ONI X MATSURI

Team Name: Kyoto Cats

Team Members: Elizabeth Churinov, Ricardo Correa Netto, Chandler Fox, Rachel West, Chase Ruskin

CIS 4914 Senior Project

Advisor: Dr. Jeremiah Blanchard

Advisor’s Email: jjb@eng.ufl.edu

https://github.com/Changreenfox/ONI-X-MATSURI

Abstract

Different types of games have long been used in all cultures to connect people and help them train different skills. The newest form of these games is the modern video game. Video games can be fun, train skills like hand-eye coordination, and educate people about different cultures. Developing video games, however, can be a complex field to join. How could we make the field easier to enter? To resolve this issue, we have developed our own video game in Godot. This game is open source, with the repository and all the code available to the public. The code and systems for the game are also well-documented and explained. The game has been designed to be as modular and moddable as possible. As such, aspiring game developers could learn how to make games by taking what is written in our code and adding to it by developing new enemies, power-up types, and more. The theming of the game is based on Japanese culture and folklore to show the use of video games and display how they can spread awareness of different cultures and traditions.

Keywords

Artificial Intelligence, Arts and Humanities, Game Design, Game Development, Interaction Design.

Introduction

Games have been created and played by humans across the globe for thousands of years. They are an integral part of cultures as they provide people with entertainment, sharpen minds with critical thinking and problem-solving, and communicate cultural ideas to broader audiences. Exposing audiences to different cultures can improve their understanding of the world and build interest in expanding one's knowledge. Our goal is to continue to fulfill this role in digital culture. Through our project, we strengthened cognitive abilities with gameplay and hardware design and exposed a different culture to a western audience through the project's theme.

The main problem we hope to address is that game development can be a complex space to enter, even with coding experience. Not only do we aim to entertain with our game, but we also aim to teach. The project is designed to be as modular as possible. That way, an individual learning game development can piece together a "modded" version of the game we ship and play something all their own.

Related Work

The team decided to build the game in the popular side-scrolling platformer style. This style is most famously depicted in the early *Super Mario Bros.* games, which Nintendo developed in the 1980s. Nintendo later used aspects of a side-scrolling platformer and mixed combat elements in the second installment of The Legend of Zelda franchise, *Zelda II: The Adventure of Link*.

A popular game that closely depicts the aesthetic of the Japanese Summer Festival (an aesthetic we wish to reproduce) is *Taiko no Tatsujin*, a rhythm drumming game in Japanese arcades and on consoles. Its menus, gameplay, and flashy artwork depict aspects of the summer festival, creating an attractive and fun vibe that inspired our idea of how our game should feel when played.

Motivation

Our product, ONI x MATSURI, is a reimagined 2D side-scrolling platformer game that features player movement, combat, enemies, powerups, bosses, and more. The game features evil Oni that invaded the Summer Festival, led by an OniBoss, who sits atop the festival taiko drum tower. The player character is a samurai cat who must fight through the Oni and defeat their evil leader to save the Summer Festival. The game also features colorful and engaging Japanese festival themes accurate to the culture and captured in a pixel art style.

In addition, a custom console and controller will be built to accompany the game, hopefully supporting other software built using the Godot Engine. A factor of this hardware is to experiment with a joystick dedicated to attacks rather than movement, similar to the *New Nintendo 3DS*'s c-stick and how it was used for strong attacks in the game *Super Smash Bros. for the 3DS*.

Problem Domain

Developing video games is often very involved and technical. This difficulty is compounded when someone wants to port the game project to a console. Our solution is to provide a game baseline with an easily-expandable codebase from which future game developers can learn. We also plan to publish documentation on creating a homemade console so people can easily play games from a console port. We hope this will inspire more people to become game developers and lower the barrier to entry into game development when using the Godot software.

Literature Review

When deciding what to do for our senior capstone project, we considered many factors before landing ONI x MATSURI.

Firstly, we had to decide what to create. We wanted a video game for two reasons: we have experience with game creation and enjoy doing it, and we have experienced the positive effects of video games first-hand. David Corso of South Carolina University writes about the positive impact of video games, saying they “train intellectual skills as well as improve physical skills” (2014). This is done “through a cyclical flow of bottom-up and top-down processing” (Corso, 2014) involving input from the game world affecting the player and the player affecting the game world.

Since two of the team members are Computer Engineering majors, we wanted room for growth of scope beyond a single semester while still creating a deliverable for this one. Since several resources are online for building a homemade console, we decided it would satisfy the hardware component without removing the original intent of making the game. One such resource we plan to use is GameDeveloper’s Building a 'homebrew' video game console, which goes into extensive detail about the steps that will be required.

The game also needed research since we wanted to tie it to Japanese mythology. The event of the game is that a band of Oni takes over the “Natsu Matsuri” (Japanese summer festival), and the hero needs to defeat them. This means we required extensive research on what Oni looks like in mythology and media. From the book Japanese Demon Lore, we know that the image of Oni is that of “typically gruesome appearances… often reflective of their allegedly evil dispositions” (Reider, 2010).

However, Oni have shape-shifting powers, so Japanese people have a loose method of describing them in Mythology. Media often depict them as red ogres with two horns sticking out from their heads. We all got to experience the Gion Matsuri festival while studying abroad in Japan, so we did not need to search for any academic articles telling us what the festival appears like.

Solution

Software Structures

The purpose of our game is not only to create a quality product but also to provide a learning module for future game designers to append to quickly. While games include several user-end functionalities, the backend is where several game mechanics and systems are implemented. Our code processed every button pressed, input received, and change in visuals in a non-user-friendly manner. We also need the game to be as dynamic as possible for the programmer side. This allows future game designers to pull from our code and change any aspect they want.

Luckily, some of the functionality is abstracted for us thanks to the Godot game engine we used to build the game. We chose Godot because it was completely open-source and easily portable to several OS options, which will better allow us to focus on the hardware side next semester. In the preceding discussion, Godot modules provided by the engine will be outlined in UML, and proposed methods of extending the code will have a green background.

Scenes

The many different levels are saved in Godot as Scenes. A Godot **Scene** can be any number of Godot Nodes and are helpful for dynamically loading/unloading the game, depending on what we need. A Godot **Node** can be anything we want to be physically in the game (displayed or not), and all scripts must be attached to a Node (most of the time). Scenes are also helpful for spawning objects dynamically (i.e., enemies, bullets, platforms, etc.). The power behind this structure includes organizational benefits and the ability to load entire structures at once. For example, we could load multiple **StartScreens** without recreating the complicated structure. A simple form of the Scene hierarchy with a UML is shown below (Figure 2) for **StartScreen**:

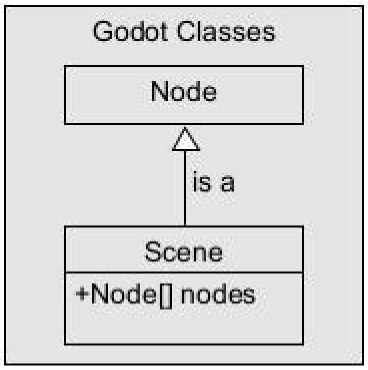


Figure 1: A simple UML describing Godot's scene hierarchy

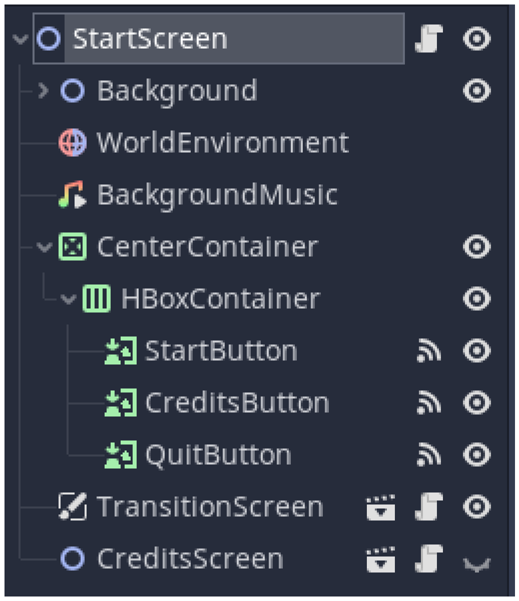


Figure 2: Highlights Godot scene hierarchy

Actors

The most crucial aspect of any video game is interaction. How does the player cause a change in the world? How does that change affect the player? Without this level of communication between the user and the system, there would be no game. Getting user input will be mentioned in a later section, but discussing how the backend handles said input is essential.

To promote dynamic structures in our code, all intelligence (either artificial intelligence or the player) inherits from a handwritten class *Actor.cs*, which in turn inherits from a **KinematicBody2D** to handle the Godot editor better. The **Actor** class is designed to use a Finite State Machine to determine behavior and interactions (Shown in Figure 3 below). This allows us to create **States** that can share behavior.

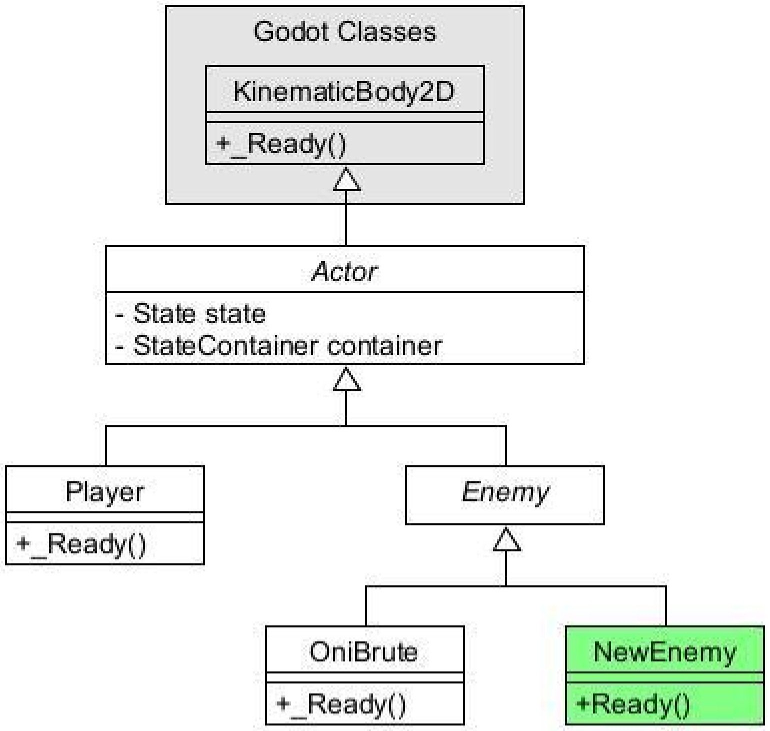


Figure 3: A UML displaying an example of how to add a new enemy

For instance, both a player and enemy can walk, but the direction and velocity are determined case-by-case. We can have an abstract *Motion.cs*, inheriting from the *State.cs* interface, with functionality to move a provided velocity in a valid direction. *PlayerMotion.cs* is a state that allows the player to change direction with user input, whereas *Approach.cs* blindly runs toward the player's current location. This enables game developers to add any functionality they want by creating new states.

An example of this is shown in the UML below:

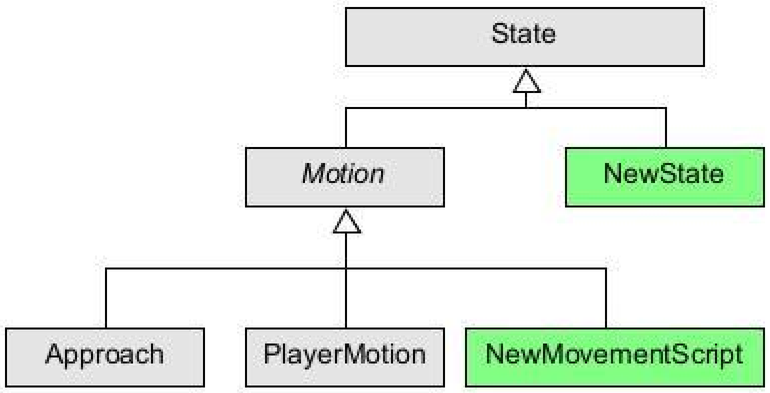


Figure 4: A UML displaying an example of how someone could add new States.

The **Player** Scene has a *Player.cs* script attached to a KinematicBody2D node. The Finite State Machine for the Player is assigned new States during the _Ready() function provided by the KinematicBody2D (shown above in Figure 3). The State Machine for the Player is shown below:

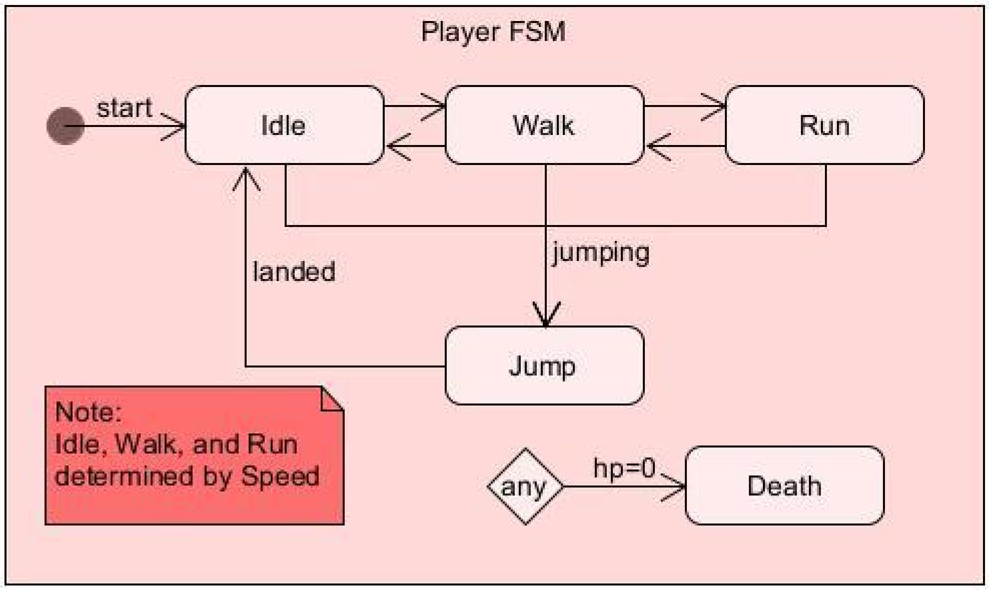


Figure 5: Displays the states for the player character.

The process for creating a new Actor would be to create a new class that inherits from Actor and overrides the KinematicBody2D _Ready() function. They would fill the state container with the desired state name and newly-created State classes (that way, a new class is not created every time it's needed) and assign the Actor's protected value state to the starting point. They can then call state.enter(), and the Finite State Machine has begun.

PowerUps

The Player can also receive **PowerUps** from within the scene by running over them. Besides the UI element shown while a powerup is active, it also affects the constant values used by the states depending on the PowerUp. For example, a **JumpPowerUp** gives the player more jump height (which is used in Jump.cs), while an **AttackPowerUp** increases the damage dealt by the Player's Attacks. There is a base functionality class called *PowerUp.cs* that, if inherited, can allow for easy new PowerUp creations for future game developers. The UML describing the above relationship is provided below:

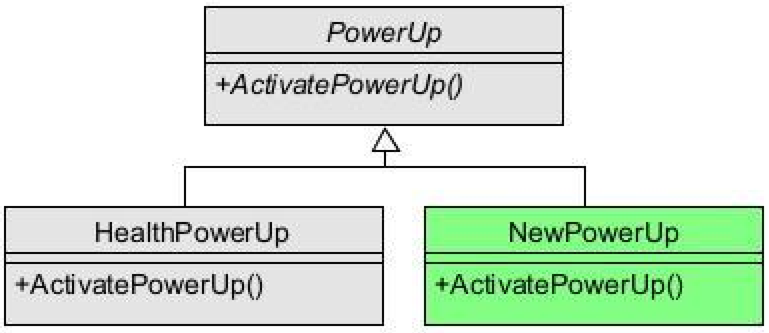


Figure 6: UML for the PowerUp functionality.

Attacks

Another important subsystem to mention would be the dynamic types of Attacks. **Attacks** are another abstract base class (allowing for extension) currently used in several different situations. The **BasicAttack** class allows a melee attack that updates which **Collider** is activated during animation (so the active Collider better matches what the user sees).

The **BulletSpawner** class spawns an instance of any **Bullet** Scene and gives it a heading to move towards. A Bullet is a different attack that moves in a constant direction (BulletSpawner sets gravity) and destroys itself after a few seconds OR on collision. With these two interactions, a BulletSpawner can spawn different bullets with different functionalities, all on the same enemy. All Attacks can be assigned to an Actor in the Node hierarchy (as children); doing this allows the Attacks to be auto-populated in the Actor's own Attacks[] attacks member variable. Any State can call StartAttack() at any time, depending on the desired behavior. The UML for this structure is shown below:

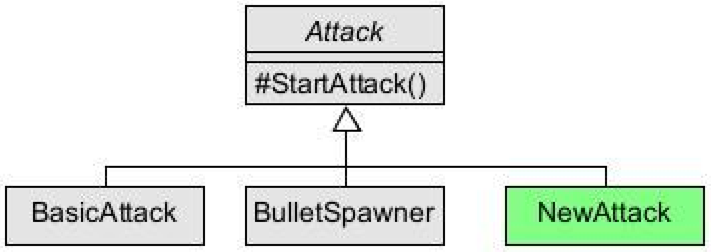


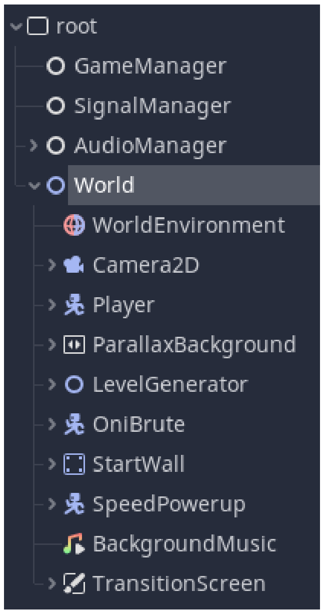
Figure 7: The UML for the attack structure.

Communication Mechanisms

The structure of the components in Godot are saved as Node tree structures. **Signals** and **Singletons** are the primary communication between the Nodes in the game tree.

Nodes can be connected to a Signal signature, along with a function to handle receiving the Signal. Any Nodes within the game tree can emit a signal to be caught by any Nodes listening for that specific signal. This is useful to transport data to a variable number of Nodes (components) that are expecting the data. Godot offers many built-in Signals defined within the Godot classes, such as **Timer**, KinematicBody2D, etc. User-defined custom Signals can also be created for custom functionality to be attached to the base Godot and user-defined classes.

Singletons are Node trees that exist outside the Scene tree but within the game tree along with the Scene tree. An example is shown below:



```
root
    GameManager
    SignalManager
  > AudioManager
  v World
      WorldEnvironment
    > Camera2D
    > Player
    > ParallaxBackground
    > LevelGenerator
    > OniBrute
    > StartWall
    > SpeedPowerup
      BackgroundMusic
    > TransitionScreen
```

Figure 8: Scene Tree

In the image above, we can see the root tree, which is the "game tree." Within the game tree, there exist Singletons, which are loaded first, and the current Scene tree–in this example, the current Scene is "World". **GameManager**, **SignalManager**, and **AudioManager** are Singletons that persist between different Scenes (menus and levels). These are useful to establish and maintain persistent data, such as maintaining the Player's HP between levels, audio levels, and user-defined custom Signals. Signals and function calls are often used to pass data to the Singletons and vice versa.

Visual Rigor

The game's aesthetic and accuracy to culture hold well with the current assets. Since most of the team members took inspiration from their personal experiences in Japan, most of the assets are sourced from folklore or photos we have taken ourselves. Many aspects of the background, such as the mountains, shrines, and temples, are based on the feel of the city of Kyoto, which is well known to be the heart of the traditional culture of Japan. Careful attention was also placed on details such as the writing and that each asset with text makes sense in the original language, despite the limitations of pixel art. Other details, such as the main character's clothing folding left over right (the reverse is used in the connotation of death) and enemy design accuracy, was also scrutinized.

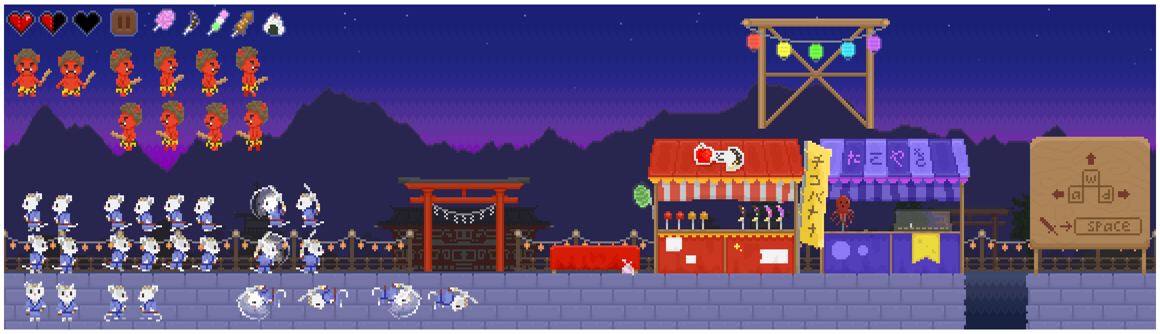


Figure 9: Asset Sheet from Early Development

To ensure that the player has a visually enjoyable and engaging experience, the game underwent a process of visual rigor. This included adding animations to static objects, incorporating bloom and other post-processing effects, parallax, and 2D sounds.

Results

Currently, the team is extremely happy with the state and look of the game. Below are snapshots of how the game looks in the primary levels of the playthrough. The first image is of the start screen of the game. This start screen shows the character standing in front of a Torii gate in front of a shrine. In the background are fireworks. This sets up the location of the game, which is at a festival at a shrine. In this scene, you can click start, which would start the game, credits which would display the credits of the game, or quit, which closes the game.

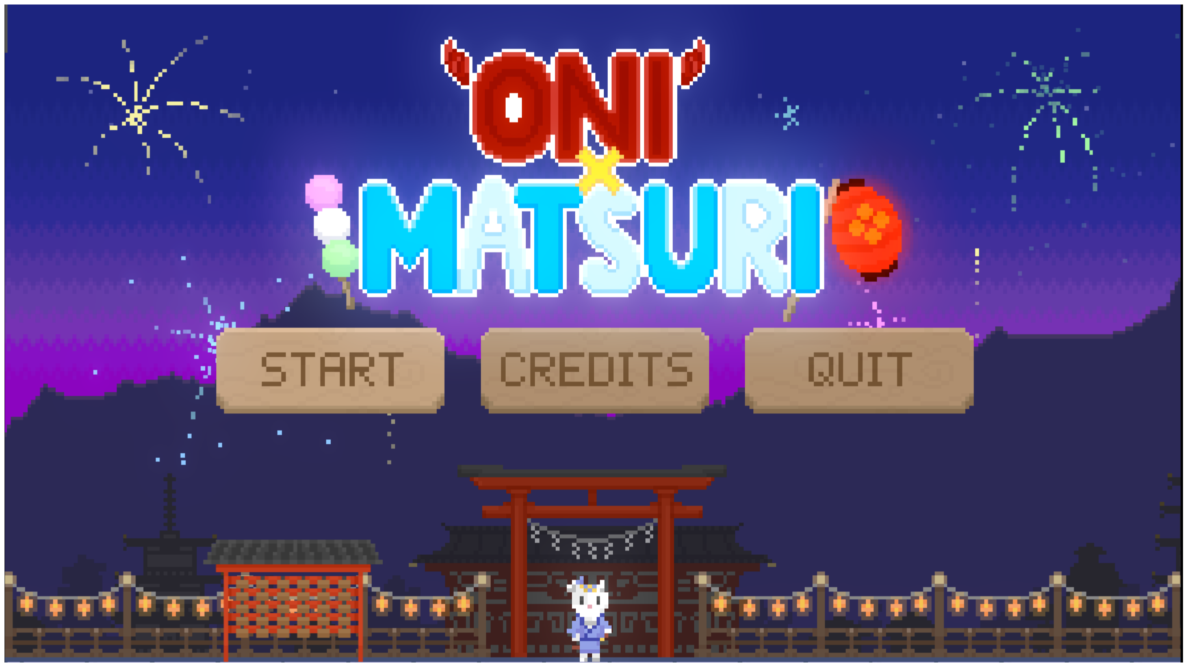


Figure 10: Start Screen

Once you start the game, you start on the first level of the game, the main world. This is a horizontally scrolling level, where you can only move forwards in the level, but not backward. This is to incentivize the player to keep forward motion and not go back for power-ups. You can see your health in the top left. In the top middle of the screen, you can see a score counter which is represented by a five yen coin. At most Japanese shrines, you complete prayers by donating five yen to the shrine. As such, we saw it as appropriate to represent our score with these coins. On the left is the pause button. In the screenshot below, you can see the main character of the game fighting against two different types of enemies. The first is an OniBrute, which is red, and the second is an OniThrower, which is blue. The OniThrower throws beans at the player, which is a reference to the Japanese tradition of throwing beans during the festival of Setsubun. Killing these enemies is how you increase your score in the game.

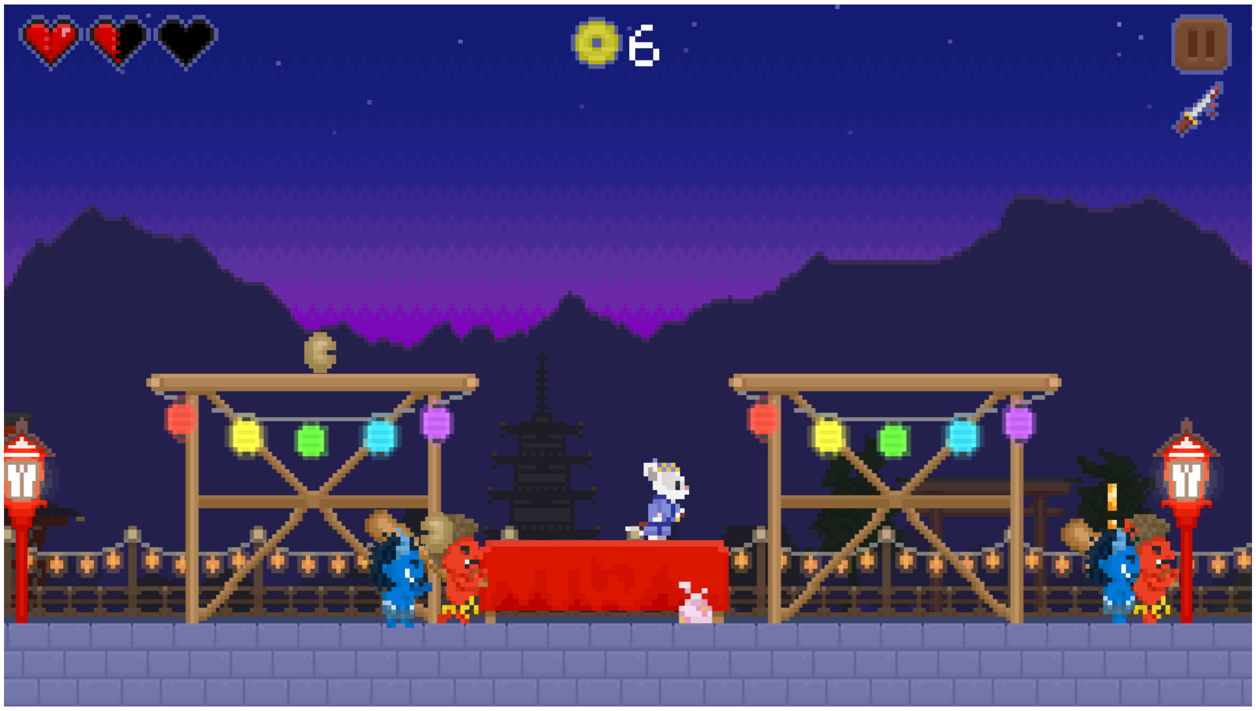


Figure 11: Level 1 (World) Screenshot

The next screenshot displays the Tower level of the game. This level is a vertically scrolling level, where the goal is to reach the top of the tower. There are new platforms here, like the Taiko, which is a type of drum used in Japan. These platforms work as jump pads to help your player reach greater heights in the level. In this screenshot, you can also see different power-ups of the game. The power-ups shown are a stick of Dango, which is a Japanese dumpling, and Onigiri, which is a Japanese seaweed rice snack. These power-ups give a jump boost and a speed boost respectively. Additional power-ups in the game are an Ikayaki, or a squid on a stick, which gives an attack boost, and cotton candy, which heals the player with one heart. These power-ups are based on snacks we observed to be popular at Japanese festivals.

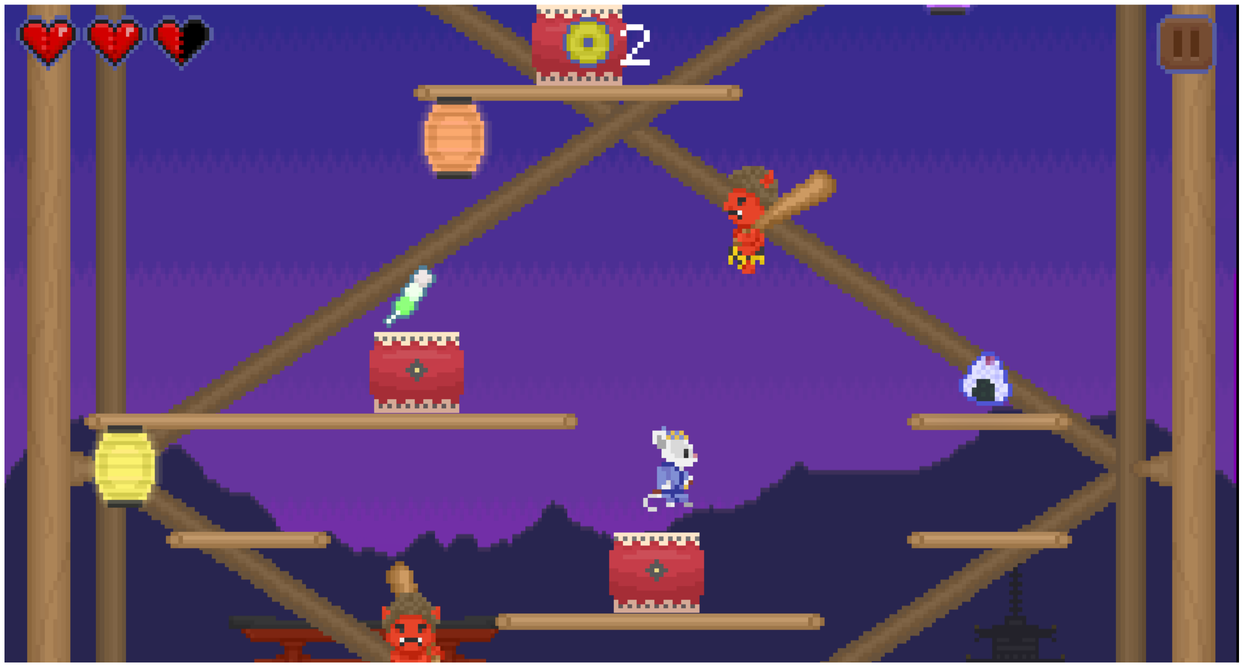


Figure 12: Level 2 (Tower) Screenshot

The last stage of the game can be seen in the screenshot below. This is a stationary screen that contains the final boss fight of the game. Defeating this boss leads the player to the win screen. The boss attacks by hitting a Taiko.

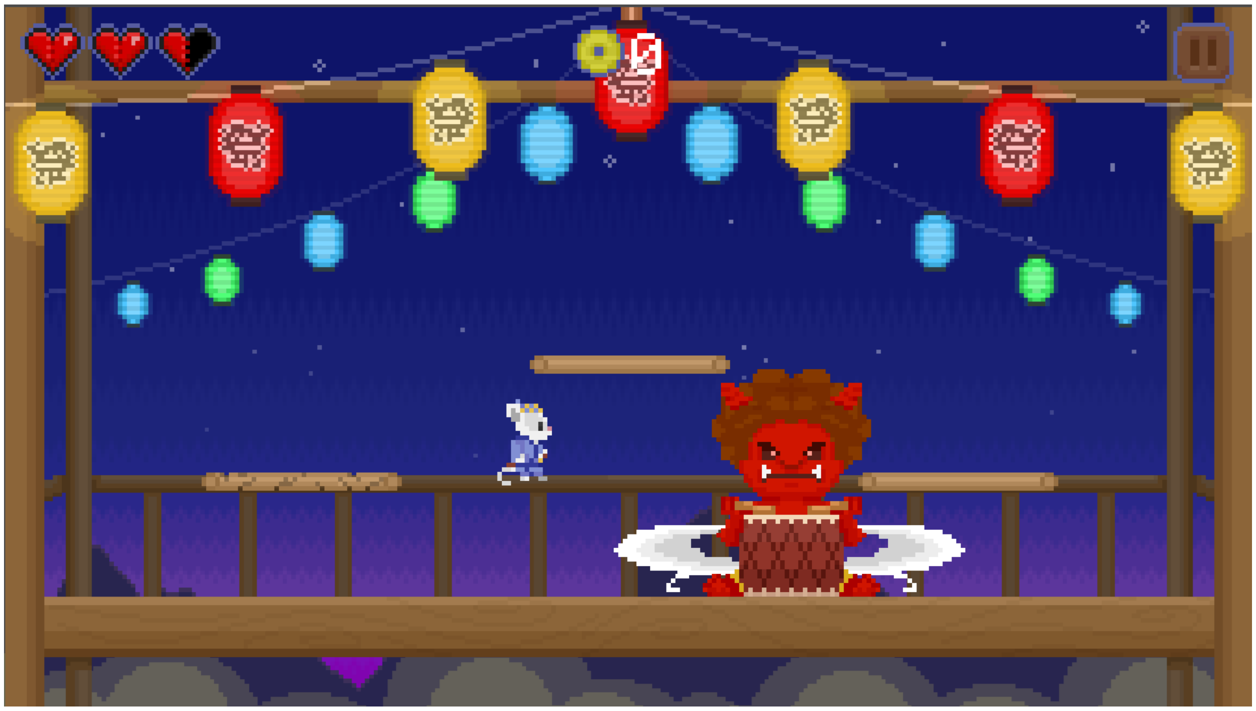


Figure 13: Boss Level

From each of the scenes, we can show that we used different aspects of Japanese cultures and traditions as parts of our game. Everything is based on research completed about and in Japan, from the scenery to the enemies and even the types of snacks displayed in the game. As such, our solution achieved its goal of spreading culture and cultural awareness to those who play it.

We accomplished the goal of creating an easily expandable codebase as well. The UMLs described in the Solution section above showcase several different methods of creating something new. So long as someone has the assets, we have detailed how to create a new Enemy, new Attack, new State, new Level, and new PowerUps. This will allow aspiring game developers to touch on each section of the game one at a time, allowing someone to learn at their own pace.

Conclusions

Development Summary

During this semester, our team continued development from previous work, which was a similar game by the same name built in Python. The development began by porting our work written in Python using the Python Arcade game engine to C# using the Godot game engine. Godot was chosen as our game engine for a few reasons. Firstly, Godot offers a more robust, powerful, and complete game development framework. Additionally, unlike most game engines like Unity, Godot is open source. This means that future work developing the game console can be done with more reliability and ease. Upon determining which game engine to use, we decided to take apart our original code and utilize Godot's features to improve our code readability and scalability. We first rebuilt the game in Godot and then improved it by adding many new features and polishing existing elements.

Despite rebuilding the game, most of the base elements of the game came from the original. The game is a side-scrolling platformer with many assets, and most of the music and sounds came from the original version of the game. Player movement, Power-Ups, their functionality, procedural generation, most of the platforms from the world stage, and the OniBoss design and attack patterns also came from this old version. However, upon completing the rebuild, we noticed many aspects of the game that could be added upon or polished.

Modifications to the new game included adding different platforms for the main stage, an entirely new level (the Tower), a score counter, different enemies, and platforms to make the boss more beatable. The visual polish was also added, including lighting effects, shaders, and new animations and movements. For the User Interface, a pause menu was added so players could pause as needed, and a credits scene was added to give credit to the developers, team advisors, and any outside assets used.

Through our continued development of previous work and porting it from Python to C#, using different game engines, our team learned a great deal about software, languages, computer science concepts, and game design and development. Game development is a field that deals with an extensive number of areas in computer science, which offers an excellent opportunity to learn and understand more about various aspects of computing. Our development involved digital art design and creation, audio, animations, and multiprocessing.

Future Work

Future development on the game and related work includes software and hardware. We would like to make some final additions and polishing touches for software work on the game. These additions include adding two more enemies to the game. These enemies are the Tengu, a flying enemy, and the Kijin, a primarily stationary enemy. These enemies will be added because there is not a large amount of diversity in enemy types, and lore is related to them. The game's goal is to include aspects of Japanese folklore; thus, having more creatures added to the game based on it would help our goal. These enemies would be added to the Tower stage of the game instead of the OniBrutes, to differentiate it more from the first stage. The Tengu would be unkillable and would follow the player around while throwing attacks for the player to dodge. The Kijin would wait on platforms for the player to get into range and complete a quick and powerful attack. The goal would be to avoid encountering the Kijin while dodging the Tengu. An additional modification to current enemies will be added, a third stage for the OniBoss to make him more challenging to defeat.

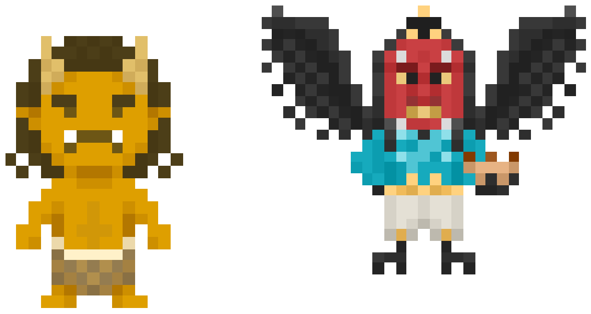


Figure 14: Kijin and Tengu Base Art

More work will be done by implementing a final score for the game. This score will be calculated by the amount of time the player took to defeat the game (where less time means a higher score) and the number of coins they collected through their playthrough. This will be done to incentivize the player further to interact more with the enemies rather than trying to dodge all of them while also not spending too much time doing so. The final planned modifications to the game include polishing with the lighting and shaders, general bug fixes, and adding more chunk variation to the procedural generation.

The bulk of this project's future work will be done on the hardware side. The goal will be to develop a game console to run this game and other games built in the Godot game engine. The console will also be paired with a controller that will be used as another medium to play the game; controller support will be extended from the keyboard support we have already integrated into the game.

The game is planned to be stored on an SD card for easily swapping between different games on a single console. The SD card reader and the video game console will communicate over the SPI communication protocol. To communicate between the external monitor displaying the game's graphics and the console, there will be support for VGA and HDMI. Regarding the hardware controller, it will behave as a standard keyboard device from the view of the video game console and communicate over the USB protocol. The planned development for the video game console is proposed to target a Raspberry Pi microprocessor to start as the baseline for building the console's software layer and integrating its hardware components together.

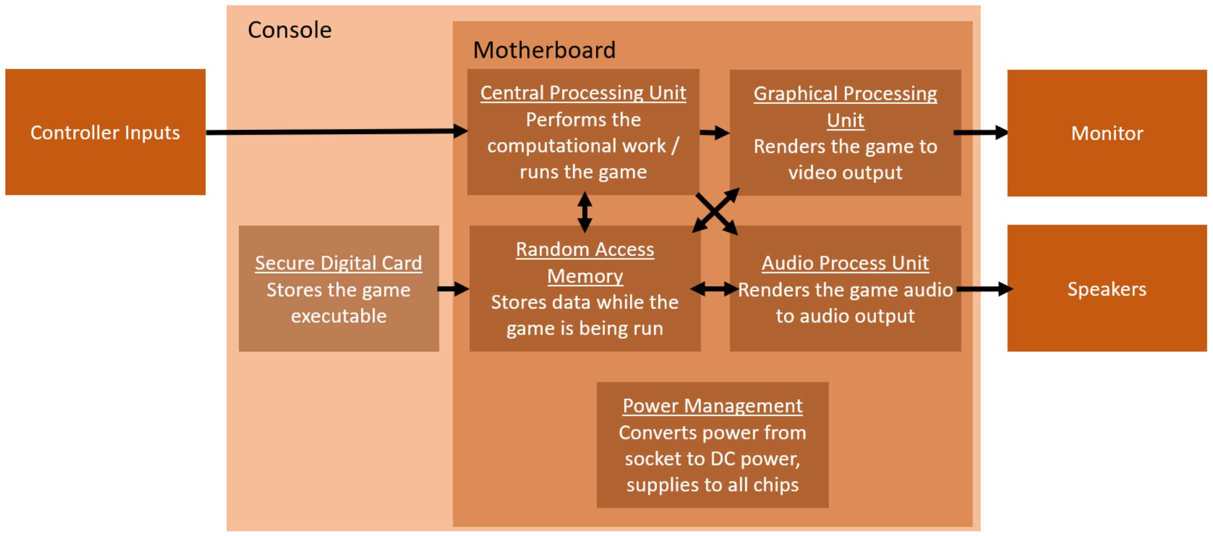

Console
Motherboard
Controller Inputs
Central Processing Unit
Performs the computational work / runs the game
Graphical Processing Unit
Renders the game to video output
Monitor
Secure Digital Card
Stores the game executable
Random Access Memory
Stores data while the game is being run
Audio Process Unit
Renders the game audio to audio output
Speakers
Power Management
Converts power from socket to DC power, supplies to all chips

Figure 15: Displays the hardware schematics for the planned console, with all major components connected to each other.

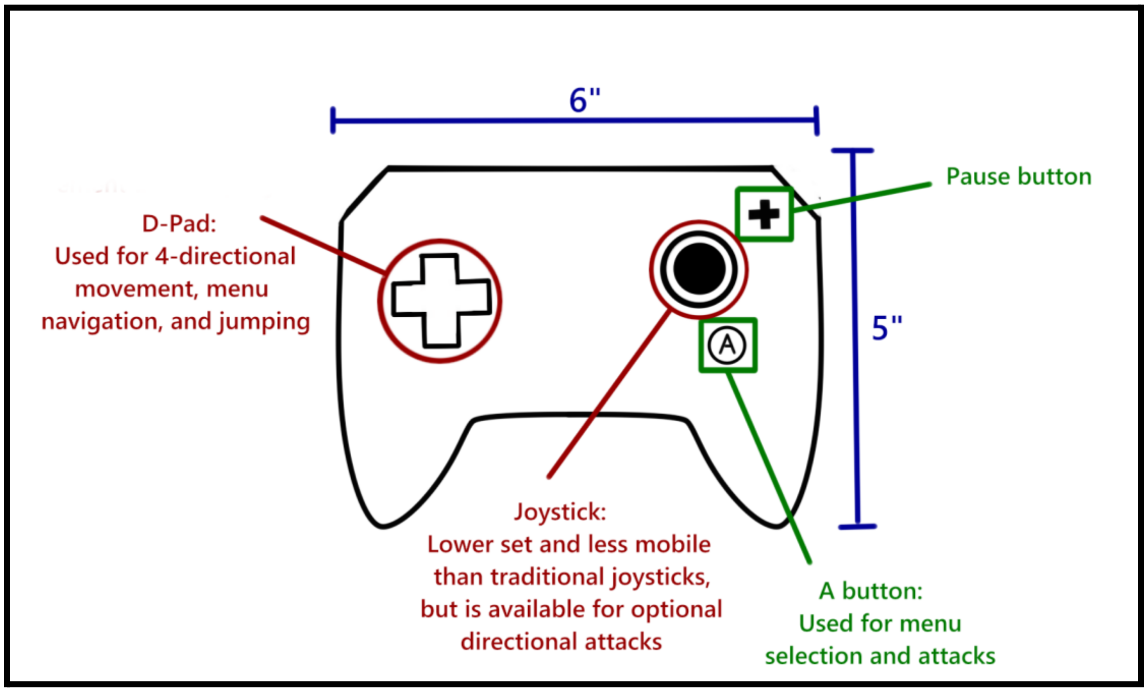

6"
5"
Pause button
D-Pad:
Used for 4-directional movement, menu navigation, and jumping
Joystick:
Lower set and less mobile than traditional joysticks, but is available for optional directional attacks
A button:
Used for menu selection and attacks

Figure 16: Design prototype of custom controller

Standards and Constraints

All C# programming conformed to the C# version integrated in Godot Mono v3.5. The software was developed so that the game reacts to user input in real time. Godot guidelines are followed for development and deployment by selecting currently supported API functions and embedding the resource pack (PCK) into the stand-alone executable. The game is constrained to accepting keyboard and mouse input to interact with the game world.

Static Code Analysis Report

We integrated an existing static code analysis tool (dotnet/code-analysis) for the C# programming language referenced by Microsoft in our existing GitHub Actions CI/CD pipeline. Every commit pushed to the remote repository will trigger the static code analysis job to run and report any detected errors in a Static Analysis Results Interchange Format (SARIF) and be uploaded as an artifact for logging and review. The GitHub action is set to load the Godot project's existing .csproj and .sln files for .NET tools to analyze the C# source code. If any code analysis violations occur, the build will break and the job will fail in the CI/CD pipeline. Our latest reports detected zero errors in the SARIF artifact from GitHub Actions.

Acknowledgments

We would like to acknowledge and thank our project advisor, Dr. Jeremiah Blanchard, for the immense amount of guidance, support, and kindness he has shown us throughout the course of our work. Thanks to Dr. Blanchard, our group was able to develop this game as our senior project and learn a great deal from the experience. This game was initially developed in Python in the Summer of 2022, and Dr. Blanchard kindly proposed that he supervise our continued game development for our senior project. Because of this opportunity, our members were able to learn from working on a long-term project, which was ported from Python to C#, using entirely different game engines.

We would also like to acknowledge and thank Andrew Watson, a former development team member. Andrew Watson worked on this project over the Summer of 2022 and was instrumental in the initial planning, design, and development of all aspects of the game. Without his work in the critical early stages, we could not have come this far in our development.

References

BANDAI NAMCO Studios, DOKIDOKI GROOVE WORKS. (2018, July 19). Taiko no Tatsujin: Nintendo Switch Version!. video game.

Corso, D. (n.d.). *A Review of Video Game Effects and Uses*. University of South Carolina. Retrieved September 6, 2022, from https://sc.edu/about/offices_and_divisions/research/news_and_pubs/caravel/archive/2014/2014-caravel-video-games.php

Game DeveloperStaff. (2019, March 27). Building a 'homebrew' video game console. Game Developer. Retrieved September 6, 2022, from https://www.gamedeveloper.com/design/building-a-homebrew-video-game-console

Knecht, P., & Reider, N. T. (2010). Introduction. In *Japanese demon lore*. essay, Utah State University, University Libraries.

Microsoft Learn. .NET-Related Github Actions. Dotnet-Bot. Retrieved December 7, 2022, fromhttps://learn.microsoft.com/en-us/dotnet/devops/dotnet-github-action-reference#net-code-analysis.

Nintendo. (1985, September 15). *Super Mario Bros.*. video game.

Nintendo. (1987, January 14). *Zelda II: The Adventure of Link.* video game.

Nintendo. (2014, September 14). *Super Smash Bros for 3DS*. video game.

Biography

Elizabeth Churinov

Elizabeth Churinov will graduate with a Bachelor's degree in Computer Engineering in May of 2023. During her time as a student, she achieved the opportunity to have a three month internship at a lab at Seoul National University in South Korea. This research internship was over the course of the summer of 2021, and was funded by the National Science Foundation. There, she focused on researching machine learning and creating datasets. Elizabeth has also studied abroad in Japan over the summer of 2022, where she experienced a different culture. Her interests include leadership, research, and game development. Her goals for employment after graduation include making enough money to fund future travels and eventually get a graduate degree.

Ricardo Correa Netto

Ricardo Correa Netto will graduate in December 2022 with a Bachelor's degree in Computer Science. As a student of the University of Florida's online program, most of his college career was limited to recorded and live video lectures. After experiencing a two-month study abroad in Japan over Summer 2022, he decided to experience life in Gainesville with his friends for his final semester. Through studying Japanese for the Summer abroad, Ricardo became interested in language and language learning. While he is still discovering his passions within Computer Science, his work on this project has piqued his interest in game development. Ricardo is set to enter the industry after graduation, and looks forward to discovering the opportunities available to him.

Chandler Fox

Chandler Fox is a 2022 graduate of UF with a Bachelor's degree in Computer Science. During his time at UF, he would develop several video games for his friends to play while grabbing any information about programming possible. He fell in love with traveling when he studied abroad in Japan, and his goal in life is to travel whenever he wants. The ideal next job is one where he is learning several different fields of programming simultaneously, while still having enough free time to program games for his loved ones. He is taking a job in UF's IT department to get some real job experience doing work that varies daily.

Chase Ruskin

Chase Ruskin is a fourth-year undergraduate at the University of Florida, set to graduate with his Bachelor’s degree in Computer Engineering in May 2023. He is interested in embedded systems, digital design, FPGAs, computer architecture, and low-level programming. In the past, he has designed and tested digital hardware systems in VHDL and has programming experience in Rust, C++, and Python. When he is not programming, he can be found on a run or in the water. He hopes to pursue graduate school focusing on FPGAs and hardware security.

Rachel West

Rachel West will graduate with a Bachelor’s degree in Digital Arts and Sciences from the University of Florida in December 2022 . During her time, she worked as an undergraduate peer mentor within the college of engineering and had the opportunity to study abroad in Japan in the Summer of 2022. Her passions within the major focused on game development and particularly 3D modeling and animation. After her undergraduate work is complete, Rachel plans to attend graduate school within the College of the Arts to explore augmented reality and virtual reality technologies. She hopes that after finishing her education, she can pursue a career either revolving around these modern technologies or 3D modeling.