS

The testing process for *Vocaodoru* was divided into three phases, each designed to test a different element of the game and the experience goals. Before development began, we ran a series of paper prototype tests in order to determine if our game idea was feasible and interested players. After our final implementation was completed, we tested both elements of the game separately; on-site testers participated in gameplay testing by playing through a vertical slice of gameplay, while off-site testers participated as the stream viewer and interacted with the cinematography AI. Testing was performed separately because both sections of the game have vastly different experience goals, and the logistics of testing both at the same time would have required much more oversight and approval to complete.

7.1 Paper Prototype

The initial phase of design for our project included a paper prototype to test feasibility of the core gameplay mechanics and the flipbook-style UI. Paper prototyping is a quick and easy method to proof game design without requiring a significant amount of implementation. The prototype consisted of a flipbook of Hatsune Miku performing a dance routine featuring indicators where players hands should be at the end of each sequence. The flipbook represented our original UI design, with each intermediate pose being a static image rather than a smooth animation. The prototype walked players through a very truncated dance animation and provided a small vertical slice of our gameplay.

Feedback from testing using our paper prototype informed the design process as we moved from analog to digital. First, we had to tackle the problem of indicating clearly to the player which side of their body they had to move at a given time. We decided that a color-based system, while not perfect for all players, provided the simplest method for tracking which hand was which. The prototype did not, however, test the grid-based gameplay that was originally planned, nor did it solve the problem that lied in full-body tracking in VR that we had to tackle. Our proposed solution after our prototype was to use a single chest-mounted tracker to get the player’s position. We wanted to use a grid-based system to have the player move around their play area while doing the dance, but a UI for this was never tested as this system was cut before final implementation.

Player responses to the paper prototype varied, but players almost unanimously agreed that the gameplay in and of itself was interesting enough to justify the further development of the project. Initial tests were performed using a hand-held flipbook version of the paper prototype, which led to human error and inconsistent tester experiences. We revised this to use a digital, .gif based prototype which ran at a consistent framerate and was unobscured the the administrator's hands. Responses to the digital "paper prototype" were significantly more favorable, with all playtesters expressing interest in the final product and offering suggestions about how to better communicate hand positions. We implemented some of these ideas into the final game, including encompassing the hands in a solid color rather than just an outline.

7.2 Gameplay Testing

Gameplay testing took place over the course of two days at Ritsumeikan University's Biwako Kusatsu Campus as well as the main campus of Osaka University. Nine playtesters were invited to play our game, resulting in a dataset containing usable data from 8 playtesters that will be analyzed in this section. First, we will discuss the testing procedure and methodology used to acquire our results, and why we were unable to use all the testing data we acquired. Next, we will analyze the qualitative data acquired from the survey provided to all testers to gauge player feedback. Finally, we will perform a what-if analysis on gameplay data captured from the game to determine improvements that can be made to the judgement system.

The tests were conducted in-person, with at least one member of the project team facilitating the playtest proper, with another member of the team administering the survey at the end of the playtest. Players were first given instructions about the calibration process and gameplay procedure before they were given the HTC Vive and motion controllers. Players were also walked through the calibration process step-by-step by the facilitator before the gameplay test began, to ensure the calibration process worked properly. Players were then instructed to play through the test song in full, ensuring they played to the best of their ability, and upon conclusion of the song were directed over to the survey station while another playtester began the facilitation process.

Players had two sets of data collected after each playtest- qualitative survey data and quantitative gameplay data, a sort of "replay" file. Players survey results and replay data were given a unique ID and reordered to ensure that they could be associated, without identifying information being included during analysis. Replay data was scoured over by hand to ensure that players really were attempting to play the game before data was analyzed further. During this process, we encountered one player who did not move more than 2 inches in any given direction for over 20 judgement ticks (~ 12 seconds), and as a result of this discovery their results were excluded from the data analysis.

The survey consisted of a series of 28 questions, separated into areas that discussed *Vocaodoru* in a broad scope as well as areas that discussed specific areas of design. Players were first asked question about *Vocaodoru*, their play experience, and if they would use the game in the future. We found that there was a 50/50 split amongst players as to their attitudes on continuing to play the game, with no correlation to their level of performance during gameplay. Players also reported that the controls were easy to use (and that they believed they'd be easy to learn over time), and players also believed the judgement system was consistent on the whole. Players reported that their main reason for playing *Vocaodoru* would be casual entertainment, but every player believed there needed to be an in-depth tutorial before the game would be ready for public release.

Players were subjected to a series of questions about the UI and pose generation system, and how easy they found the system to "read". Players were split evenly, with half finding the system easy to use and the other half finding it difficult. Players were also evenly split on the pose scrolling speed, indicating that the best solution may be to use a customizable UI. Some games, such as *osu!* and *Stepmania*, allow players to determine their own scroll speed on a per-song basis, and this level of customization will assist players in finding settings that work for them. Players were split on which guide they used to follow the dance patterns, which is in keeping with our

design goals. We did find that some users wanted to see more than one pose scrolling at a time, but we believe this may be too confusing for newer players who already have trouble focusing on more than one guide at once.

Players were surveyed on the art direction of our limited test stage and provided feedback on the art assets that were implemented. Perhaps unsurprisingly, the character models were rated across the board as the best art assets in the game, while the stage and background were rated the lowest. This was due to the final stage not being implemented in the test version due to time constraints. Players also noted that lighting, particle effects, and other special effects needed improvement before they were ready for a final release. Players directly requested changes to the particle effects used for the judgement system, finding them hard to read, and also asked for the implementation of more characters than were previously available.

Being that *Vocaodoru* is a rhythm game, an extra amount of emphasis was placed on the audio in our survey. Players were asked about their opinions on the music that was in the game, the music they’d like to see added, and their opinion on auditory feedback, which was not yet implemented. Players were of a mixed opinion on the included testing song (*No title* by Reol), which is to be expected; our testers were primarily American, while the test song was entirely in Japanese. Players across the board asked for more variety in the song selection for *Vocaodoru*, as currently only J-Pop and K-Pop are represented in our song choices. Players asked for more Vocaloid songs, both specifically (*Senbonzakura*, *Last Dance*) and in general (players asked for Kagamine Rin songs to be added) as well as a general request for American music.

Auditory feedback was another point of contention for players, as many felt that adding auditory feedback would have helped them play the game better. During the testing period, there were no feedback sounds associated with gameplay and players would receive only visual feedback from the game. We found that players wanted some sort of hitsounding, with distinct sounds available for at least “Fantastic” and “Abysmal” judgements. This is consistent with other popular rhythm games, with the best and worse judgements providing the most distinctive auditory feedback. One player also asked for a click-track or metronome to be added to the song to help players with less dance experience determine where the beat falls.

In the final section, players were asked about the gameplay feel and the perceived accuracy of the judgement system. Players found that it was mostly clear when they missed a pose in the judgement system, but did note that they felt like they missed poses that they actually hit more often than we would have liked. This was a known issue that we hoped to alleviate with further analysis after gameplay testing. Players also stated that more feedback about combo length or a scoring system should be implemented and displayed to players to better quantify player skill and allow for direct comparisons between players. Players also provided some suggestions for the scoring system, most notably to take aspects of speed from Beat Saber, but we feel that many of these would be counterintuitive to the nature of a dancing game.

The judgement system was under the most scrutiny during gameplay testing as this was a novel system that had little to no testing before other than to gauge the general accuracy of the system. As a result, our original numbers for judgements were based off of naught more than educated guesses. One of our primary goals in gameplay testing was to capture data from regular players and use this to rebuild the judgement system in a spreadsheet for further testing. Our log system in *Vocaodoru* captures data at each judgement about the component error of each player’s arm as well as the judgement passed to the player, allowing us to make changes and tweaks instantly without requiring a full testing setup. We captured approximately 10,000 points of data from 9 playtest sessions, allowing us to reconstruct each session inside of a new judgement environment.

The original judgement system determined a player’s judgement based off of their highest component error for either hand. Our what-if analysis started by tweaking this method of passing judgement rather than tweaking any of the numbers. We found that by instead requiring at least two components to be off by a certain amount, we could improve the accuracy of the judgement system and provide players with better feedback. Our average miss or “abysmal” count dropped from over 50 to barely inside the double digits, while we also saw a large increase in the number of great and good ratings. This did not fix the number of “fantastic” judgements, however, and

may still leave players feeling as if they have not been judged fairly. We have also tweaked the system to allow for a "fantastic" judgement if a player has at least 2 components in the "fantastic" range, and no more than 2 components in the "good" range. This new judgement system is waiting to be implemented for the next round of user testing, to be performed outside of the scope of the WPI MQP.

8 POST-MORTEM

8.1 What Did We Learn?

Vocaodoru was an eye-opening project for everyone involved, and was the team's first foray into the world of VR development. From the design phase, we learned a lot about different VR development environments and settled on Unreal Engine as our engine of choice. In our research, we learned about the pros and cons of the major engines for VR (namely Unreal and Unity) and settled on Unreal thanks to its more robust set of plugins to aid in development and ease of use with blueprints. We encountered many issues throughout the development process, but were quickly able to rectify all of them thanks to the litany of learning materials and Stack Overflow-esque websites available for the engine. As a result, we have all become much more well-versed in Unreal Engine and the standards and practices of VR development.

This project also served as an opportunity to hone our C++ development skills and focus on development outside of the standard libraries. For some, this was our first exposure to complex C++ development using extended libraries and required us to work on learning to read the API and to adapt to new best practices. Working in Unreal Engine also stressed how important it was to understand the underlying systems and how and when to instantiate variables, as the object lifecycle in Unreal is complex and unforgiving. This project also required us to make significant use of Unreal-specific object types and functions that we had not worked with before, such as the *TArray* container for sets of objects as well as the *UFunction* and *UPROPERTY* types for crafting blueprint-available methods and parameters.

This was also the first major team-based experience for members of our team, and served as a sort of trial by fire for the entire team. Working in a production environment on a team project with many interlocking elements is quite different from working on a solo project or a project with many discrete, independent elements. Being part of a team of computer science and game design students with wildly different skill sets and levels of expertise required us to learn to compromise and cooperate in a way that we hadn't really thought about in the past. This led to some harsh, eye-opening arguments over portions of the code but also helped us to develop as team members, programmers, and designers over the three-month development cycle of *Vocaodoru*.

8.2 What Went Right?

Vocaodoru was an ambitious project that pioneered a few different techniques for VR games that have not been seen before. The pose-based judgement system and cinematography AI implemented into *Vocaodoru* are first-of-their-kind developments and have been successful to varying degrees. To start with, the judgement system is an accurate measure that can determine where a player is positioned relative to a model that is easily calibrated to players of any size. This dynamic calibration system allows us to create one set of pose data that can then be used with any player without a need for significant preprocessing beyond the initial calibration that players would perform with any rhythm game. This system is something that should be explored for other VR games as it will provide a boon for differently-abled players and may be the first time that they would get to experience VR. The judgement system, however, is not perfect, and requires more feedback to the player about their arm position before it can be considered something that should be the standard for all games going forward.

The cinematography AI was a massive success by comparison, pioneering a new method for streaming VR games in a dynamic and interesting manner. Other games have provided an outside-the-body camera for streamers, such as Beat Saber in recent updates, but they frequently have issues such as poorly rigged or animated skeletons

for the player and usually stick to static or pre-programmed camera angles. Using pre-rigged skeletons and combining them with canned animation solves the first problem by giving us smoothly moving player characters. The AI itself, along with its ability to develop new camera angles and movement on the fly allows for a dynamic streaming experience. Early feedback has let us know that viewers really enjoyed the much more vibrant and lively camera system compared to just a static camera, and we believe that with further refinement we can improve this camera system even further.

Finally, we believe that we have delivered a complete project at the end of our development cycle that has tackled our major objectives to a satisfactory degree. Frequently, we were told that scope would be an issue for our project and that we would have to cut major features in order to finish on time. *Vocaodoru* managed to finish on-time within the 3 month development time with both major features implemented and merged into one project. We did have to cut minor features and additions such as tool development and some shot types from the AI, but thankfully we avoided cutting any major features that were tied to our experience goals. We feel that the project overall, despite its difficulties, was a great success.

8.3 What Went Wrong?

Vocaodoru suffered from a tumultuous development cycle marred by poor team dynamics and issues regarding compatibility of different features. The biggest issue that our team faced was an inability to communicate effectively and clearly our goals, standards, and current progress amongst the group. Frequent conflicts between group members started and escalated before fizzling out without being truly resolved, which resulted in the team being fairly fractured by the end of the project. Personal differences aside, the team also struggled to communicate in a professional manner with each other in a way that would stop tempers from flaring and egos from being hurt. Overall, the communication within the team and between the team and advisors needed a significant amount of work that couldn’t have happened in the short development time.

Beyond communication issues, there were frequent issues with unit testing not ensuring that a portion of the game was ready for final implementation into the production version of the game. Frequently, pieces of code that passed unit tests would fail in the production environment due to a difference in how unit tests operated compared to normal operation. An example of this would be the calibration system, which passed unit tests when the camera was fixed to a certain location, but began failing when the player changed their orientation. This required a significant rewrite and respecification of how the calibration process would work, but resulted in a stronger system in the end. The loss of time spent redeveloping systems like this resulted in a less polished final project that we feel could have been improved upon greatly.

The final problem that our game encountered were frequent delays due to factors outside of our control that resulted in long delays that no one could have foreseen. The art pipeline was a common instigator of these delays, with many “solutions” for creating .fbx files from MikuMikuDance assets either failing entirely or being too outdated for use with Unreal Engine. This meant that the creation of the first set of art assets used for testing our game took several weeks, a process that should have taken only an hour or two at most. While we were able to reduce the time taken on future art assets moving through the pipeline, this did not get back the time that we spent on our test assets. Beyond this, issues with Unreal Engine bugs and faults in other plugins (octahedral impostors, VRExpansion) held us back by several dozen manhours in terms of effective work. The sum of all these delays

8.4 Future Developments

From the outset, *Vocaodoru* was planned to be a project that would outlive the scope of the WPI MQP. As such, we have a set of planned future developments surrounding all facets of our game as we transition from the closed-source MQP development process towards an open-source, community maintained project. This

section will divide future developments into two broadly construed categories- gameplay improvements and AI improvements. A third, smaller, category refers primarily to polish and improvements to less major areas of development. We will cover polish-related developments alongside major area improvements that we plan on implementing as we see fit.

Gameplay currently is in a fairly rudimentary state and needs significant improvement to reach the robustness of other popular rhythm game platforms like *Stepmania* or *osu!*. One of our first and foremost areas that we wish to improve on is the lack of tools for content creation. Other rhythm games have survived for a long time thanks to a feature-complete set of tools to make creating custom content easy to do and easy to use. Finishing a file format that packs together all the necessary .uasset files and will easily extract to the gameplay directory for use in gameplay is our top priority upon *Vocaodoru*'s release into the open-source world.

Beyond content creation tools, an updated, rebalanced judgement system is another big point of contention for the team. Currently, the judgement system seems to harsh towards players, and while the what-if analysis has shored up some of these issues, we hope that a wider release will give us more feedback on the judgement system. We feel that an average player should be capable of scoring at least 80% on the fairly simple test dances that we have implemented, and are targeting this metric in our current rebalancing efforts. A large-scale survey will be implemented to gather feedback from a larger, more dedicated community of players in the future to gather more data. Players will also be encouraged to submit their play session logs to allow us to perform more analysis using their data.

Improving the current feedback system and gameplay art is also a major area that we hope to tackle in the future with the community. Our team did not feature any artists or anyone experienced enough with digital art to polish the game to a level of quality that we would like to achieve. As a result, we have plans to reach out to or hire artists to come work on a first pass of the art assets included in *Vocaodoru* and to help us improve the visual look and feel of the game. We want to improve our 3D environmental assets and texturing to help immerse the player more, as well as implement a moving crowd with an animated penlight effect. We also want to rework the UI and judgement particle effects system to make them have a clearer and more consistent visual design, in keeping with playtester feedback. Combined with a new audio feedback system, we believe that the player experience will improve greatly with a new design system.

Gameplay utility functions also require some changes to better fit the requirements of popular songs in game. The file format requires several changes to the scroll velocity (SV) system to allow for proper SV changes during gameplay, as well as the display of several poses at once. We also need to rework the time signature system to better support odd, non-standard, and irrational time signatures. Currently, irrational time signatures are impossible to implement and odd time signatures (such as 9/8 time) have anomalies with how important beats are selected by the pose generation system. We hope to find an algorithmic method to determine these poses, or barring this, to allow song creators to override the algorithm and determine the important beats themselves on a section-by-section basis. These changes to the utility functions will allow for more freedom in content creation and increase the variety of songs available for players and charters alike.

A IRB CONSENT FORM AND PLAYTEST SURVEY

Investigators: Aidan O'Keefe, David Giangrave, Patrick Critz
Contact Information: +1-860-986-2990 dlgiangrave@wpi.edu
Title of Research Study: Vocaloid VR Experience

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 user testing of Vocaloid VR Experience (project name subject to change) is to collect data on: a) user gameplay experience in a rhythm based virtual reality game based on Japanese singers, and how easy the game is to understand and play with and without instruction; b) balance, movement, and related health statistics based on movements and dance produced in gameplay, and comfort levels of a user in game; and c) viewing satisfaction levels of watching another player through an online stream and contributing to camera controls and viewing angles of all streaming users. Gathering data from users is the only way to understand how the game will be played and understood without supervision of the creators. The data collected from these testing sessions will be used to revise gameplay and user experience components of Vocaloid VR Experience.

Procedures

During the study the tester will be using an HTC Vive Virtual Reality Headset for approximately 3 minutes. The tester will not be using the headset for any longer than 10 minutes at a time. Afterwards they will answer several questions based on their experience.

Entering VR:

When ready testers will be instructed to step into the center of the testing area, an area without any obstructions allowing free movement and containing two HTC Vive base stations pointing towards the player. From there they will put on an HTC Vive head mounted display. Next, they will be handed two HTC Vive controllers that they will securely strap onto their wrists.

In VR:

At this stage the users will navigate the in-game menus to select a song they would like to demo. The song will begin, and the tester will be expected to perform a dance routine that is displayed in several segments through 3D images and replicated by a non-player character in game. When the song is complete the tester will inform the inspector that they are ready to leave the VR experience.

Exiting VR:

The inspector will then remove the Vive controllers from the testers hands. The tester will remove the headset and hand it to the inspector. The tester can now leave the test area and then fill out the related survey.

Survey:

The survey will be online and available to be taken from the user's personal device or from a provided tablet. The questions will ask simple questions about satisfaction and comfort levels.

Risks to study participants

Dizziness or Nausea:

While using the Virtual Reality Headset, users may experience dizziness or nausea. Should you experience these effects at any time, please notify the inspector and they will quickly and safely remove the head mounted display.

Falling or Tripping:

In rare cases, users may become so disoriented that they may fall over. In the event that a test subject begins to fall over, the nearest inspector will attempt to catch the user and help them into a safe position before removing the display.

The headset also contains series of cords that connect from the back of the user's head to the computer running the software. Should the user turn more than necessary they may become wrapped in the cords. The inspector will notify the tester if this happens and instruct them to remove cords before continuing.

Damage to surroundings:

While in VR the user should stay within the displayed bounding zone that will show up when reaching the extents of the testing area. Should the user leave this testing zone, they risk potentially damaging their surrounding and injuring themselves. Should they reach the bounds of the test area the inspector will instruct them to move back into the center of the testing area before continuing.

Discomfort:

When using the headset, users will be unable to see anything happening in the real world including their own body. This can cause unnecessary discomfort to the user. Should this happen they can ask to exit the testing process at any time. Additionally, if they wish, they can have their test conducted without anyone watching except the inspectors for spotting.

Benefits to research participants

There are no benefits to research participants.

Alternative procedures or treatments available to potential research participants

Research participants may request to have their testing session be conducted in a private setting. In this case the only people watching the test session will be the inspectors to assure the safety of the user. Under normal conditions the testing will be taking place in an open lab with other students coming and going.

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, the sponsor or it's designee 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.

For more information about this research or about the rights of research participants, or in case of research-related injury, contact: Researcher David Giangrave Tel. 860-986-2990, Email: dlgiangrave@wpi.edu, IRB Chair, Professor Kent Rissmiller, Tel. 508- 831-5019, Email: kjr@wpi.edu, and the Human Protection Administrator Gabriel Johnson, Tel. 508-831-4989, Email: gjohnson@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




Signature of person who explained this study




▼ Gameplay Mechanics

Q1.1  **Thank you for testing out the virtual reality rhythm experience. To improve the user experience of the game, please complete the following survey, answering all questions to the best of your ability. If you have any questions please ask one of the inspectors conducting your testing session.**

Q1.2  Please describe your overall game play experience below, selecting the option that best fits:


	Disagree	Somewhat Disagree	Somewhat Agree	Agree	Not Applicable or No Opinion
I would use this game frequently.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The controls were easy to use.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I think most people would learn this system very quickly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
There was a high amount of inconsistency in the system.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Q1.3  Did the tutorial sufficiently prepare you for the actual game play?

Yes

No

Skipped the tutorial

Q1.4  Why would you use a product like this? (Check all that apply)

Casual entertainment

Competitive play

Building skills

Relaxation

Waste time



Q2.1

User Interface

The following questions refer to the scrolling panel that displayed what move or position that you needed to match next.



Q2.2

Was this system easy to understand and follow along?

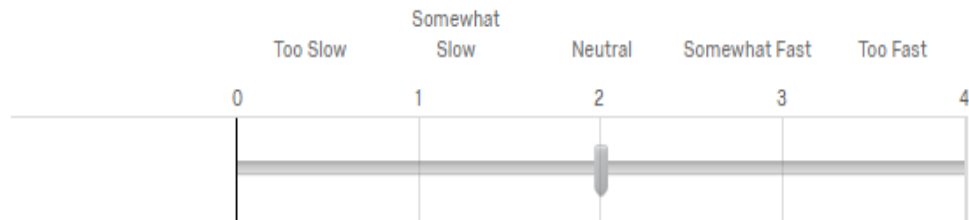


- Agree
- Somewhat agree
- Somewhat disagree
- Disagree
- Not Applicable or No Opinion



Q2.3

Please rate the speed of the poses being displayed.



Q2.4

Did you follow the 3D guide or the scrolling panel?



- 3D guide (Character displaying full dance)
- Scrolling panel (panel indicating future moves)
- A mixture of the two




Q2.5


Please provide any recommendations or additional comments specific to the user interface. **(Optional)**



Q3.1 **Graphics**



Q3.2 Please describe your graphical experience below:



	Low quality or broken	Needs Improvement	High quality
Character models	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Stage and environment	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Lighting	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Special effects	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Q3.3 Additional comments on graphics (**Optional**)



Q4.1 **Audio**

Q4.2 Please rate your audio experience below:

	Low quality or broken	Needs improvement	High quality
Selected song	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Hit or Miss sound effects	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Menu audio	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Q4.3 How did you feel about the song selection options?

- I found exactly what I wanted
- I like it but needs a few more options
- Needs a lot more songs and variety

Q4.4 What songs would you like to see added? **(Optional)**

Q4.5 Where would you like to see this game move to next? Example: K-Pop **(Optional)**

Q4.6 Additional comments on audio **(Optional)**

Q5.1

Scoring and Judgment System



Q5.2

Was it clear when you hit or miss a pose?



- Very clear
- Needs improvement
- Didn't understand

Q5.3

How often did you feel you hit a pose that the system said you missed?



- Never
- Sometimes
- About half the time
- Most of the time
- Always

Q5.4

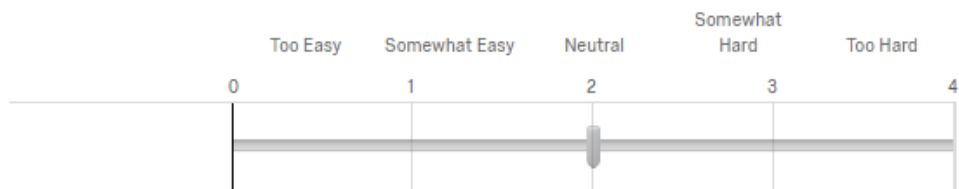
Do understand why you received the final score that you did?



- Yes
- No
- What score?

Q5.5


Overall difficulty?





Q5.6


What additional methods of scoring the player might you want to see implemented? **(Optional)**



Q6.1 **Final Thoughts (Optional)**


Q6.2 What questions do you still have about the game, the way its played, or the reasoning for the way it was created?


Q6.3 Any additional recommendations?


Q6.4 Final comments?


B SONG FILE EXAMPLE

“No_Title_Example.txt”
mikugame file v1

[General]

SongTitle:No title
UnicodeTitle:No title
ArtistName:Reol
UnicodeArtist:Reol
FileName:no_title.wav
ModelCharacter:UE4_Miku_no_title
GameAnimation:UE4_Miku_no_title_anim
PreviewTime:67143
Tags:Reol Giga-P no title Cable non-vocaloid
Creator:Cable
Difficulty:Hard
SongID:000001

[Timing]

1098,200,4,4,0,0,1,0
68161,200,4,4,0,0,0,1

[Colors]

STAGE,151,255,151,50
STAGE,255,89,89,50
PENLIGHT,213,255,170,50
PENLIGHT,0,170,170,50

C PUZZLE-SOLVING ROBOT

Part of the expectations of both Worcester Polytechnic Institute, our home university, and Ritsumeikan University, our host universities, were for us to work on a joint project with Ritsumeikan students. The assigned task was to modify a 3D XY plotter to create a puzzle-solving robot. Each lab was tasked to work together with the visiting students, and their work would be graded based on the complexity of the puzzle they can solve. The goal of the project was to build a relationship and work together with some of the other students in our lab. However, due to time constraints and unfamiliarity with Arduino and robotics engineering in general, we were unable to complete the puzzle-solving robot.

Our design was to use a camera on the robot to scan each piece and use computer vision to determine the correct location for each piece. This computer vision algorithm required significant preprocessing and required the puzzle to be solved first in order to determine the location of each piece. This method is slow, but we determined two methods to decrease the operational complexity of this work. First, we would divide pieces into three buckets: central pieces, with no flat sides, edges, with one flat side, and corners, with two flat sides. We could identify these sides quickly and reduce the number of pieces that we need to compare to with this. Our second method involved checking the average color of a piece based on a normal random sampling of color swatches on the piece, and using a confidence interval to limit the number of comparisons further.

Without anyone with an engineering background, the largest problem that our team encountered was designing and building the robot itself. The XY plotter kit that was provided came with incomplete documentation and a severe lack of instructions. Construction went smoothly, but when it came to operating the plotter, it became necessary to look beyond official documentation to achieve movement. This was the easy part; designing and assembling an appendage that would be capable of lifting, rotating, and otherwise finely manipulating a puzzle piece was another challenge entirely. We did not begin to tackle this part of the puzzle-solving robot before we left Japan.

Several organizational issues led to the puzzle-solving robot project being left incomplete before we left Japan. First and foremost, we were unable to secure a consistent meeting time with the Ritsumeikan University students we collaborated with. A significant portion of our time in Japan took up their entire summer break, and therefore we could not easily meet in person to work on the project. Beyond this, no one from WPI or Ritsumeikan on the team had any experience with engineering an appendage that was capable of fine manipulation, nor any robotics project of this sort. Unlike last year’s dash button project, the puzzle-solving robot fell well outside of the proficiency of the group members and was likely to fail from the beginning.

D AUTHORSHIP

Section	Primary Author	Secondary Author
1 Introduction	Aidan O’Keefe	
2 Background	Aidan O’Keefe	
2.5 The Unreal Engine Game Structure	Patrick Critz	
3 Design and Gameplay		
3.1 Content Sourcing and Art Design	Aidan O’Keefe	
3.2 Environment Design	Patrick Critz	David Giangrave
3.3 Gameplay Design	Aidan O’Keefe	
3.4 User Interface Design	David Giangrave	Patrick Critz
4 Gameplay Implementation	Aidan O’Keefe	
5 Utilities	Patrick Critz	
6 Cinematic Artificial Intelligence	David Giangrave	
7 Testing Process		
7.1 Paper Prototype	David Giangrave	Aidan O’Keefe
7.2 Gameplay Testing	Aidan O’Keefe	David Giangrave
8 Post-Mortem	Everyone	
Appendix A: IRB Consent Form and Playtest Survey	David Giangrave	Aidan O’Keefe
Appendix B: Song File (.vdg) Example	Aidan O’Keefe	
Appendix C: Puzzle-Solving Robot	Patrick Critz	

REFERENCES

- [1] blanche0u0. 2018. かな子ちゃんとお願ひ！シンデレラちょっと踊ってみました デレステARありがとう...! Twitter post. <https://twitter.com/blanche0u0/status/1038990000279281664>
- [2] Blain Brown. 2012. *Cinematography: Theory and Practice* (2nd ed.). Elsevier Inc., Waltham, Massachusetts.
- [3] G. C. Burdea and P. Coiffet. 2003. *Virtual Reality Technology* (1st ed.). John Wiley & Sons, Hoboken, NJ.
- [4] Eric Chiang. 2014. Python Multi-armed Bandits (and Beer!). Retrieved October 11, 2018 from <http://blog.yhat.com/posts/the-beer-bandit.html>
- [5] Cygames. 2015. The Idolm@ster Cinderella Girls: Starlight Stage. iOS and Android application.
- [6] deresute_eng. 2018. AR studio. Twitter post.. https://twitter.com/deresute_eng/status/1036286639092297728
- [7] Joshua Eckroth. 2017. Reinforcement Learning. Retrieved October 11, 2018 from <http://web.cse.ohio-state.edu/~stiff.4/cse3521/reinforcement-learning.html>
- [8] Ubisoft Paris et al. 2017. Just Dance 2018. Xbox One Application.
- [9] Marco Ghislanzoni. 2017. Super-easy VR body with Arm IK and thumbstick locomotion – Unreal Engine 4 Tutorial. Video. <https://www.youtube.com/watch?v=EKR8ogonD68>
- [10] David Graham. 2013. *Game AI Pro: Collected Wisdom of Game AI Professionals*. Vol. 1st. CRC Press, New York, NY, 113–126.
- [11] Harmonix. 2010. Dance Central. Xbox 360 application.
- [12] HatsuneMiku. 2016. Anamanaguchi - Miku ft. Hatsune Miku (Lyric Video). Video. <https://www.youtube.com/watch?v=NocXEwsJGOQ>
- [13] S. Hayden. 2018. ‘Project LUX’ Creators Announce New VR Visual Novel ‘Spice & Wolf VR’. <https://www.roadtovr.com/project-lux-creators-announce-new-vr-visual-novel-spice-wolf-vr/>
- [14] Gregory Hollows and Nicholas James. 2016. Understanding Focal Length and Field of View. Retrieved October 11, 2018 from <https://www.edmundoptics.com/resources/application-notes/imaging/understanding-focal-length-and-field-of-view/>
- [15] HoshinoRico. 2018. デレステAR...最高だよ.....!(感涙). Twitter post. <https://twitter.com/HoshinoRico/status/1038300762793627648?s=19>
- [16] IBISWorld. 2018. Arcade, food, I& entertainment complexes – US market research report. <https://www.ibisworld.com/industry-trends/market-research-reports/arts-entertainment-recreation/arcade-food-entertainment-complexes.html>

Publication date: October 2018.

- [17] Implustechnology. 2017. Multi-Armed Bandits Intro. Video. Retrieved October 11, 2018 from <https://www.youtube.com/watch?v=qAvY2tkMHH4>
- [18] Arthur Juliani. 2016. Simple Reinforcement Learning with Tensorflow Part 1.5: Contextual Bandits. Retrieved October 11, 2018 from <https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-1-5-contextual-bandits-bff01d1aad9c>
- [19] H. Kenmochi. 2010. VOCALOID and Hatsune Miku Phenomenon in Japan. In *InterSinging 2010 - First Interdisciplinary Workshop on Singing Voice*. InterSinging Committee, Tokyo, Japan.
- [20] Konami. 1998. Dance Dance Revolution. Arcade application.
- [21] Konami. 2017. Beatmania IIDX 25: CANNON BALLERS. Arcade application.
- [22] Konami. 2017. SOUND VOLTEX III: GRAVITY WARS. Windows application.
- [23] Konami. 2018. DANCERUSH Stardom. Arcade application.
- [24] ky_gtr. 2018. テレステARで夢が叶いました。方々の皆様、本当にすみません。 Twitter post. https://twitter.com/ky_gtr/status/1038081423695990784
- [25] M. Lalwani. 2016. It Takes a Village: The Rise of Virtual Pop Star Hatsune Miku. <https://www.engadget.com/2016/02/02/hatsune-miku/>
- [26] L. K. Le. [n. d.]. Examining the Rise of Hatsune Miku: The First International Virtual Idol. *The UCI Undergraduate Research Journal* ([n. d.]), 1–12. http://www.urop.uci.edu/journal/journal13/01_Le.pdf
- [27] D. Liu. 2002. A Case History of the Success of Dance Dance Revolution in the United States. http://web.stanford.edu/group/htgg/sts145papers/dliu_2002_1.pdf
- [28] E. Loo. 2013. Miku English is Hatsune Miku’s 1st Windows/Mac Release. <http://www.animenewsnetwork.com/news/2013-07-07/miku-english-is-hatsune-miku-1st-windows/mac-release>
- [29] Hyperbolic Magnetism. 2018. Beat Saber. Windows Application.
- [30] J. Marsh. 2017. How VR Theme Parks are Changing Entertainment in Japan. <https://www.cnn.com/travel/article/vr-parks-on-japan/index.html>
- [31] Crypton Future Media. 2018. Hatsune Miku Magical Mirai 2018. https://magicalmirai.com/2018/index_en.html
- [32] P. St. Michel. 2014. The Making of Vocaloid. <http://daily.redbullmusicacademy.com/2014/11/vocaloid-feature>
- [33] nmm_fujiko. 2018. 絶対同じこと考えた人いるだろうけどやりたかったの...!! Twitter post. https://twitter.com/nmm_fujiko/status/1038052730739208192
- [34] J. Oloiza. 2014. Does Hatsune Miku’s Ascent Mean the End of Music as We Know It? <https://tmagazine.blogs.nytimes.com/2014/10/16/hatsune-miku-music/>
- [35] The Japan Times Online. 2017. Japanese planetariums and art museums tap VR craze to offer immersive experiences. <https://www.japantimes.co.jp/news/2017/12/07/national/japanese-planetariums-art-museums-tap-vr-craze-offer-immersive-experiences/>
- [36] The Japan Times Online. 2018. Vaio to be First in Nation to Screen VR Movies in a Theater. <https://www.japantimes.co.jp/news/2018/06/26/business/tech/vaio-first-nation-screen-vr-movies-theater/>
- [37] Robert Penner. [n. d.]. Robert Penner’s Easing Functions. <http://www.robertpenner.com/easing>
- [38] Dean ‘peppy’ Herbert. 2007. osu! Windows Application.
- [39] E. Petrarca. 2016. Meet Hatsune Miku, the Japanese Pop Star Hologram. <https://www.wmagazine.com/story/hatsune-miku-crypton-future>
- [40] Rama. [n. d.]. Rama’s Victory Plugin. <https://github.com/EverNewJoy/VictoryPlugin>
- [41] Nick Robinson. 2018. What happened to ‘Domino’s App feat. Hatsune Miku?’ (Documentary). Video. <https://www.youtube.com/watch?v=V2RytpvQ9BA>
- [42] Stuart Russel and Peter Norvig. 2010. *Artificial Intelligence A Modern Approach* (3rd ed.). Pearson Education Inc., Upper Saddle River, New Jersey.
- [43] Y. Sambe. 2009. Japan’s arcade games and their technology. *Entertainment Computing – ICEC 2009* (2009), 338.
- [44] Sega. 2017. CHUNITHM STAR. Arcade application.
- [45] Sega. 2018. ONGEKI. Arcade application.
- [46] Sega and Crypton Future Media. 2016. Hatsune Miku Project Diva X. Playstation Vita Application.
- [47] Sega and Crypton Future Media. 2017. Hatsune Miku Project Diva: Future Tone. Playstation 4 Application.
- [48] sekiyu_p. 2018. 志希にゃんの単独ライブin我が家!! ちょっと巨人になったり宙に浮いたりしたけどなんくるないさー!! Twitter post. https://twitter.com/sekiyu_p/status/1037987451610882048
- [49] Faizan Shaikh. 2017. Simple Beginner’s guide to Reinforcement Learning & its implementation. Retrieved October 11, 2018 from <https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/>
- [50] shuntan_sherlo1. 2018. #SS3A に行けなかった僕のもとに、五十嵐響子ちゃんが来てくれました!! Twitter post. https://twitter.com/shuntan_sherlo1/status/1038765838801620992
- [51] Sareesh Sidhakaran. 2014. What is the 35mm Equivalent and Why is it Confusing. Retrieved October 11, 2018 from <https://wolfcrow.com/blog/what-is-the-35mm-equivalent-and-why-is-it-confusing/>
- [52] Rock Band Database Team. 2018. Rock Band Database. Database. <https://rddb.online/>

- [53] Stepmania Team. 2001. Stepmania. Windows Application.
- [54] VocaDB. 2018. Songs, Albums, and Artists Database. Database. <https://vocadb.net>
- [55] Keisuke Yamada. 2017. Thoughts on Convergence and Divergence in Vocaloid Culture (and Beyond). <https://ethnomusicologyreview.ucla.edu/content/thoughts-convergence-and-divergence-vocaloid-culture-and-beyond>
- [56] 米津玄師. 2017. ハチ MV 「砂の惑星 feat.初音ミク」 HACHI / DUNE ft.Miku Hatsune. Video. <https://www.youtube.com/watch?v=AS4q9yaWJkI>