

Oliver Wilkes
Nottingham College

Computer Science
Programming Project

Crazy Golf

2023/24

Contents

Analysis	4
Problem Identification	4
Clients	4
Computational Methods	4
Problem Recognition	4
Problem Decomposition	4
Divide and Conquer	5
Abstraction	5
Interview	5
First Interview	5
Questions	5
Responses	5
Analysis	8
Second Interview	8
Questions	8
Responses	8
Analysis	12
Existing Solutions	12
Features of the Proposed Solution	16
Limitations of the Proposed Solution	17
Requirements	17
Stakeholder Requirements	17
Design	17
Functionality	18
Hardware and Software	18
Measurable Success Criteria for the Proposed Solution	19
Design	20
User Interface	20
Main Menu	20
Options Menu	21
Main HUD	21
Stakeholder Input	22
Responses	23
Analysis	24
Improvements	24
Development	25
Stage 1: Setup & Menus	26

Main Menu	26
Options Menu	27
Audio Settings	28
Video Settings	29
Window Mode	29
Anti-Aliasing	31
Vertical Sync	33
Control Settings	34
Control Inputs	34
Textures	34
Buttons	39
Ongoing Testing	42
Controller Input Passthrough	42
Keyboard Input Loading	43
Controller Input Initial Focus	43
Saving Settings	43
Restructuring	44
File Saving	44
Video Settings	44
Audio Settings	45
Control Settings	46
Ongoing Testing	47
Anti-Aliasing Loading	47
File Loading	47
Audio Settings	47

Analysis

Problem Identification

In exam time, students can get stressed from revising for exams. Many students use video games as a way to escape from revision from time to time, but many games require too much of a time investment to be able to play for short periods of time. Multiplayer games can help during exam time as it can be a time to socialise during breaks while revising. Many recent games also require powerful computers to run the graphics requirements they meet, so I propose a game which has simpler graphics, shorter round times, and is multiplayer.

Clients

My users are a group of 16-19 year old students which sometimes play video games. Many of them prefer smaller games where you do not have to worry if you have time to finish the game/round. Many of them also have low powered laptops, so a game that is easier on the graphics is more enjoyable due to less lag.

Computational Methods

Problem Recognition

The first part of the problem is taking user input and translating it into in-game movement. This movement needs to be calculated on the course, and the ball has to interact with any obstacles in the way. Once this is done as single player, multiplayer would need to be implemented. This involves connecting other players to the game, and synchronising the game state between all players, while using minimal bandwidth. Players can also interact with each other via items, which involves player to player interactions.

Problem Decomposition

The different components the game involves are:

1. Connecting to the network to play multiplayer
2. Take user input and serialise it to send to the server
3. Process multiple players' movements on the server
4. Synchronise state to the client, which shows the ball(s) moving

Divide and Conquer

These steps can be solved on their own for the most part, as each component can be designed separately, and implemented modularly.

Abstraction

Many different abstractions can be done between these components. When movements are sent to the server, the server is not concerned with what buttons the user pressed, but what direction the ball will move in. This abstraction allows the server to process movement without having to worry about the input method.

Similarly, the client should not have to process collisions with obstacles, but just show the ball moving via instructions from the server. This abstraction allows all players to have synchronised states easily, reduces processing on the client, and makes cheating harder.

Interview

First Interview

Questions

The clients I have chosen to interview have played arcade-style games before. The aim of my first interview is to gather a general idea of mechanics and play styles that my clients enjoy. I will ask them questions about similar games they have played, and what they liked and disliked about them.

1. Have you played a multiplayer arcade-style game before?
2. If so, how many different games in this genre have you played?
3. What were your favourite parts of these games?
4. What were your least favourite parts of these games?
5. What would you like to see in a game like this?

Responses

Milan

1. Have you played a multiplayer arcade-style game before?

Yes, I have.

2. If so, how many different games in this genre have you played?

Probably something around 20. The new Discord activities, Golf With Your Friends, Ultimate Chicken Horse, etc.

3. What were your favourite parts of these games?

I particularly enjoy how these games facilitate a competitive environment while not being particularly frustrating to lose. Compared to real competitive games, it's easy to jump into a game like this with a group of friends and just have a good time trying to win, even if you end up doing poorly.

4. What were your least favourite parts of these games?

Generally, there's a very simple premise behind this type of game. After a couple hours of play it can feel like I've seen everything the game has to offer, which leads to me growing tired of such a game particularly quickly.

5. What would you like to see in a game like this?

5. This mostly relates to my answer to (4.), where I think some extra thought into replayability can go a long way. Something like impactful power-ups can drastically change the flow of a game, which can go a long way towards keeping it interesting.

Extra Comments

Another pretty cool example of this is Ultimate Chicken Horse, where the whole premise of the game is to have the players "create" the levels on the fly in a match. Each player gets to pick one of a few random items at the beginning of each round which they then place. Levels generally start off impossible to clear so players need to cooperate to make it winnable, and from there they start placing stuff to make it hard to clear. The goal of the game is to clear the stage while making others unable to clear it.

Alex

1. Have you played a multiplayer arcade-style game before?

Yes

2. If so, how many different games in this genre have you played?

At least 10

3. What were your favourite parts of these games?

Playing against/with friends -> competition / teamwork

4. What were your least favourite parts of these games?

When someone is a lot better than the rest of the group (skill gap)

5. What would you like to see in a game like this?

Skill levelling (chess.com has this when playing with friends, making it harder for one of the players)

Gwen

1. Have you played a multiplayer arcade-style game before?

Yes

2. If so, how many different games in this genre have you played?

About 5

3. What were your favourite parts of these games?

I like it when the rules are pretty simple but the game can be played in different ways based on the player

4. What were your least favourite parts of these games?

I don't like it when seemingly benign aspects of a game (i.e. character selection or locations on a map) give a player an unfair advantage over their competitors

5. What would you like to see in a game like this?

More collaboration between teammates

Extra Comments

I like Pico Park where several players (2+) have to learn and use the physics system of the game to their advantage, reach certain checkpoints, and use teamwork in order to complete a level. I want more of that in arcade-style games.

Enoki

1. Have you played a multiplayer arcade-style game before?

Yes

2. If so, how many different games in this genre have you played?

10-15

3. What were your favourite parts of these games?

The multiplayer aspect of it with different player interacting with each other in meaningful ways

4. What were your least favourite parts of these games?

The meta progression that some games have that give players unfair advantages over others

5. What would you like to see in a game like this?

Players being able to hijack other players' controls

Analysis

From those questions, I have gathered a general idea of what my clients would like to see in a game like this. They would like to see a game that is simple to understand, but has a lot of replayability. They would also like to see a game that has a competitive aspect, but is not frustrating to lose. They would also like to see a game that has a multiplayer aspect, but is not unfair to players that are new to the game. Every game should act almost ephemeral in a way, where the previous game does not affect the next game. This keeps it to the arcade style as standard arcade games do not have progression between games.

A co-operative aspect is something that Milan, Alex and Gwen mentioned in their responses. This could be something implemented as an extra game-mode in the game, but it would not be the main focus of the game. It would be something that could be played if the players wanted to, but it would not be forced upon them.

Second Interview

In this second interview, I asked my clients more specific questions about the mechanics of the game. I asked about the controls, the camera, the items and the co-op aspect of the game. This allows me to curate success criteria for my game, and a set of requirements that I can use to make my game.

Questions

1. How would you like to control your ball and camera?
2. Should there be items/power-ups, and if so, what should they do?
3. How should the co-op part of the game work?
4. What kind of music would you like in the background?
5. How should you be able to join a multiplayer game (invite code, add via in-game friend, etc.)?

Responses

Milan

1. How would you like to control your ball and camera?

I think the most intuitive camera setup would be

- The camera is centred onto the ball.
- You can zoom in and out using the scroll wheel,
- You can click and drag with middle mouse/right click to rotate the camera around the ball,
- You can click and drag with left click to shoot the ball at different levels of power.

(Essentially the same as Golf With Your Friends/Putt Party) It could be interesting to experiment with some control fine-tuning like gear effect (think Wii Sports where you can shoot the ball along different trajectories by introducing effect), though that probably only makes any real amount of sense in sufficiently large courses. Lastly, it may be nice to have a way to detach the camera from the ball to scout ahead and discover the course.

2. Should there be items/power-ups, and if so, what should they do?

Given the focus of a “multiplayer arcade-style” game, I do believe there should be power-ups. I feel like the main focus for these power-ups should be affecting every player in the game. Essentially, anything that isn’t downright frustrating but is still sufficient to put a wrench in the works for the “victims” should be good. An example of this could be a wind effect that can influence every player’s next shot(s).

3. How should the co-op part of the game work?

As far as co-op is concerned, I feel like it’s important that players get to play on the same hole simultaneously, especially so when taking power-ups into account. You could then make a decision between whether you want there to be distinct turns (i.e. each player gets to do one swing, then waits for all other players to have finished their swing, until the hole is cleared). Introducing turns in this fashion could make power-ups more impactful as you could have them stay active for a number of turns, making players unable to just wait it out.

4. What kind of music would you like in the background?

You could lean into the arcade side of things and make peppy music you would hear in actual arcade games. Something chiptune-like could also work depending on the graphical style of the game. Overall, as long as the music matches the creative direction of the game, I think a lot of different styles could work. An interesting example here is Golf With Your Friends. This game

heavily leans into creative map design with an exploration element, and this is reflected well in the music. The bgm for that game could actually fit pretty well into an RPG-style game.

5. How should you be able to join a multiplayer game (invite code, add via in-game friend, etc.)?

I think at a base level, using invite codes is fine, as long as groups aren't disbanded after each game. Entering an invite code once for a session of play with friends is more than acceptable. If the game were to be launched on a platform such as Steam, integrating with the platform's built-in friends system would be a nice addition. For a standalone game, however, I don't think a friends system is really a hard requirement.

Alex

1. How would you like to control your ball and camera?

I would like to control the ball with the mouse, and the camera with the keyboard.

2. Should there be items/power-ups, and if so, what should they do?

Yes, not sure.

3. How should the co-op part of the game work?

A few ideas:

1. One person can edit the course, other one shoots the ball.
2. One player moves the camera, one controls the ball
3. One player has a camera locked onto the ball but can't see where it's going to hit, other player only has a top down view but can see where the ball hits

4. What kind of music would you like in the background?

Tetr.io battle music

5. How should you be able to join a multiplayer game (invite code, add via in-game friend, etc.)?

Invite code / link and friends seem reasonable (so you can play with "strangers" and friends)

Gwen

1. How would you like to control your ball and camera?

The camera should loosely follow the ball, if 3d the ball should have a 'forward' direction that the camera should follow behind

2. Should there be items/power-ups, and if so, what should they do?

Items/power ups should be available but not necessary to complete a level, they should essentially give the player ability modify the game difficulty

3. How should the co-op part of the game work?

There should be sections of the game that require multiple players to be in different places to achieve a goal

4. What kind of music would you like in the background?

There should be a quiet ambient music at all/most times and louder music based on events that happen in-game

5. How should you be able to join a multiplayer game (invite code, add via in-game friend, etc.)?

Players should be able to send invitations from in a game to other players they want to play with, for the receiver to accept or deny

Enoki

1. How would you like to control your ball and camera?

Similar to putt party

2. Should there be items/power-ups, and if so, what should they do?

Yes, ideally more focused on affecting others rather than yourself, more extreme stuff is always more fun

3. How should the co-op part of the game work?

Potentially either have them take turns controlling the ball or have one ball that's not controlled by anyone but everyone has to hit it (like in pool) until it gets in the hole

4. What kind of music would you like in the background?

Anything works

5. How should you be able to join a multiplayer game (invite code, add via in-game friend, etc.)?

In game friend is more steps, a code / url is easier, more transient and more accessible

Analysis

From these questions, I now have a more specific set of requirements for my game. I will use these requirements to create a set of success criteria below.

Question 1 has shown that putt party provides an ideal control scheme, but a possible keyboard control as suggested by Alex could be more accessible too.

Question 2's results explain how items provide a different experience for every run of the game, and Enoki suggested that power ups that affect others more than yourself are more fun.

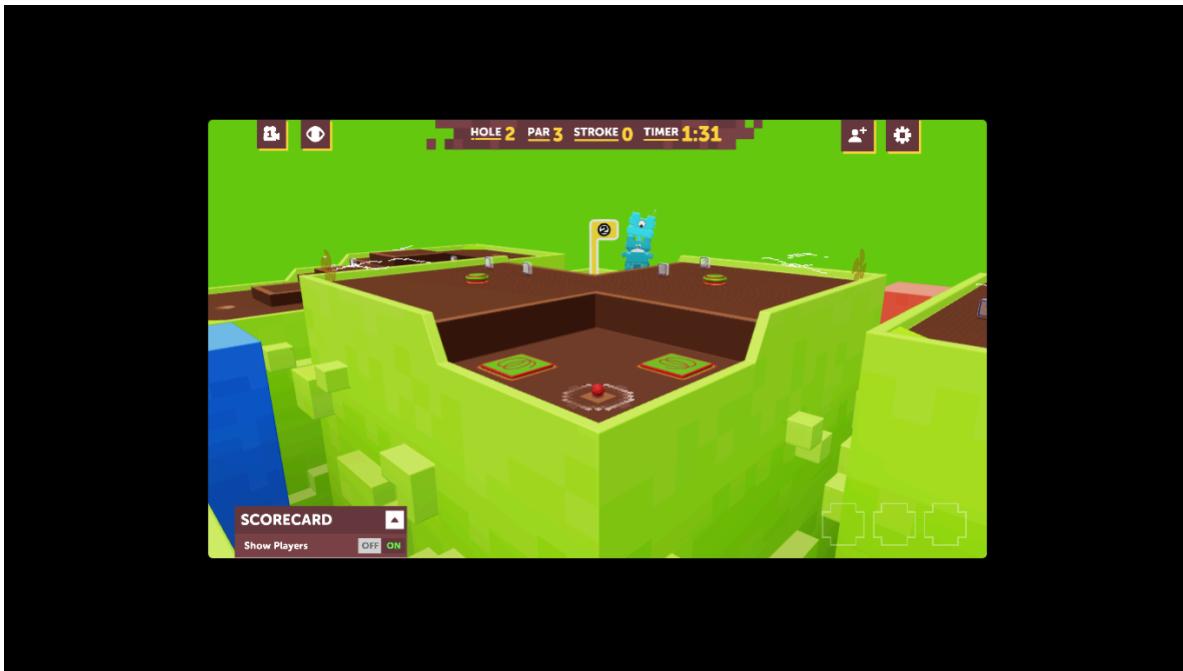
Question 3 is still a bit undecided, but Milan suggested that players should be able to play on the same hole simultaneously, and Gwen suggested that there should be sections of the game that require multiple players to be in different places to achieve a goal.

Question 4's results can be combined into one idea. Arcade-like ambient music, with more intense music for events such as when time is running out.

Question 5 demonstrates a simple URL/code is good enough to join a multiplayer game, and Enoki suggested that an in-game friend system is not necessary but a nice to have.

Existing Solutions

Putt Party



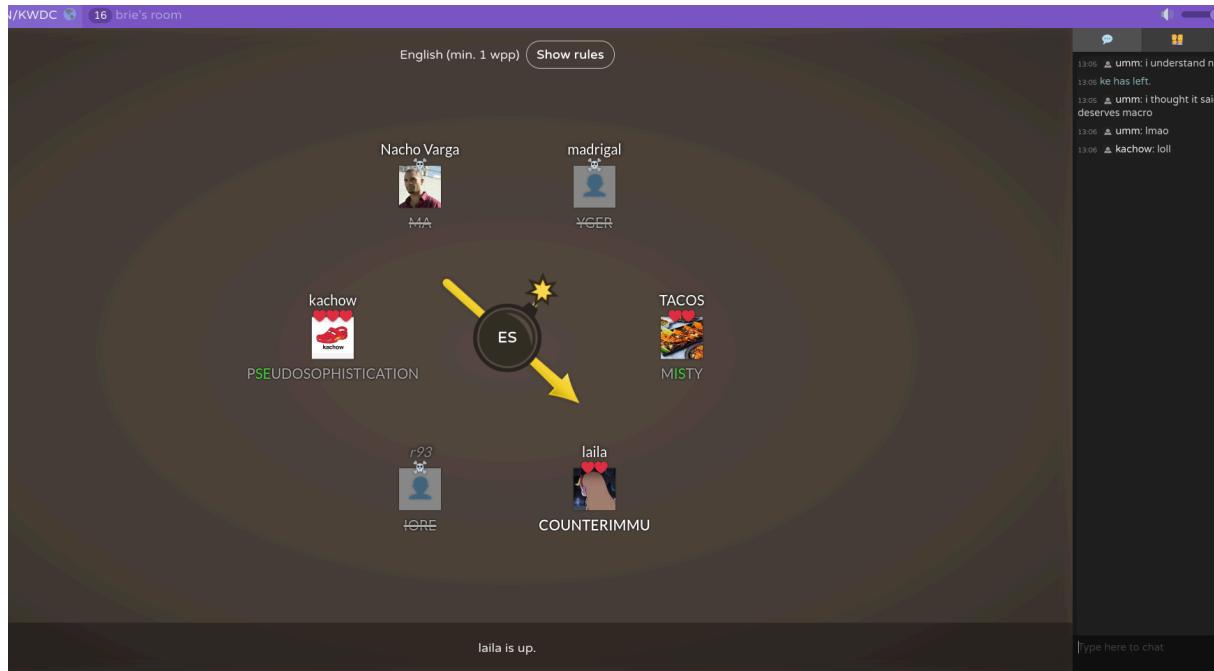
Putt party is a Discord “activity” built in to voice channels. This is an example of what my game will be like, as it is arcade-style and multiplayer. It can be played many times with different results as there are items in the many courses, and it is not frustrating to lose at. It implements multiplayer very well as you just join the same voice channel as your friends. It is simple to understand as the controls are just drag and point with the mouse, and it is not too graphically intensive. It is also not too long, as each hole has a time limit.

One of the downsides though is that you all have to have Discord to be able to play together. The window is quite small as it is a small section of the voice channel overview, so is difficult to see sometimes, especially on smaller screens like laptops.

Parts of Putt Party I Can Apply

Multiple of my interviewees have mentioned the controls of putt party, or similar controls. I will take inspiration from this as a way to control the ball.

Bomb Party

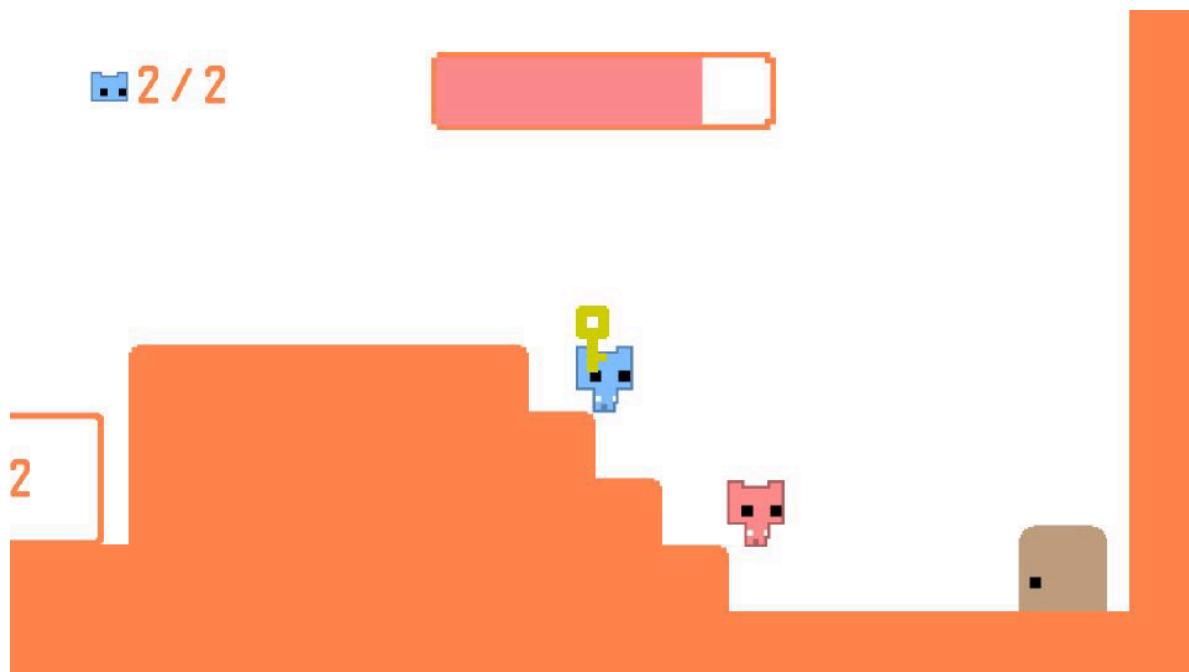


Bomb Party is a small party game where you have to type words based on a prompt. It is fast-paced, multiplayer, and has a competitive aspect. It is simple to understand as the controls are just typing, and it is not too graphically intensive. It is also usually not too long as you can configure turn limits and difficulty. The multiplayer aspect is implemented with a 4 letter code that can be entered or used via a link (`domain.tld/CODE`).

Parts of Bomb Party I Can Apply

The multiplayer aspect of Bomb Party is something I can apply to my game. It is simple to understand and easy to use, so I will use a similar system in my game. This system was supported by Enoki, Alex and Milan in their responses.

Pico Park



Pico Park is a co-operative multiplayer game where you have to work together to complete levels. It also has simple graphics and has short levels, which can get quite difficult as it requires co-ordination of multiple people. It is simple to understand as the controls are just arrow keys and space.

Parts of Pico Park I Can Apply

I will consider the ways they make players work together and see how they can fit in a golf-style game. Gwen suggested to look at this game as it has a co-operative aspect.

Golf With Your Friends



Golf With Your Friends is a 3D multiplayer golf game. It has a bit more complex graphics compared to the other games, but still simple to understand. It has a lot of different courses, and a lot of different game modes. It has a lot of replayability as there are many different ways to play the game. It is also not too long as each hole has a time limit. It has both online and local multiplayer support. There are no items but it has the concept of choosing where to hit your ball, and the power of the shot which makes the controls more complex.

Parts of Golf With Your Friends I Can Apply

As this is a 3D game, I can take into consideration how the camera follows the ball on this, which was mentioned by Milan and Gwen in their responses. I can also look at how local multiplayer works too to consider adding.

Features of the Proposed Solution

My solution will be a game with a main menu to select whether to host a room, join a room or change settings. When entering a round, there will be a heads-up display with information about the game such as the time left, scoreboard, and the current hole. The controls will be drag and point, the camera will follow the ball but still be possible to rotate with the mouse or keyboard, and the power of the shot will be controlled by how far you drag the mouse. There will be items that affect other players, and there will be a co-operative aspect to the game as a separate game mode. There will be a simple URL/code to join a multiplayer game, and there will be arcade-style ambient music.

Limitations of the Proposed Solution

My game would require a Wi-Fi connection. It could use a LAN for local play, but not bluetooth. This is a limitation as school/college Wi-Fi can be unreliable or not accessible at times, and some students may not have access to Wi-Fi at home.

Some visual impairments may be a limitation too. As the game will be 3D, it may be difficult to interpret with a visual impairment. I will try to make the game as accessible as possible with changes such as a high contrast interface, but it may not be possible to make it accessible for all visual impairments.

The game will be controlled with a keyboard and mouse. This is a limitation as some students may not have access to a mouse, or may not be able to use a mouse due to a physical impairment. This would not be something I can solve completely initially, but a low amount of keyboard inputs would mean they can be remapped to a controller or an accessible input device.

The game could also act as a distraction from revision or school work. This is a limitation as it is not the intended purpose of the game. This could be solved by having a setting to lock the game at certain times of the day, but this could be bypassed by changing the system time.

Requirements

Stakeholder Requirements

Design

Requirement	Explanation
Main menu	The game should have a main menu so the player can choose what to do.
Full screen and windowed	The option of full screen is useful for smaller screens like laptops, but windowed is useful for large screens like desktops.
Simple to understand	The game should be simple to understand so it is easy to pick up and play.

Simple graphics	The game should have simple graphics so it is not too graphically intensive.
Simple controls	The game should have minimal controls so it can be learned straight away.
Following camera	The camera should follow the ball so the player can see where they are going.

Functionality

Requirement	Explanation
High Contrast Colour Palette	The game should have a high contrast colour palette so it is accessible to people with visual impairments, whilst also looking playful too.
Drag and point controls	The controls should be drag and point so it is simple to understand.
Power of shot controlled by drag	The power of the shot should be controlled by how far the player drags the mouse.
Items that affect other players	There should be items that affect other players so the game is more interesting.
Co-operative aspect	There should be a co-operative aspect to the game so players can work together if they wish.
Simple URL/code to join a multiplayer game	There should be a simple URL/code to join a multiplayer game so it is easy to join.
Arcade-style ambient music	There should be arcade-style ambient music so it is not too distracting.

Hardware and Software

Requirement	Explanation
Keyboard and mouse	The game should be playable with a keyboard and mouse.

Minimum ...	The game should be playable on a minimum of ... (pc specs, todo, no realistic target right now).
Windows >=10, macOS >=13.1, Linux x86_64	These are the operating system versions that the game should be at minimum playable on.

Measurable Success Criteria for the Proposed Solution

Criteria	How to get evidence
Clear main menu	Screenshot of the main menu and user feedback.
Full-screen and windowed options	Screenshot of the application being full-screen and windowed.
Simple to understand controls	Screenshot of the controls menu and user feedback of the controls.
In-game camera follows the ball	Screenshot which shows the ball is visible at all times.
High contrast colour palette	Screenshot of the game interface and relevant contrast ratios.
Drag and point controls	Screenshot of the ball being controlled.
Power of shot controlled via dragging the ball	Screenshot of the ball being controlled.
Items that affect other players	Screenshot of the items in effect during gameplay.
Multiplayer functionality	Screenshot of game with multiple players
Co-operative aspect	Screenshot and explanation of the co-operative aspect.
Simple URL/code to join a multiplayer game	Screenshot of the URL/code being used.
Arcade-style ambient music	The code for the music, and where it came from.
Keyboard and mouse controls	Screenshot of the controls menu.
Includes a settings menu	Screenshot of the settings menu.

Simple graphics	Screenshot of the game, and framerate/GPU usage statistics with user feedback.
Scoreboard	Screenshot of the scoreboard.
Time limit	Screenshot of the time limit.
Configurable controls	Screenshot of the controls menu.

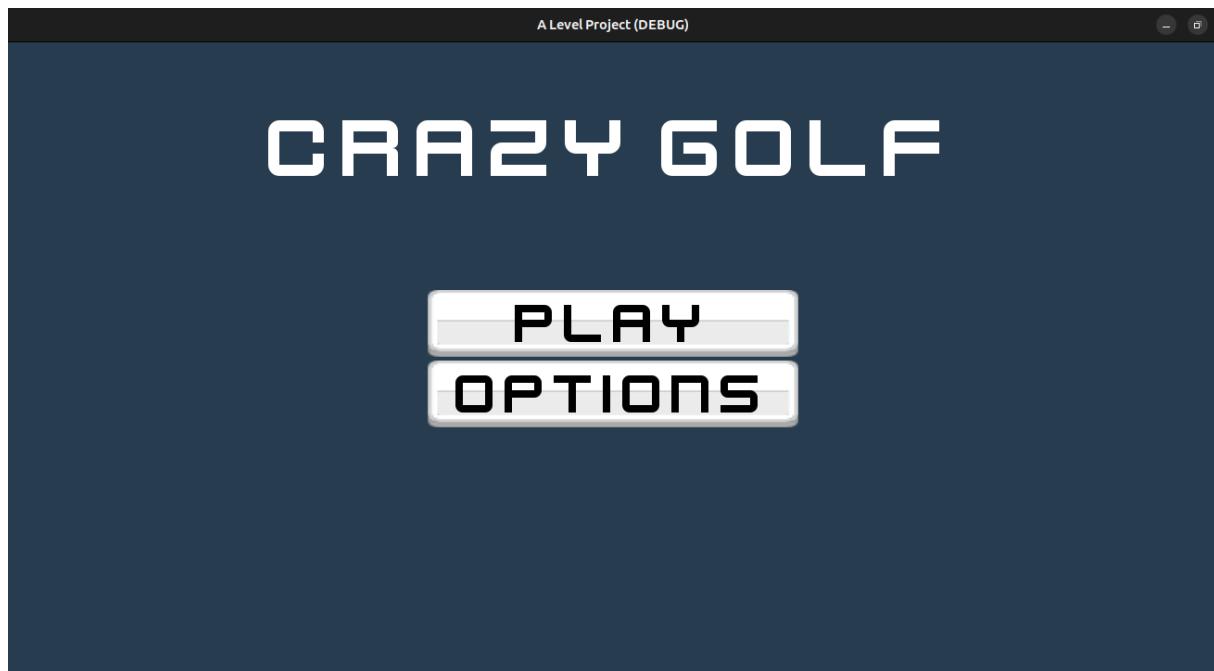
Design

Key points:

- Interface design
- Program structure
- Algorithms - validation
- Features
- Key variables/data structures/classes
- Testing approach

User Interface

Main Menu



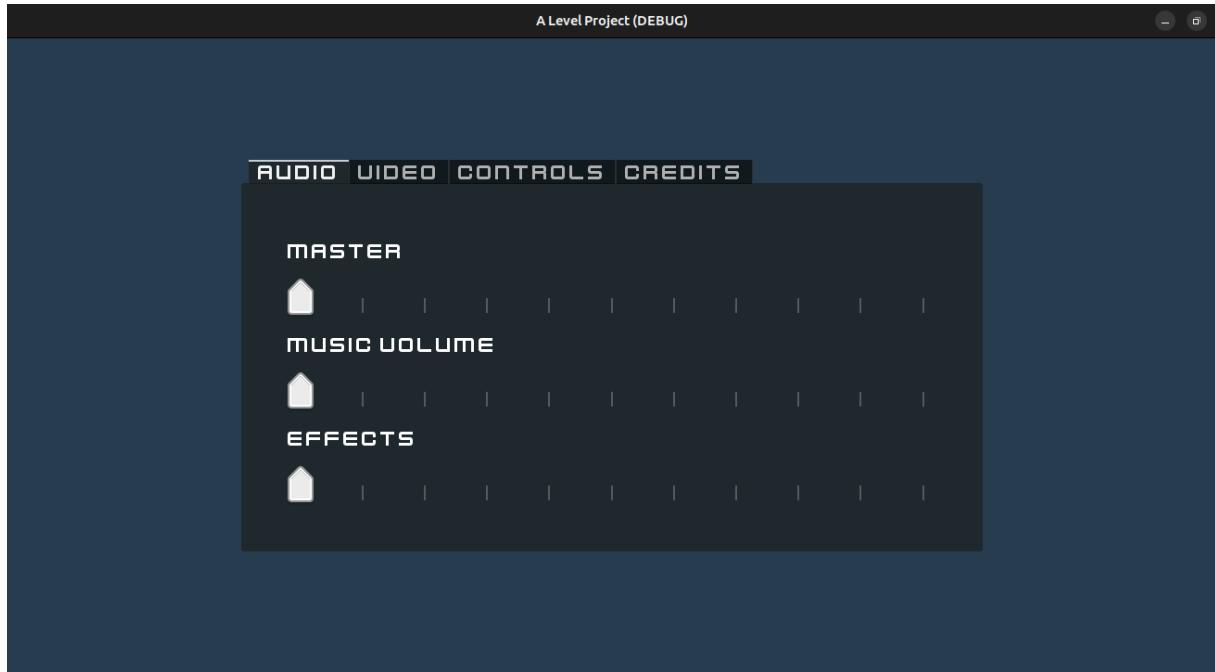
This is the main menu the user sees when starting the game. There are only two main clear options on if the user is ready to play or if they want to change any settings. The background is a placeholder and may be replaced with a preview of one of the levels, to show the player what the game is like before

they play. The options button will give the player a list of settings they can change for their own experience, such as graphics settings and controls. The buttons are in the center of the screen, with clear and large labels to let the player know what these buttons do.

Links to success criteria:

- Clear main menu
- Simple graphics

Options Menu

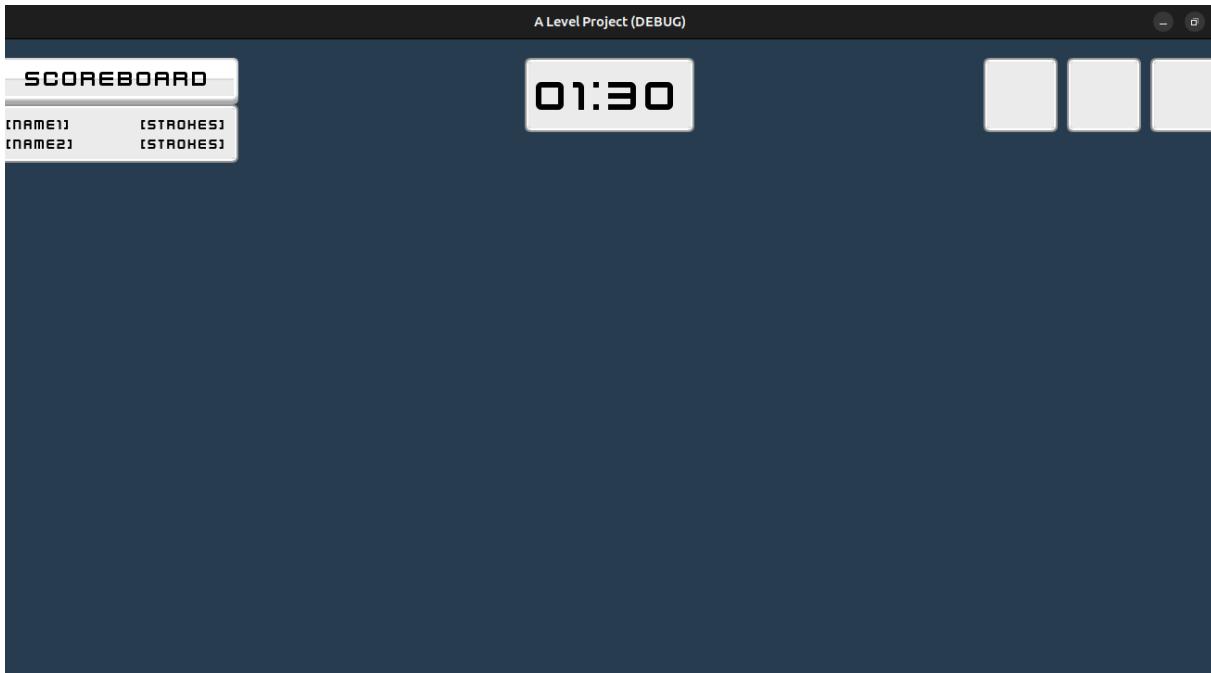


This is the menu which is shown after selecting the options button on the main menu or pause menu. This is where the player can change settings such as graphics settings, controls, and audio settings. There are tabs at the top for each category of setting, with clear and concise labels.

Links to success criteria:

- Full-screen and windowed options
- Simple to understand controls
- Keyboard and mouse controls
- Includes a settings menu

Main HUD



This is the heads up display that will display in the main game. It shows the scoreboard of all the players (which can be collapsed as it could get big with many players). It also shows the time left on the hole, and the player's inventory of items.

Links to success criteria:

- Scoreboard
- Time limit

Stakeholder Input

I sent the following message to my stakeholders for feedback on the user interface designs.

"Hi,

Attached are some screenshots of the designed layout of the menus for the game. These demonstrate the layout and roughly what they will look like in the game. They are the main menu, the options menu, and the main HUD in the game. The buttons are large to easily understand and read, with clear labels. The options menu has tabs at the top to easily navigate between different settings. The HUD is simple with everything at the top, with a collapsible scoreboard for if it gets in the way.

I would love to hear your feedback on these or any other comments you have.

Thanks,

Oliver"

Responses

Enoki

IMO the buttons look a bit weird on the corners due to the rounding of the inner sprite, and the font doesn't differentiate between H and K. I personally prefer the slider to be a bit higher, but that is not necessary. Besides that there isn't much I'd comment on.

Gwen

The scoreboard could have the label "toggle scoreboard" to demonstrate it is toggleable, but that is the only thing I could find after some deliberation. It looks really good and you don't have to implement that change but that's my bit of feedback.

Alex

- The individual slider markers are a bit hard to see for me, they need a bit more contrast / need to be more bright
- Scoreboard content text seems a bit small to parse at a glance, especially because it is collapsible I would make the text bigger
- Nitpick: Why is it "MUSIC VOLUME" but not "MASTER VOLUME"

I like the big buttons and large text and the very central timer

Milan

The overall placement of HUD elements looks fine to me, though it would be nice to have some clear indication of how to access settings while in-game, perhaps through a "pause" menu.

I feel like a button to quit the game is missing from the main menu. While it isn't strictly necessary, it can be convenient for certain users, for example those playing with a gamepad. (controller support is cool!!! [video referring to accessibility advantages of controller support])

For the options menu, I like the layout with navigable menus at the top and the actual options laid out underneath. I think the sliders specifically would be improved a little if the actual slider bar was visible too, instead of only the ticks.

The overall styling seems fine, if a bit plain, which is to be expected for a draft version. As long as this is updated when the overall style of the game is

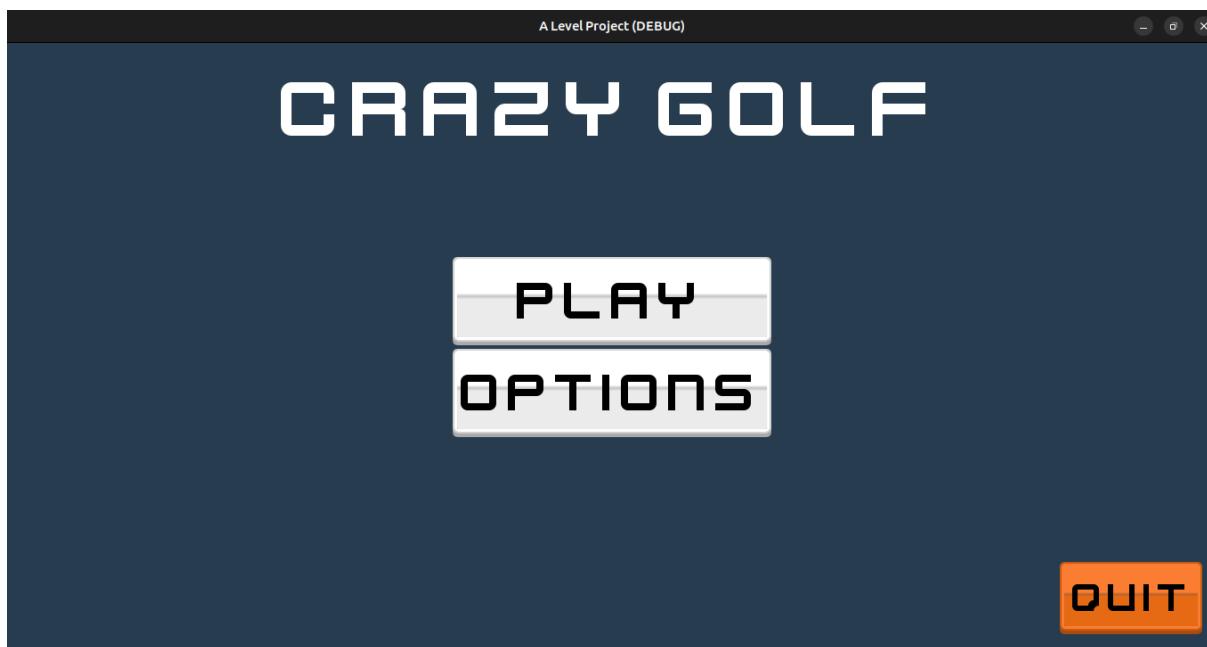
decided upon, I think the UI should work well. Lastly, I think the font could do with some modification, as the V looks a lot like a U. In context this isn't an issue (e.g. it's obviously "Video" and not "Uideo"), but I do personally find it a bit awkward to read.

Analysis

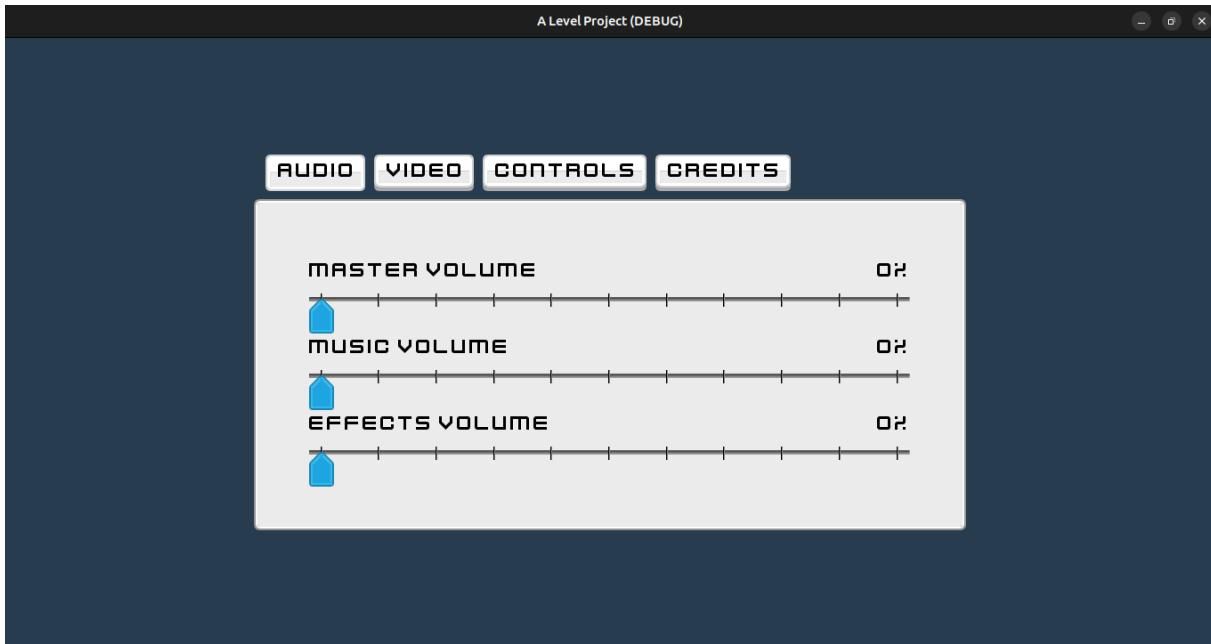
The feedback I received was generally positive, with some minor changes suggested. The following changes will be made to the designs based on the feedback:

- The inner corner of the buttons will be made more square to make the buttons look less weird.
- The font will be modified to differentiate between H and K, U and V.
- The slider markers will be made more bright to be easier to see.
- The scoreboard content text will be made bigger.
- The settings will be accessible through a pause menu with a physical button in the corner (as well as the escape button).
- A button to quit the game will be added to the main menu.

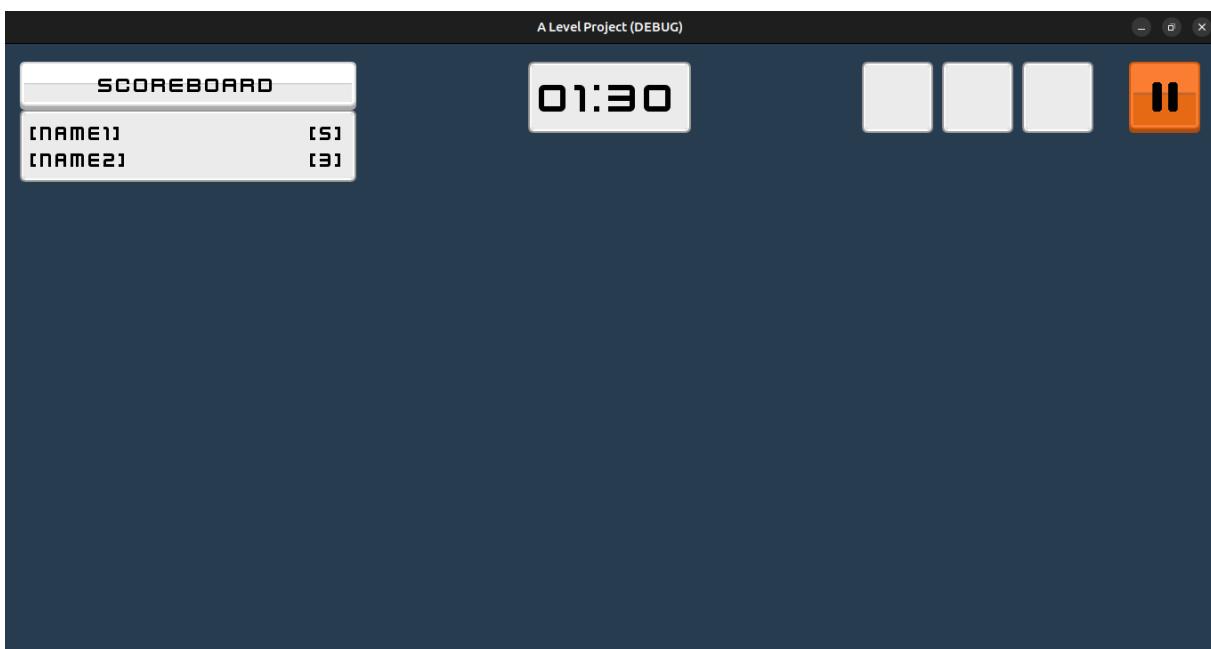
Improvements



There is now a button to quit the game in the main menu.



The sliders are now more visible, and the font has been changed to differentiate between H and K, U and V.



The scoreboard content text is now bigger, and there is a physical button to access the pause menu.

Development

The first stage of development is about creating the user interfaces. This includes linking the buttons to the different interfaces, and adding the relevant settings to the options menu.

Stage 1: Setup & Menus

Main Menu

In the main menu, the buttons are added via Godot's 2D editor, and each button is linked to a central script via signals. These signals are responded to below. Clicking the quit button sends a request to the `SceneTree` to quit.

The `play_scene` and `options_scene` are exported properties, so they can be set in the editor. This reduces coupling of the scenes and means the path of those scenes can change without issue. These `PackedScenes` are used to change the current scene to the play or options scene.

```
# main_menu.gd
extends MarginContainer

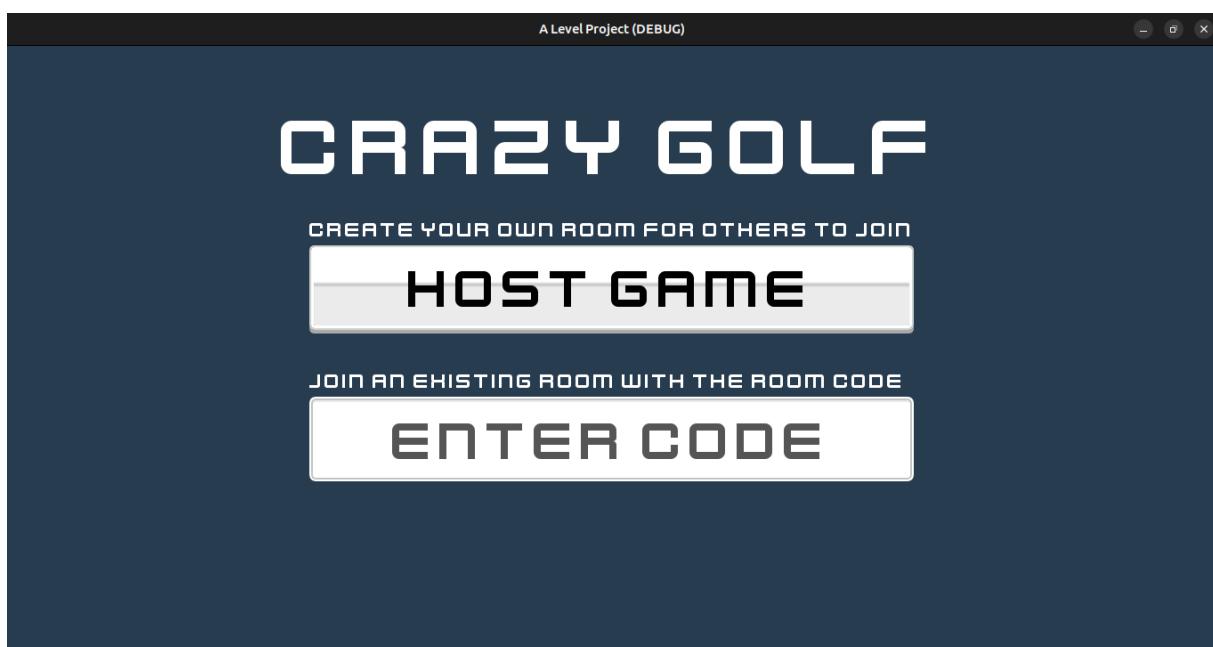
# Take in scenes as an exported property, to reduce coupling.
@export var play_scene: PackedScene
@export var options_scene: PackedScene

func _on_quit_button_pressed() -> void:
    # `get_tree()` gets the `SceneTree` which manages the game loop.
    get_tree().quit()

func _on_options_button_pressed() -> void:
    get_tree().change_scene_to_packed(options_scene)

func _on_play_button_pressed() -> void:
    get_tree().change_scene_to_packed(play_scene)
```

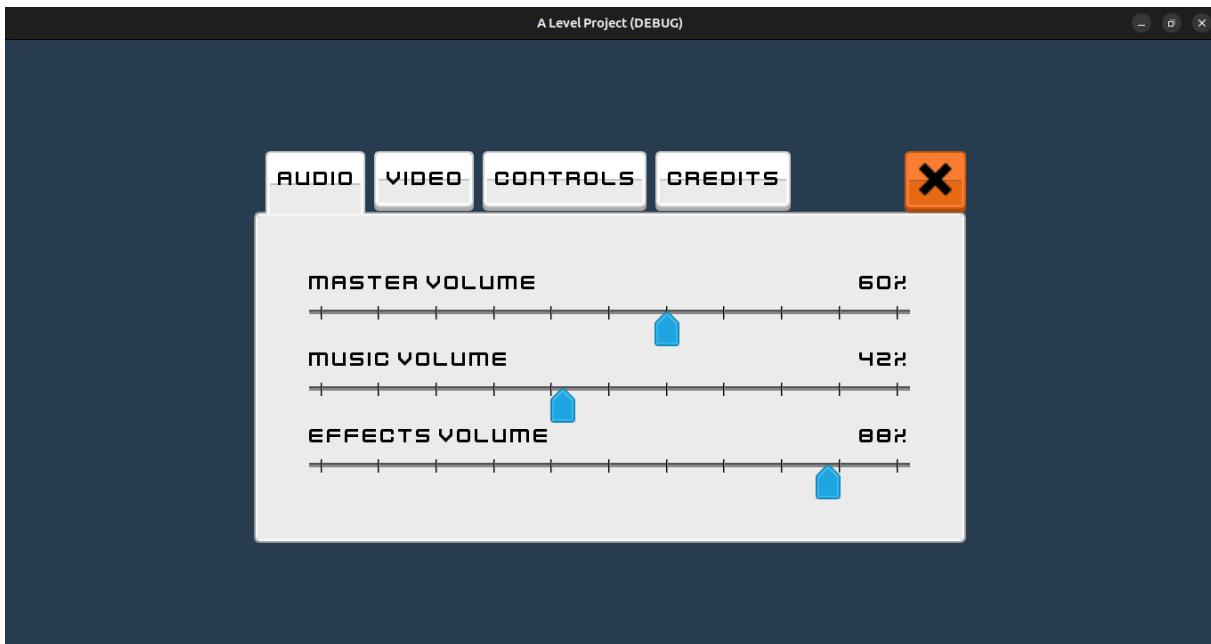
The `play_scene` looks like this:



Options Menu

The options menu is also designed with the Godot 2D editor. The tabs are nodes underneath a TabContainer which handles switching the tabs. I added an exit button on the top right to go back to the previous scene - which can be either the main menu or pause menu. As the scene to switch back to depends on the previous scene, this is handled via a global script.

The tabs have also been extended downwards when selected so it is clearer as to if they are selected.



The following script is a global autoload ‘singleton’ script which holds the previous scene. This is used to determine which scene to switch back to when the exit button is pressed.

```
# global.gd
extends Control

@ready var global: Global = $"/root/Global"

func _on_close_button_pressed() -> void:
    # global.previous_scene could be either options or pause, this is set
    # before switching to options.
    get_tree().change_scene_to_file(global.previous_scene)
```

This is used in `main_menu.gd` (and soon to be used in `pause_menu.gd` too) to set the previous scene before switching to the options scene. This is so the options scene knows which scene to switch back to when the exit button is pressed.

```

# main_menu.gd
@ready var global: Global = $"root/Global"

# [...]

func _on_options_button_pressed() -> void:
    var tree := get_tree()
    global.previous_scene = tree.current_scene.scene_file_path
    tree.change_scene_to_packed(options_scene)

```

In the options menu, the close button retrieves the previous scene from the global script and switches back to it.

```

extends Control

@ready var global: Global = $"root/Global"

func _on_close_button_pressed() -> void:
    # global.previous_scene could be either options or pause, this is set
    # before switching to options.
    get_tree().change_scene_to_file(global.previous_scene)

```

Audio Settings

Each volume slider is a child of a parent volume slider scene, which contains the slider, the label which contains the volume, and the script to handle the volume.

`audio_bus_name` is different per slider, so it is set per slider in the editor via an exported property. `audio_bus_index` is a unique number relating to this audio bus, used in the `AudioServer` API.

`_ready` is ran whenever the script is loaded. It retrieves the current bus volume from the `AudioServer` and sets the slider and label to that value. The `value_changed` signal is emitted whenever the slider is moved, and the `set_bus_volume_db` function is called to set the volume of the bus.

Both of these require converting to/from decibels, as the `AudioServer` API uses decibels for volume and that is a logarithmic scale.

```

extends VBoxContainer

@export var audio_bus_name: StringName
# The index of the audio bus in all buses.
@ready var audio_bus_index := AudioServer.get_bus_index(audio_bus_name)
@ready var value_node: Label = %Value
@ready var slider: HSlider = %Slider

func _ready() -> void:
    # Retrieve existing volume and set that on the slider and label.
    var db := AudioServer.get_bus_volume_db(audio_bus_index)
    var percentage := db_to_linear(db)
    var value := roundi(percentage * 100)

```

```

value_node.text = str(value) + "%"
slider.value = value

func _on_slider_value_changed(value: float) -> void:
    # Godot uses decibels, which is logarithmic. This function converts a number
    # from 0-1 into decibels (-80 to 24dB)
    var db := linear_to_db(value / 100)
    AudioServer.set_bus_volume_db(audio_bus_index, db)
    value_node.text = str(value) + "%"

```

Video Settings

The video settings I have decided to implement, to keep it simple, are:

- Window mode (windowed/fullscreen)
- Anti-aliasing (MSAA)
- Vertical-sync (VSync)

Window Mode

The first setting to implement is the window mode - whether the window is windowed or fullscreen. For this I asked one of my stakeholders which I know uses multiple (virtual) monitors on Windows 10 **and** Linux. I asked if they would like a borderless windowed fullscreen option, and they said they would like it. This is because it is more consistent across multiple monitors and is generally better for alt+tabbing especially in Windows.

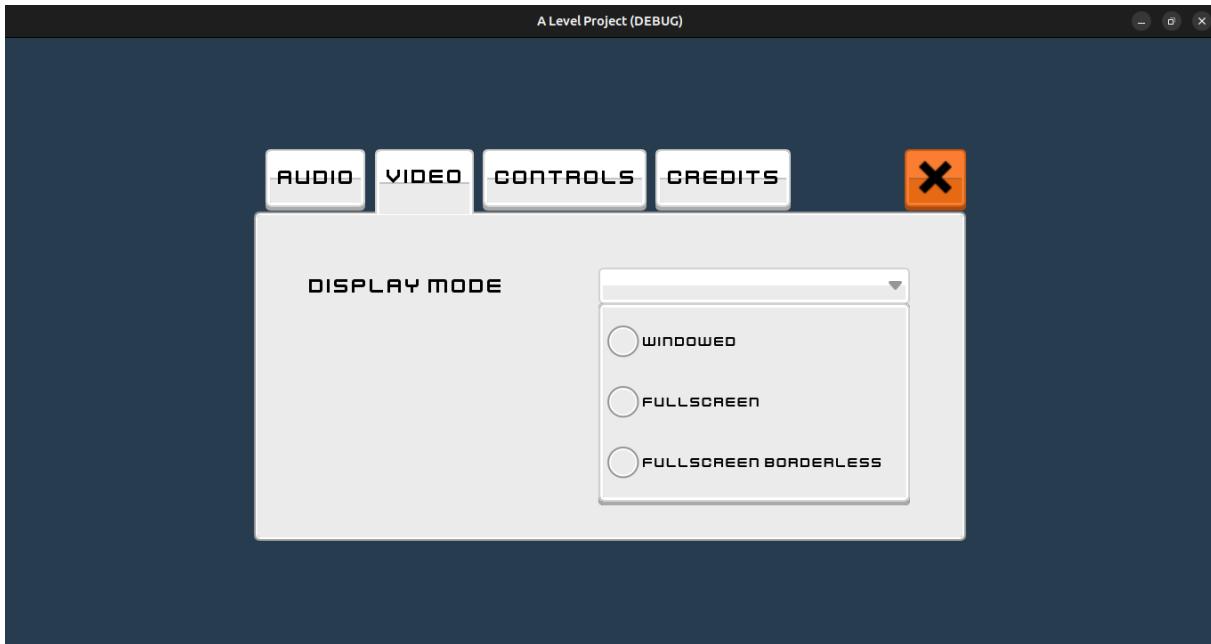
```

# video.gd
extends MarginContainer

@ready var display_mode: OptionButton = %DisplayMode

func _on_display_mode_item_selected(index: int) -> void:
    # 0: Windowed
    # 1: Fullscreen
    # 2: Fullscreen Borderless
    if index == 0:
        # Windowed windows have a border and this sets it to maximised for a
        # consistent size when switching away from fullscreen.
        DisplayServer.window_set_flag(DisplayServer.WINDOW_FLAG_BORDERLESS, false)
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_MAXIMIZED)
    elif index == 1:
        DisplayServer.window_set_position(Vector2i(0, 0))
        # Exclusive fullscreen has a lower overhead as it usually avoids
        # the display compositor.
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_EXCLUSIVE_FULLSCREEN)
    elif index == 2:
        # Fullscreen borderless is a full screen sized window without a border.
        # It usually works better with multiple monitor setups when alt+tab/esc
        # in windows.
        DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_MAXIMIZED)
        DisplayServer.window_set_size(DisplayServer.screen_get_size())
        DisplayServer.window_set_flag(DisplayServer.WINDOW_FLAG_BORDERLESS, true)
        DisplayServer.window_set_position(Vector2i(0, 0))

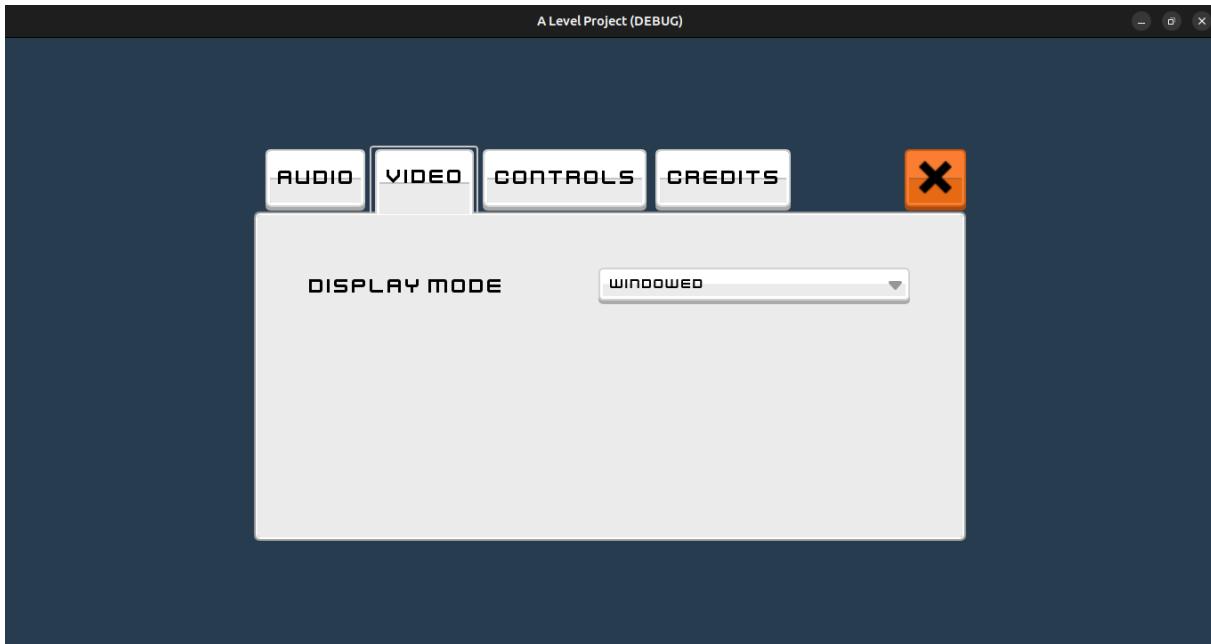
```



This only shows the dropdown, where selecting each option will change the window mode. The next thing to do is setting the initial value of this dropdown so it shows the currently selected mode.

This defaults to assuming the window mode is windowed. If it finds that it is fullscreen, then sets it to fullscreen. If it finds that the window is borderless and windowed (maximised), then it sets it to fullscreen borderless.

```
# video.gd
func _ready() -> void:
    # 0: Windowed
    # 1: Fullscreen
    # 2: Fullscreen Borderless
    var mode := DisplayServer.window_get_mode()
    var borderless := DisplayServer.window_get_flag(DisplayServer.WINDOW_FLAG_BORDERLESS)
    var id := 0
    if mode == DisplayServer.WINDOW_MODE_EXCLUSIVE_FULLSCREEN:
        id = 1
    elif mode == DisplayServer.WINDOW_MODE_MAXIMIZED and borderless:
        id = 2
    display_mode.select(id)
```



Anti-Aliasing

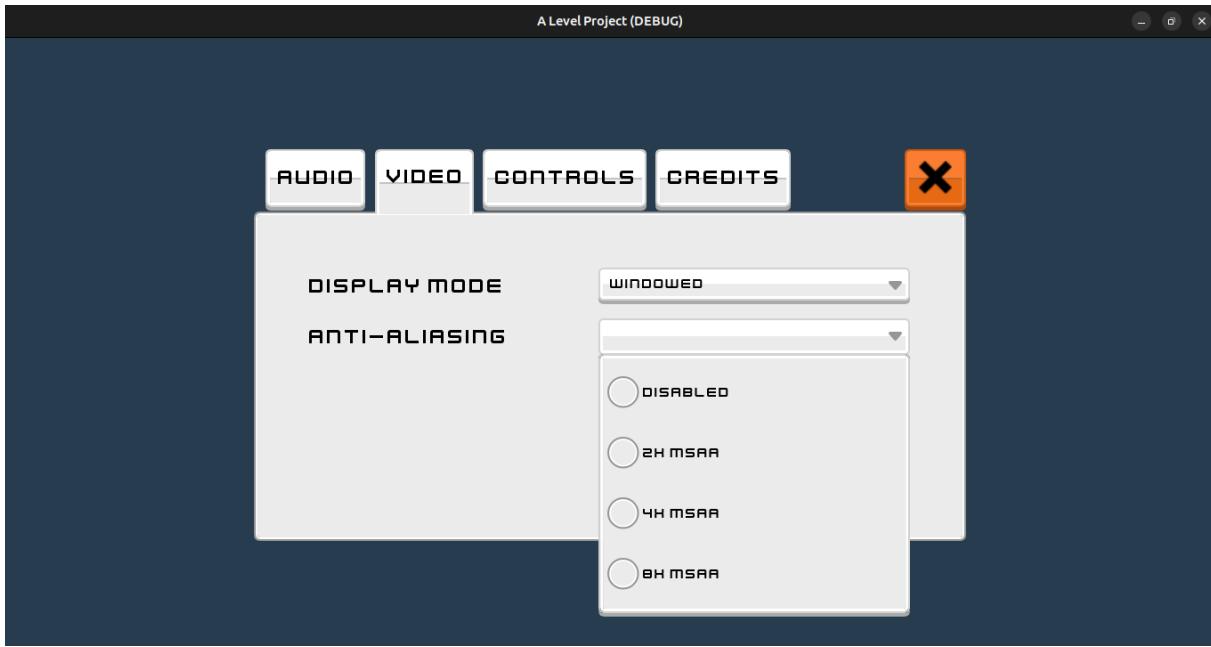
The next setting to implement is anti-aliasing. This keeps the 2D MSAA and 3D MSAA settings equal, as users are only going to enable anti-aliasing if their graphics card can handle it. The script is fairly simple, it sets the anti-aliasing quality in the project settings to the selected value.

```
# video.gd
# settings paths for anti-aliasing.
const ANTIALIASING_2D = &"rendering/anti_aliasing/quality/msaa_2d"
const ANTIALIASING_3D = &"rendering/anti_aliasing/quality/msaa_3d"

@onready var anti_aliasing: OptionButton = %AntiAliasing

# ...

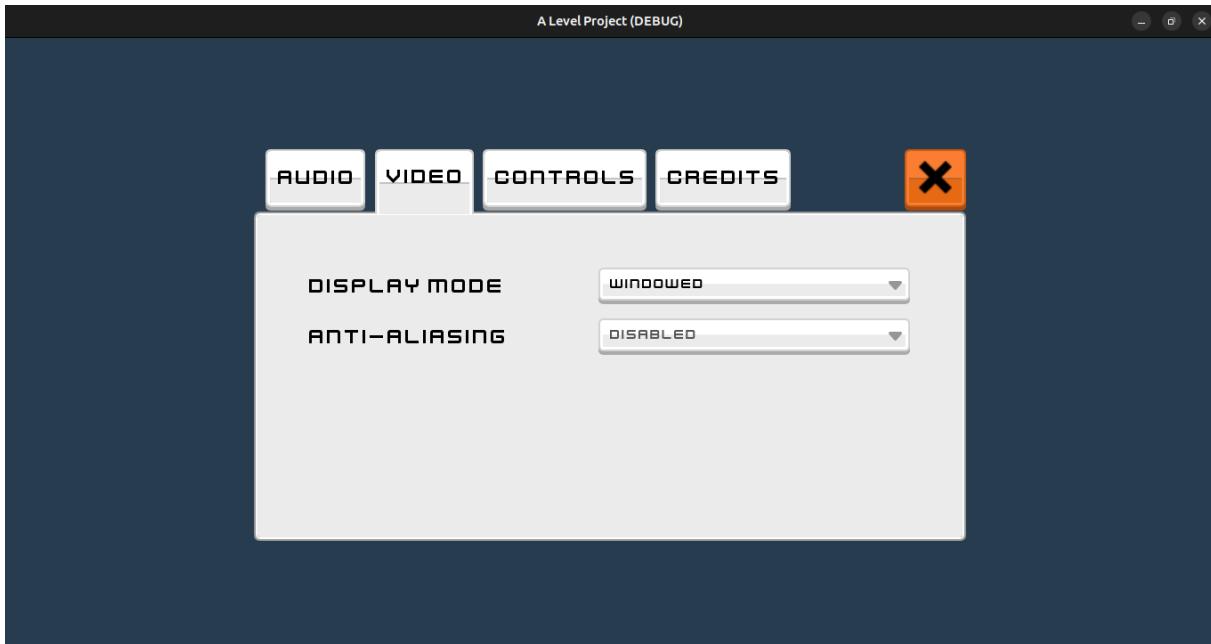
func _on_antialiasing_item_selected(index: int) -> void:
    # 0: Disabled
    # 1: MSAA 2x
    # 2: MSAA 4x
    # 3: MSAA 8x
    if index == 0:
        ProjectSettings.set_setting(ANTIALIASING_2D, Viewport.MSAA_DISABLED)
        ProjectSettings.set_setting(ANTIALIASING_3D, Viewport.MSAA_DISABLED)
    elif index == 1:
        ProjectSettings.set_setting(ANTIALIASING_2D, Viewport.MSAA_2X)
        ProjectSettings.set_setting(ANTIALIASING_3D, Viewport.MSAA_2X)
    elif index == 2:
        ProjectSettings.set_setting(ANTIALIASING_2D, Viewport.MSAA_4X)
        ProjectSettings.set_setting(ANTIALIASING_3D, Viewport.MSAA_4X)
    elif index == 3:
        ProjectSettings.set_setting(ANTIALIASING_2D, Viewport.MSAA_8X)
        ProjectSettings.set_setting(ANTIALIASING_3D, Viewport.MSAA_8X)
```



The dropdown also needs to set the initial value of the dropdown to the currently selected anti-aliasing setting. This is done by checking the project settings and setting the dropdown to the correct value.

```
# video.gd
func _ready() -> void:
    # ...

    # msaa_2d and msaa_3d are the same here
    var antialiasing := get_viewport().msaa_2d
    id = 0
    if antialiasing == Viewport.MSAA_2X:
        id = 1
    elif antialiasing == Viewport.MSAA_4X:
        id = 2
    elif antialiasing == Viewport.MSAA_8X:
        id = 3
    anti_aliasing.select(id)
```



Vertical Sync

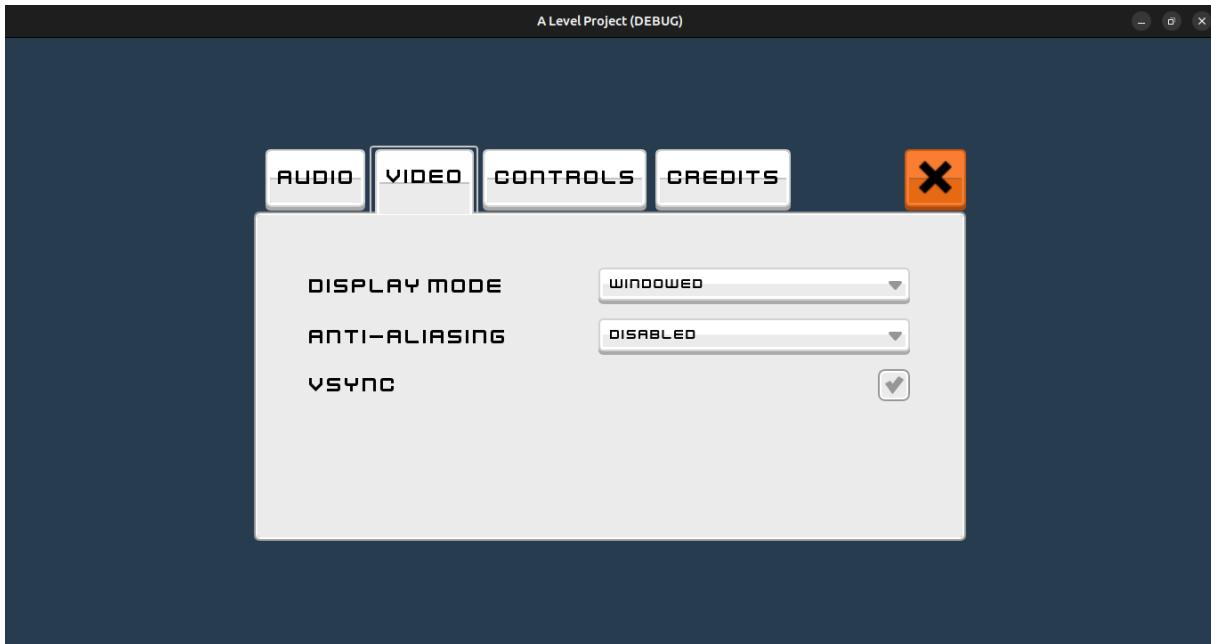
The final setting to implement is vertical sync. This is a simple setting, as it only has two options - on or off. This is set in the project settings, and the dropdown sets the initial value to the currently selected setting.

```
# video.gd
@onready var v_sync: CheckBox = %VSync

func _ready() -> void:
    #

    var vsync := DisplayServer.window_get_vsync_mode()
    if vsync == DisplayServer.VSYNC_ENABLED:
        v_sync.set_pressed_no_signal(true)
    else:
        v_sync.set_pressed_no_signal(false)

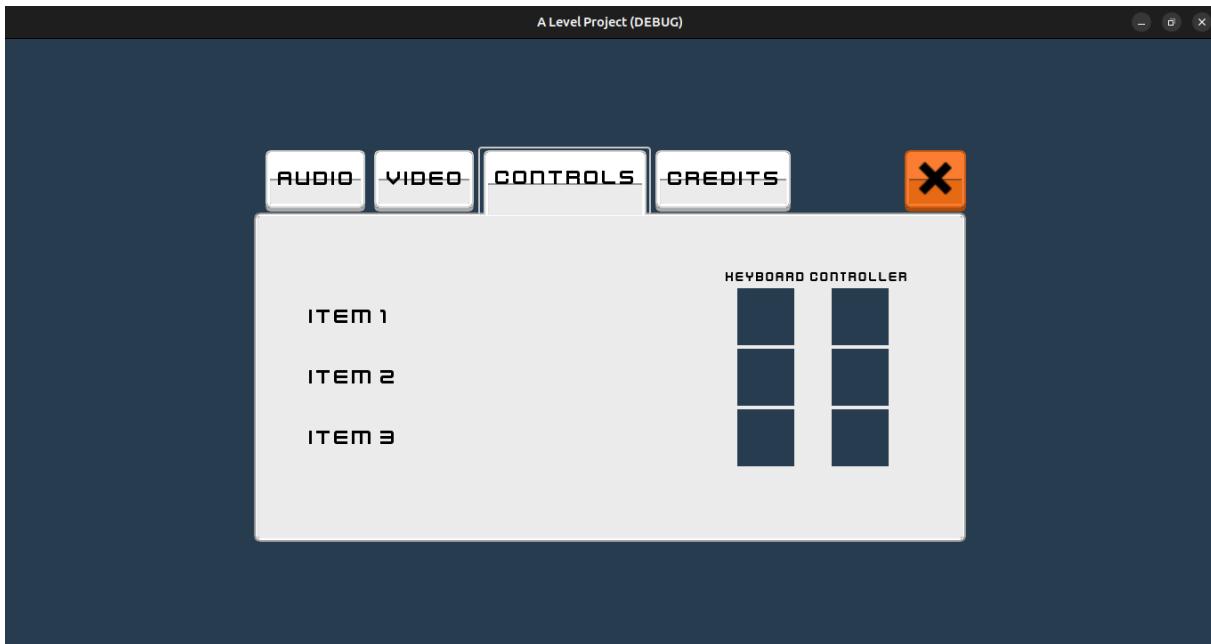
func _on_vsync_toggled(toggled_on: bool) -> void:
    if toggled_on:
        DisplayServer.window_set_vsync_mode(DisplayServer.VSYNC_ENABLED)
    else:
        DisplayServer.window_set_vsync_mode(DisplayServer.VSYNC_DISABLED)
```



Control Settings

Control Inputs

Textures



The controls menu contains separate controls for both keyboard and controller. This is so the user can set custom settings for both so they can play how they want to.

The textures for these buttons need to be stored in a resource to be used later. The following script provides properties that can be set in the editor with each

texture for all the supported keys. It also implements a `get_texture` method to retrieve the corresponding texture for an `InputEvent`.

```
# keys.gd
@tool
class_name KeyboardTextures
extends Resource

# All keys that can reasonably be used as custom controls.
# This means no modifier keys, no mouse buttons, no keys which do not usually
# pass through to programs such as numlock,
const _KEYS: Array[Key] = [
    KEY_0,
    KEY_1,
    KEY_2,
    KEY_3,
    KEY_4,
    KEY_5,
    KEY_6,
    KEY_7,
    KEY_8,
    KEY_9,
    KEY_A,
    KEY_B,
    KEY_C,
    KEY_D,
    KEY_E,
    KEY_F,
    KEY_G,
    KEY_H,
    KEY_I,
    KEY_J,
    KEY_K,
    KEY_L,
    KEY_M,
    KEY_N,
    KEY_O,
    KEY_P,
    KEY_Q,
    KEY_R,
    KEY_S,
    KEY_T,
    KEY_U,
    KEY_V,
    KEY_W,
    KEY_X,
    KEY_Y,
    KEY_Z,
    KEY_APOSTROPHE,
    KEY_LEFT,
    KEY_RIGHT,
    KEY_UP,
    KEY_DOWN,
    KEY_ASTERISK,
    KEY_BACKSPACE,
    KEY_BRACKETLEFT,
    KEY_BRACKETRIGHT,
    KEY_GREATER,
    KEY_LESS,
    KEY_CAPSLOCK,
    KEY_ASCIICIRCUM,
```

```

KEY_COLON,
KEY_COMMA,
KEY_DELETE,
KEY_END,
KEY_ENTER,
KEY_ESCAPE,
KEY_EXCLAM,
KEY_F1,
KEY_F2,
KEY_F3,
KEY_F4,
KEY_F5,
KEY_F6,
KEY_F7,
KEY_F8,
KEY_F9,
KEY_F10,
KEY_F11,
KEY_F12,
KEY_HOME,
KEY_INSERT,
KEY_MINUS,
KEY_PAGEDOWN,
KEY_PAGEUP,
KEY_PERIOD,
KEY_PLUS,
KEY_PRINT,
KEY_QUESTION,
KEY_QUOTEDBL,
KEY_SEMICOLON,
KEY_SLASH,
KEY_BACKSLASH,
KEY_SPACE,
KEY_ASCIITILDE,
KEY_TAB,
]

var textures: Dictionary = {}

func _init() -> void:
    for k in _KEYS:
        textures[OS.get_keycode_string(k)] = null

# Make this Resource act like all of its properties are contained in `textures`.
func _get(property: StringName) -> Variant:
    if property in textures.keys():
        return textures[property]
    return null

func _set(property: StringName, value: Variant) -> bool:
    if property in textures.keys():
        textures[property] = value
        return true
    return false

# Fake the properties we have as all the supported keys.
func _get_property_list() -> Array[Dictionary]:
    var properties: Array[Dictionary] = []

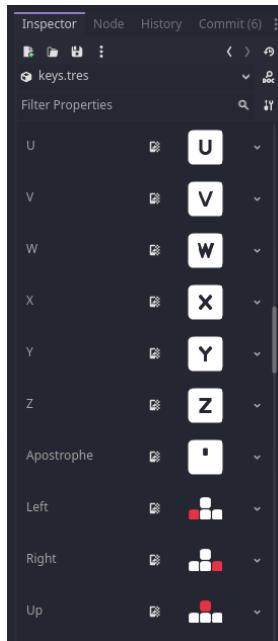
```

```

for k in _KEYS:
    properties.append(
        {
            name = OS.get_keycode_string(k),
            type = TYPE_OBJECT,
            hint = PROPERTY_HINT_RESOURCE_TYPE,
            hint_string = "Texture2D"
        }
    )
return properties

func get_texture(event: InputEvent) -> Texture2D:
    if not event is InputEventKey:
        return
    var key_event := event as InputEventKey
    var scancode := key_event.keycode
    return textures.get(OS.get_keycode_string(scancode), null)

```



This allows me to insert the textures into a new resource file and use them in the controls menu.

And similar with the controller textures, but with the controller buttons instead of keyboard keys. There are much less controller buttons so the properties do not need to be created dynamically, and allows me to add docstrings for what each button corresponds to.

```

# controller.gd
@tool
class_name ControllerTextures
extends Resource

## Bottom action (PS X, Xbox/Steam A, Nintendo B)
@export var button_0: Texture2D = null
## Right action (PS 0, Xbox/Steam B, Nintendo A)
@export var button_1: Texture2D = null

```

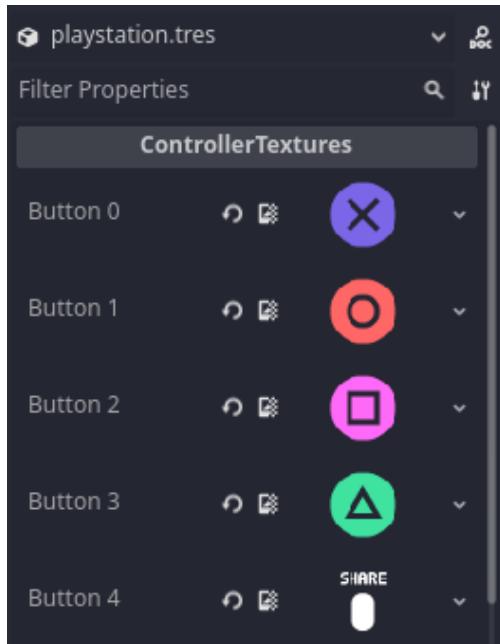
```

## Left action (PS □, Xbox/Steam X, Nintendo Y)
@export var button_2: Texture2D = null
## Top action (PS △, Xbox/Steam Y, Nintendo X)
@export var button_3: Texture2D = null
## Back (PS 1/2/3 Select, PS 4/5 Share, Xbox Back, Nintendo - )
@export var button_4: Texture2D = null
## Guide (PS PS button, Xbox home, Nintendo home)
@export var button_5: Texture2D = null
## Start (PS 1/2/3 Start, PS4/5 Options, Xbox menu, Nintendo +)
@export var button_6: Texture2D = null
## Left stick (PS L3, Xbox L/LS, Nintendo left stick)
@export var button_7: Texture2D = null
## Right stick (PS R3, Xbox R/RS, Nintendo right stick)
@export var button_8: Texture2D = null
## Left shoulder (PS L1, Xbox LB, Nintendo L)
@export var button_9: Texture2D = null
## Right shoulder (PS R1, Xbox RB, Nintendo R)
@export var button_10: Texture2D = null
## D-pad up
@export var button_11: Texture2D = null
## D-pad down
@export var button_12: Texture2D = null
## D-pad left
@export var button_13: Texture2D = null
## D-pad right
@export var button_14: Texture2D = null
## PS5 Microphone, Xbox Share, Nintendo capture
@export var button_15: Texture2D = null
## Paddle 1
@export var button_16: Texture2D = null
## Paddle 2
@export var button_17: Texture2D = null
## Paddle 3
@export var button_18: Texture2D = null
## Paddle 4
@export var button_19: Texture2D = null
## PS4/5 touchpad
@export var button_20: Texture2D = null

func get_texture(event: InputEvent) -> Texture2D:
    if not event is InputEventJoypadButton:
        return null

    var joypad_event := event as InputEventJoypadButton
    var button := joypad_event.button_index
    return get("button_" + str(button))

```



And this is similar with controllers, where I have created multiple resource files for different controller platforms.

To ensure only one button can be waiting for input, I have added a property to `global.gd` to share the currently listening prompt.

Buttons

```
# global.gd
## The control input that is currently listening
var listening_control: ControlInput
```

I then created a generic input prompt script which can be used for both keyboard and controller inputs. `get_texture` is left unimplemented as that is the part which differs depending on keyboard or controller inputs.

This uses the `global.gd` script to ensure only one input prompt can be waiting for input at a time. If a new input prompt is selected, the old one is unselected by calling `unlisten()` on it, and then setting the current prompt as the global property.

```
# control_input.gd
class_name ControlInput
extends Button

# Whether to act upon new `gui_input` events.
var listening: bool = false
# The texture to revert to if this input is unselected.
var previous_texture: Texture2D = null

@onready var texture_rect: TextureRect = %TextureRect
@onready var label: Label = %Label
@onready var global: Global = $"root/Global"
```

```

func _on_pressed() -> void:
    # Don't redo the same process if we are already listening, as that will
    # call `unlisten` below.
    if listening:
        return
    # Store the previous texture to revert to later if needed.
    previous_texture = texture_rect.texture
    texture_rect.texture = null
    label.text = "Waiting for input..."

    listening = true
    if global.listening_control != null:
        global.listening_control.unlisten()
    global.listening_control = self


func _on_gui_input(event: InputEvent) -> void:
    if not listening:
        return
    var texture := get_texture(event)
    if texture == null:
        return

    label.text = ""
    texture_rect.texture = texture

    listening = false
    global.listening_control = null


func _on_tree_exited() -> void:
    unlisten()

# Called either when this node is not visible (`tree_exited`), or a different
# control input is selected (see above).
func unlisten() -> void:
    listening = false
    label.text = ""
    texture_rect.texture = previous_texture


# keep `get_texture` abstract, for separate key inputs and controller inputs.
func get_texture(_event: InputEvent) -> Texture2D:
    return null

```

And then the individual implementations for keyboard and controller inputs.

```

# key_input.gd
class_name KeyInput
extends ControlInput

const textures: KeyboardTextures = preload("./resources/keys.tres")

func get_texture(event: InputEvent) -> Texture2D:
    var texture := textures.get_texture(event)
    # If it is not null, this is a valid texture, store this event as action`.
    if texture != null:
        store_action(event)
    return texture

```

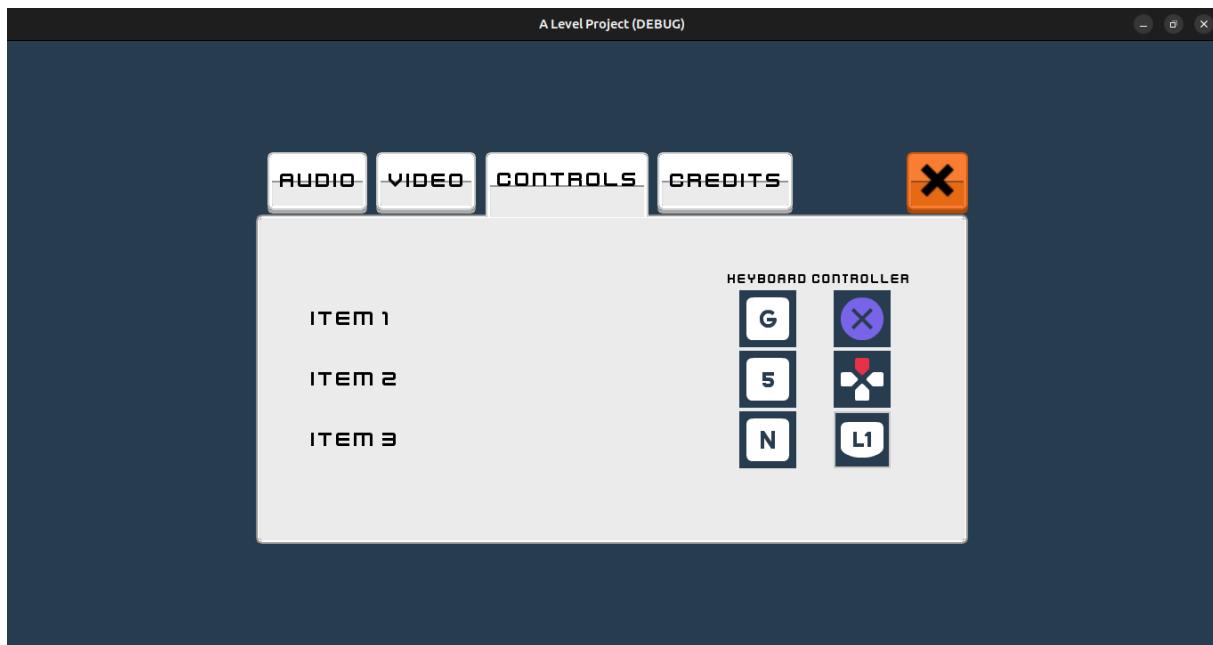
```

# controller_input.gd
class_name ControllerInput
extends ControlInput

const xb_textures: ControllerTextures = preload("./resources/xbox.tres")
const ps_textures: ControllerTextures = preload("./resources/playstation.tres")
const ni_textures: ControllerTextures = preload("./resources/nintendo.tres")
const st_textures: ControllerTextures = preload("./resources/steam.tres")

func get_texture(event: InputEvent) -> Texture2D:
    var device := event.device
    var joy_name := Input.get_joy_name(device)
    # The following conditions come from the public SDL controller database
    # https://github.com/mdqinc/SDL_GameControllerDB/
    if joy_name.contains("Xbox"):
        return xb_textures.get_texture(event)
    elif (
        joy_name.contains("PlayStation")
        or joy_name.contains("PS")
        or joy_name.contains("DualShock")
    ):
        return ps_textures.get_texture(event)
    elif joy_name.contains("Nintendo") or joy_name.contains("Switch"):
        return ni_textures.get_texture(event)
    elif joy_name.contains("Steam"):
        return st_textures.get_texture(event)
    else:
        return xb_textures.get_texture(event)

```



Only item 1, 2 and 3 are implemented as controls right now, as I am not sure what the other controls will be yet. This will be implemented later when I have a better idea of what the controls will be.

These settings do not apply to the game yet. The following code sets them in the global InputMap so they can be used in the game. Each control input takes in the action which is the name of the action in the InputMap.

```
# key_input.gd
@export var action: StringName

func _ready() -> void:
    var current_events := InputMap.action_get_events(action)
    for existing_event in current_events:
        if existing_event is InputEventKey:
            texture_rect.texture = get_texture(existing_event)
            break

func store_action(event: InputEvent) -> void:
    var current_events := InputMap.action_get_events(action)
    # Clear any existing key events for this current action and replace it with
    # the new one.

    for existing_event in current_events:
        if existing_event is InputEventKey:
            InputMap.action_erase_event(action, existing_event)

    InputMap.action_add_event(action, event)

# controller_input.gd
@export var action: StringName

func _ready() -> void:
    var current_events := InputMap.action_get_events(action)
    for existing_event in current_events:
        if existing_event is InputEventJoypadButton:
            texture_rect.texture = get_texture(existing_event)
            break

func store_action(event: InputEvent) -> void:
    var current_events := InputMap.action_get_events(action)
    # Clear any existing key events for this current action and replace it with
    # the new one.
    for existing_event in current_events:
        if existing_event is InputEventJoypadButton:
            InputMap.action_erase_event(action, existing_event)

    InputMap.action_add_event(action, event)
```

Ongoing Testing

Controller Input Passthrough

During testing of the controller inputs, I found that when assigning a D-pad button, the GUI would also accept that input and move the focus to the next button. This is not what I want, as I want the GUI to ignore the input and only

use it for the control input. I found that this can be prevented with a simple `Control.accept_event()` when handling an `InputEvent` in a specific `Control`.

```
# control_input.gd
func _on_gui_input(event: InputEvent) -> void:
    #
    if texture == null:
        return

    accept_event()

    label.text = ""
    texture_rect.texture = texture

    #
    # ...
```

Keyboard Input Loading

During testing of the keyboard inputs, I found that the existing keys from the `InputMap` are not properly loaded. There seems to be an inconsistency with `InputEvents` from real input, and those stored in `InputMap`, where `InputEventKey.keycode` is 0. Thankfully the Godot docs provide another property - `InputEventKey.physical_keycode` - which correlates to a US QWERTY keyboard layout. This can be converted back to a `Key` enum for the current keyboard layout.

```
func get_texture(event: InputEvent) -> Texture2D:
    if not event is InputEventKey:
        return
    var key_event := event as InputEventKey
    var scancode := key_event.keycode
    # When accessing InputMap, this is 0
    if scancode == 0:
        var physical := key_event.physical_keycode
        scancode = DisplayServer.keyboard_get_keycode_from_physical(physical)

    return textures.get(OS.get_keycode_string(scancode), null)
```

Controller Input Initial Focus

I also found that for controller input to work in menus, one `Control` should grab the focus initially so the controller focus knows where to start.

```
# main_menu.gd
func _ready() -> void:
    play_button.grab_focus()

# options.gd
func _ready() -> void:
    tab_container.get_tab_bar().grab_focus()
```

Saving Settings

The settings need to be saved between sessions, so the user does not have to set them every time they play the game. This is done by saving the settings to a

file in the user’s home directory. This is done in the `global.gd` script, which is autoloaded so `save_settings` can always be accessed.

Restructuring

Since the video mode will be retrieved here, I have moved that logic to `global.gd`, which is used in `video.gd` to set the initial value of the dropdown.

```
# global.gd
enum {WINDOW_WINDOWED, WINDOW_FULLSCREEN, WINDOW_BORDERLESS}
enum {ANTIALIASING_DISABLED, ANTIALIASING_2X, ANTIALIASING_4X, ANTIALIASING_8X}

func get_window_mode() -> int:
    var mode := DisplayServer.window_get_mode()
    var borderless := DisplayServer.window_get_flag(DisplayServer.WINDOW_FLAG_BORDERLESS)
    var id := WINDOW_WINDOWED
    if mode == DisplayServer.WINDOW_MODE_EXCLUSIVE_FULLSCREEN:
        id = WINDOW_FULLSCREEN
    elif mode == DisplayServer.WINDOW_MODE_WINDOWED and borderless:
        id = WINDOW_BORDERLESS

    return id

func get_antialiasing() -> int:
    # msaa_2d and msaa_3d are the same here
    var antialiasing := get_viewport().msaa_2d
    var id = ANTIALIASING_DISABLED
    if antialiasing == Viewport.MSAA_2X:
        id = ANTIALIASING_2X
    elif antialiasing == Viewport.MSAA_4X:
        id = ANTIALIASING_4X
    elif antialiasing == Viewport.MSAA_8X:
        id = ANTIALIASING_8X

    return id

# video.gd
func _ready() -> void:
    var id: int = global.get_window_mode()
    display_mode.select(id)

    var antialiasing: int = global.get_antialiasing()
    anti_aliasing.select(id)

    # ...
```

File Saving

Video Settings

The settings are saved using a `ConfigFile`. This is a key-value store which maps to an “ini” (non-standard) file format. This is saved in the user’s home directory.

```

# global.gd
## Config file for holding modified settings.
var settings := ConfigFile.new()

func save_settings() -> void:
    var vsync := DisplayServer.window_get_vsync_mode()
    var video_mode := get_window_mode()
    var video_antialiasing := get_antialiasing()
    var video_vsync_enabled := vsync == DisplayServer.VSYNC_ENABLED

    settings.set_value("video", "mode", video_mode)
    settings.set_value("video", "antialiasing", video_antialiasing)
    settings.set_value("video", "vsync", video_vsync_enabled)

    settings.save("user://settings.ini")

```

This is then called in `options.gd` when the options are being exited.

```

# options.gd
func _on_tree_exiting() -> void:
    global.save_settings()

```

oliver in ~/.local/share/godot/app_userdata/A Level Project λ cat settings.ini	
	File: <code>settings.ini</code>
1	[video]
2	
3	mode=0
4	antialiasing=1
5	vsync=true

Audio Settings

The audio settings saving is similar to the video settings. The settings are saved as a mapping of bus index to volume in decibels.

```

# global.gd
func save_settings() -> void:
    var bus_count := AudioServer.bus_count
    for bus in range(bus_count):
        var db := AudioServer.get_bus_volume_db(bus)
        settings.set_value("audio", str(bus), db)

    #

```

File: /home/oliver/.local/share/godot/app_userdata/A Level Project/settings.ini	
	File: /home/oliver/.local/share/godot/app_userdata/A Level Project/settings.ini
1	[audio]
2	
3	0=0.0
4	1=0.0
5	2=0.0
6	
7	[video]
8	
9	mode=0
10	antialiasing=0
11	vsync=true

Control Settings

The control settings need to store a key event and a joypad button event for each action. `store_event` is a separate function to reduce the size of `save_settings` as there were too many indents for it to be easily readable.

```
# global.gd
func save_settings() -> void:
    # ...

    var actions := InputMap.get_actions()
    for action in actions:
        # ui_* are default UI traversal controls.
        if not action.begins_with("ui_"):
            var events := InputMap.action_get_events(action)
            for event in events:
                store_event(action, event)

    # ...

func store_event(action: StringName, event: InputEvent) -> void:
    if event is InputEventKey:
        # Similar to prompts/resources/keys.gd#get_texture
        var key_event := event as InputEventKey
        var keycode := key_event.keycode
        if keycode == 0:
            var physical := key_event.physical_keycode
            keycode = DisplayServer.keyboard_get_keycode_from_physical(physical)

        settings.set_value("controls", action + "_key", keycode)
    elif event is InputEventJoypadButton:
        var joypad_event := event as InputEventJoypadButton
        var button := joypad_event.button_index

    settings.set_value("controls", action + "_control", button)
```

File: /home/oliver/.local/share/godot/app_userdata/A Level Project/settings.ini	
1	[audio]
2	0=0.0
3	1=0.0
4	2=0.0
5	[video]
6	mode=0
7	antialiasing=0
8	vsync=true
9	[controls]
10	use_item_1_key=54
11	use_item_1_control=13
12	use_item_2_key=50
13	use_item_2_control=11
14	use_item_3_key=51
15	use_item_3_control=14

Ongoing Testing

Anti-Aliasing Loading

During testing of saving video settings, I found that antialiasing was always disabled. I found out that `ProjectSettings` is not needed for this, and it can be set directly as `Viewport.msaa_2d` and `Viewport.msaa_3d` in `video.gd`.

```
# video.gd
func _on_anti_aliasing_item_selected(index: int) -> void:
    var viewport := get_viewport()
    if index == 0:
        viewport.msaa_2d = Viewport.MSAA_DISABLED
        viewport.msaa_3d = Viewport.MSAA_DISABLED
    elif index == 1:
        viewport.msaa_2d = Viewport.MSAA_2X
        viewport.msaa_3d = Viewport.MSAA_2X
    elif index == 2:
        viewport.msaa_2d = Viewport.MSAA_4X
        viewport.msaa_3d = Viewport.MSAA_4X
    elif index == 3:
        viewport.msaa_2d = Viewport.MSAA_8X
        viewport.msaa_3d = Viewport.MSAA_8X
```

File Loading

Audio Settings

The settings are now saved to `settings.ini`. They need to be loaded when the game is ran. This can be done in `Global._ready` by loading the settings from the file and setting the values in the game in the global singletons `DisplayServer`, `AudioServer` and `InputMap`.

`SETTINGS_PATH` is moved to a constant as it is used in both `save_settings` and `load_settings`.

```
# global.gd
const SETTINGS_PATH = "user://settings.ini"

# ...

func _ready() -> void:
    load_settings()

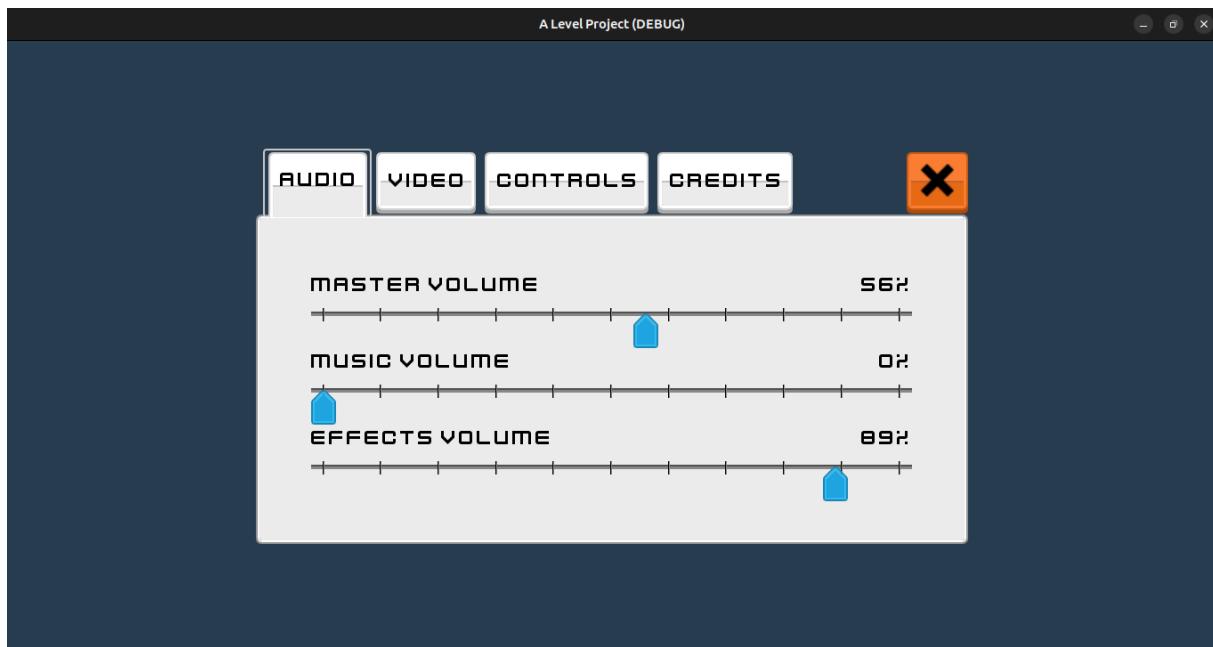
# ...

func load_settings() -> void:
    settings.load(SETTINGS_PATH)

    var sections := settings.get_sections()

    if sections.has("audio"):
        var buses := settings.get_section_keys("audio")
```

```
for bus in buses:  
    var db: int = settings.get_value("audio", bus)  
    var index := int(bus)  
    AudioServer.set_bus_volume_db(index, db)
```



The options shown above in the screenshots are being loaded as decibels and stored in `AudioServer`.