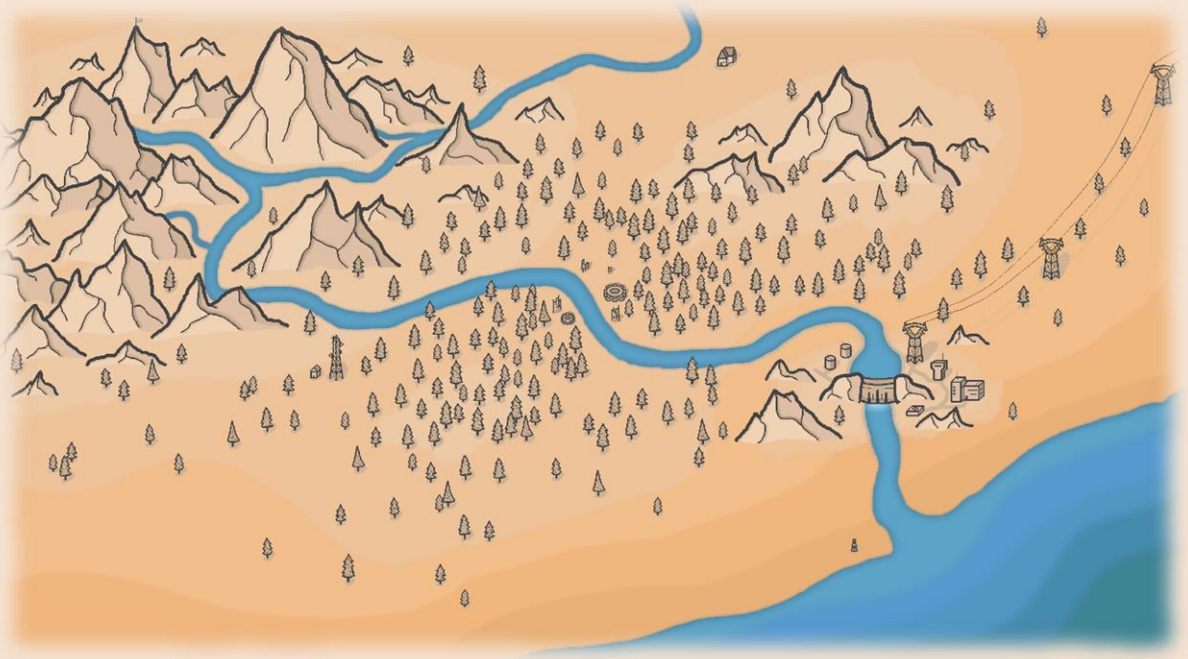
Australian STEM Video Games Challenge 2025

BUOYANT VOYAGE

Game Design Document



Team Jaraph

Daniel Smith | Patrick Henschke

Contents

Important Notes 2
A Brief Introduction 3
Inspiration & Organisation 4
The Team 4
Meeting Submission Guidelines 5
Inspiration & Originality 6
Development Environment: Love2D / Lua 11
Resourcing 14
Timeline and Workflow 18
Game Design: Overview 21
Theme 21
Game overview 22
Perspective and User Interface 24
Movement Controls 26
World & Levels 28
Game Design: Visual & Audio 32
Visual Design 32
Audio Design 36
Music Design 39
Game Design: River Generation 41
The Procedural Generation Thread 41
Obstacles 47
Other Technical Aspects 50
Player 50
Camera 51
Dynamic Loading 52
Other Tweaks & Quality of Life 53
Testing & Reflecting 55
Testing 55
Fixing 56
Project Execution & Reflection 62
Reference list 64
Appendices 65
Appendix A: Obstacles 65
Appendix B: Game Systems Tree 68
Appendix C: Some concept art and design sketches 70
Appendix D: Sound Attributions 73
A Closing Note 73

IMPORTANT NOTES

Cheat Code

To unlock all levels, hold:

[Left Shift] + [U]

...**for <u>one second</u>, <u>while in the level select</u>**. This is not intended for any version apart from the STEM VGC release. Using the cheat code may cause certain dialogue/info-boxes not to appear, so we recommend you give the game a shot first.

Download Notes

To play *Buoyant Voyage,* you have to unzip the downloaded zipped folder. The executable file (BuoyantVoyage.exe) **must** remain in the folder it is downloaded in, as this folder also contains Love2D's required binaries (.dll files).

GitHub Repository

Accessing the source code can be done though the GitHub page, linked below:
https://github.com/DanteGraet/ASVGC-25

Itch.io

A compiled version of *Buoyant Voyage* is available through the itch.io page:
https://dantegraet.itch.io/bouyant-voyage

Gameplay Footage

Gameplay footage containing extracts from all 3 stages:
https://youtu.be/rshq8sYVIis

Our Usernames

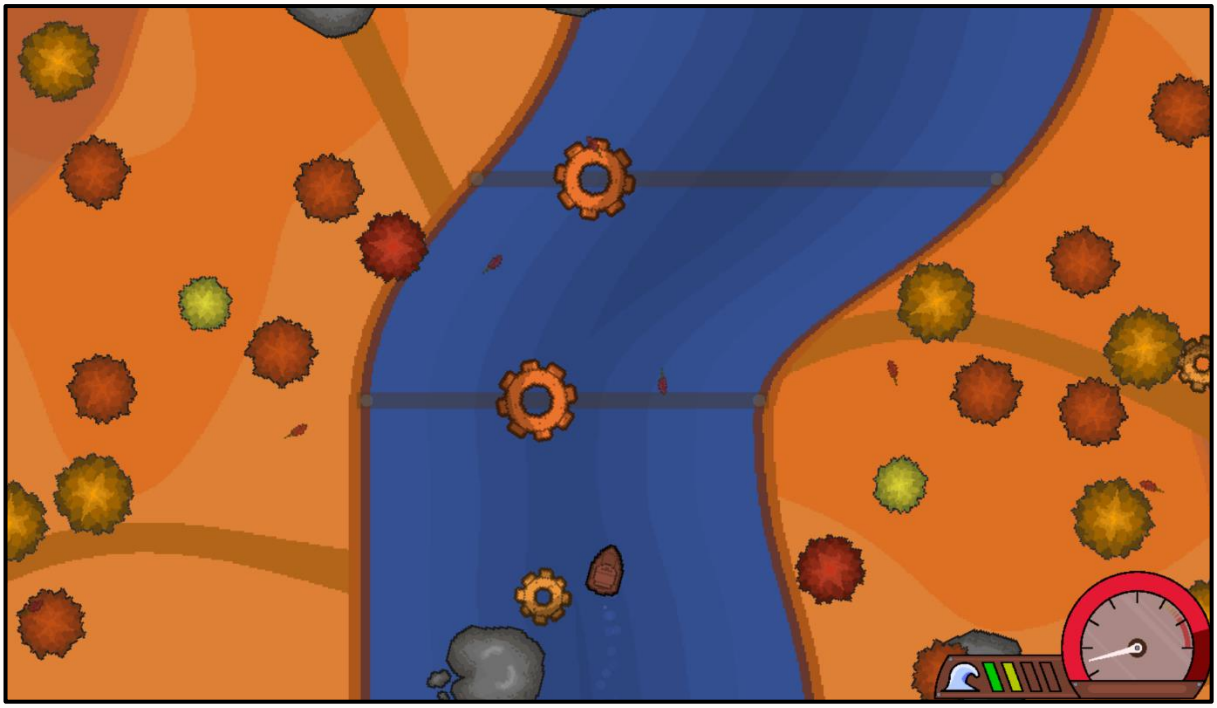
Throughout this document, on the GitHub repository, or the Itch.io page, you may come across our online usernames, so we've put them here to minimise any confusion:

Quindocah - Patrick

DanteGraet - Daniel

A Brief Introduction

Buoyant Voyage is a 2D arcade-style game in which you journey down a procedurally generated river in a wooden boat. Watch out – rocks, storms, high current, turbine blades, and ancient clockwork constructions are just a few of the obstacles between you and the ocean.



The "Clockwork Ruins" sub-biome in Level 2

Buoyant Voyage is the successor to the first project we created 4 years ago, *Boat Game With a Name*. *Buoyant Voyage* is written in Love2D, a lightweight Lua framework which gave us a lot of freedom, but also not much to work with. If we had to pick *Buoyant Voyage's* most impressive feature, it would be the procedurally generated river.

This was our biggest project ever, and we had a lot of fun making it whilst juggling all the other year 12 work we unfortunately also had to do. Hopefully, you'll also have some fun playing it and following its development in this Game Design Document. We've added some humour here and there in the hopes of keeping the document fun.

"Daniel, we probably want a quote for the Introduction"
"Ah jeez. How about, 'nothing is as permanent as a temporary solution?'"
"Um, okay... I'll put that as the placeholder until we find a better quote"

Inspiration & Organisation

The Team

Jaraph is a team of two – Daniel and Patrick. We're two year 12s from different schools in Adelaide, SA. We're both somewhat competent in programming, art, and design, but for a project of this size it was essential that we divide work and focus our strengths. Daniel had much more technical experience, and Patrick was better at detailed pixel art, so we split tasks as follows:

Daniel is our primary programmer, and given that Love2D is a very code-heavy environment, he has a lot to do:

- Building the many systems the game relies on. Daniel made the river generator, the player, a system for creating obstacles, key binds and settings, and a lot more. He handles everything which works behind the scenes to allow Patrick to layer gameplay on top.
- Putting together *Buoyant Voyage's* numerous UI systems, for navigating between levels or changing settings. We designed their layout together, but Daniel made them functional.
- Daniel organises the directory and code files, which is important as Love2D has no interface like other engines. He also made the GitHub repository.
- No game is complete without SFX, so Daniel does a lot of the sound design and recording. He also does the code to make those sounds play in-game.

Patrick does most other things- art, game design, music and some programming:

- Level design. The river's curves and bends are generated procedurally, but the obstacles that fill it are designed and programmed (often working with Daniel) by him. He also conceived level's themes and the game's progression.
- Art; both the pixel art (during gameplay) and high-resolution art (for level select and UI) is done mostly by Patrick, in pixel art program Aseprite. We have quite distinct art styles, so Patrick handled most of it for consistency.
- Patrick composed all of *Buoyant Voyage's* music using a midi keyboard and composition software *Logic Pro*. To him, music is a very important part of any game, so he spent time making it atmospheric and adaptive.
- We both worked on the Game Dev Doc, but Patrick was just in charge of editing and structuring the document.

Meeting Submission Guidelines

Keeping *Buoyant Voyage* aligned with the STEM VGC submission guidelines was mostly straightforward. Our game would most likely be G-rated; it has very mild violence and no mature themes or profanity.

STEM VGC games should run primarily on Windows, and this was no issue. Both of us use Windows computers, and Love2D has always worked smoothly with Windows; we can compile to a .exe executable with just a few lines in command prompt.

As our game is a purely solo experience, it fulfills the 'primarily playable in single-player' criteria. The control scheme is keyboard and mouse. Once compiled to executable, no extra software is required to run the game (aside from Love2D's .dll files, which come included with the game in the zip folder)

Code visibility - while there's no interface for Love2D, we can directly upload the game's uncompiled directory as a .zip, so that judges have access to all the raw code files and assets. There's also the GitHub repository which is publicly visible.

"Wait, doesn't that mean the judges can see all the typos?"
-Patrick

Online Assets

Both of us place value on creating original work (which is part of why we chose Love2D, as it meant more originality in our programming), and so we never considered using any store-bought assets. All the art and music we needed Patrick made. We **did** source some free sounds with Creative Commons or Public Domain licences to fill the gaps in sound design left by what we couldn't synthesise ourselves – for instance, a diesel motor sound which formed a component of our final boat engine noise. When we used any CC sounds we made sure to give attribution in the in-game credits menu. You can also find a list of attributions in ***Appendix D***.

Use of AI

Our game (and GDD) features minimal use of AI. No art assets, music, or writing is AI-generated. There are a few small sections in the codebase which are AI-assisted. Occasionally, we would use ChatGPT to explain advanced concepts to us, e.g. "how to use 'channels' in multithreading". At most, we would ask for pseudocode and then translate that to Love2D, but we kept even that to an absolute minimum. Given how obscure Love2D is, large sections of AI code would probably fall apart anyway.

```python
import math

def hex_to_rgb(hex_code):
    hex_code = hex_code.lstrip('#')

    if len(hex_code) != 6 or not all(c in "0123456789ABCDEFabcdef" for c in hex_code):
        raise ValueError("Bad hex code.")

    r = round(int(hex_code[0:2], 16) / 255)
    g = round(int(hex_code[2:4], 16) / 255)
    b = round(int(hex_code[4:6], 16) / 255)

    return r, g, b

def round(num):
    return math.floor(num*100)/100

while True:

    user_input = input("\nEnter hex code: ").strip()
    try:
        rgb = hex_to_rgb(user_input)
        print("{"+str(rgb[0])+","+str(rgb[1])+","+str(rgb[2])+"}")
    except ValueError as e:
        print(e)
```

The 'hexcodinator', a development tool (written in Python)

This code snippet here is by far the most AI-rich section of our codebase, which we've included in the GDD for the sake of transparency. It's a Python script which turns Hex codes into a RGB format. **This code is not within the release of the game**, but a development tool just to save a few minutes of time. It's written by ChatGPT and slightly modified by Patrick to work with Love2D's specific RGB format.

As we're not associated with a school or club it was up to us to set our own AI 'ethics' guidelines, and we felt that this was an appropriate use. Either of us could have written this script, but we felt it 'relatively harmless' as it wasn't in the final game anyway. Regardless, this small tool probably would've taken more time to write than it would have saved, had one of us written it.

Inspiration & Originality

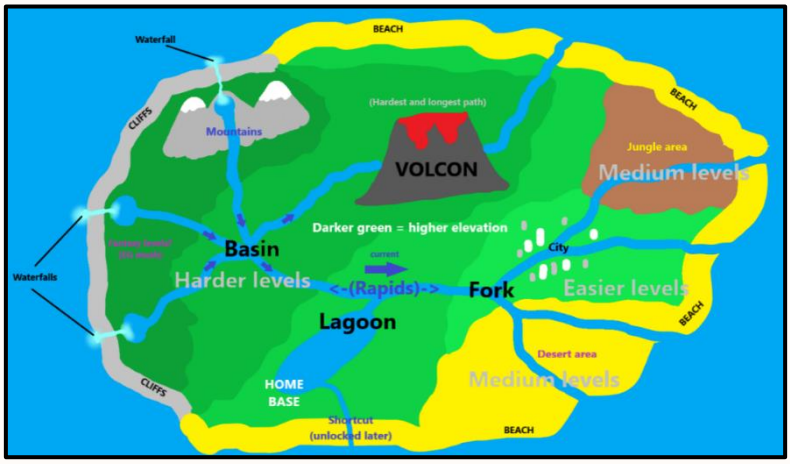
Buoyant Voyage draws inspiration from many other games, and also the real world. Our top three points of inspiration for *Buoyant Voyage* would be:

- ***Boat Game With a Name***, created by us all the way back in 2021. This was our first proper collaborative project. Being its successor, the themes and gameplay of *Buoyant Voyage* are similar, though there are key differences.
- ***Build a Boat for Treasure***, a Roblox boat-building adventure game which was a large inspiration point for *Boat Game With a Name.*
- And a real-life inspiration point: Real-world creeks and rivers, particularly our walks along the creeks in **Belair National Park**, South Australia.

Buoyant Voyage's predecessor: Boat Game With a Name

Gameplay from inspiration source #1: Boat Game With a Name

We created *Boat Game With a Name* (Abbreviated as "BGWaN") using scratch back in middle school. While we were happy with this project, we always wanted to create a much more professional sequel. Patrick was designing world ideas for *Buoyant Voyage* all the way back in 2022, and by 2023 Daniel had written a very bare-bones prototype in Pygame which was almost functional. But these ideas never got anywhere, and we ended up abandoning them.



An ancient (2022) concept for Buoyant Voyage's map. It never came into fruition.

Going into the 2024 STEM VGC, there was no way a boat game could fit the 'Stars' theme, so we pretty much forgot about the idea. But when we heard the theme for 2025 was 'Journey', we knew it was time for *BGWaN's* successor.

Buoyant Voyage is probably better described as a *BGWaN's* spiritual successor, rather than a direct sequel. The core gameplay is very similar to that of *BGWaN* – steering a boat down a river was a formula that we already knew was fun, and so we refined it. Other aspects of gameplay, such as the game's progression and world, are completely different.

BGWaN followed a rouge-lite progression format; players would earn gold based on how far they got through the river, and spend that gold in upgrading their boat in hopes to get farther. Patrick had always felt this progression loop rather unsatisfying for reasons he'll talk about later in the ***Game Design: Overview Section***. Long story short, we removed this aspect; in *Buoyant Voyage* the only progression is that of the player's skill improving. We completely reworked the world and biomes, too. While *BGWaN* saw players exploring a huge variety of biomes, *Buoyant Voyage* has just three levels each with multiple sub-biomes which share a consistent theme.

Compared to *BGWaN, Buoyant Voyage* is a more polished, 'professional'-feeling game. It retains the pixel-art style, albeit with a higher resolution. *Buoyant Voyage* also features high-resolution / 'smooth' art for UI and the world map, as we didn't like how sticking to the pixel art style made *BGWaN's* shop menu hard to read. Finally, *Buoyant Voyage's* river is procedurally generated as opposed to handcrafted. It also has bends and curves, which didn't work in Scratch's 4:3 aspect ratio.

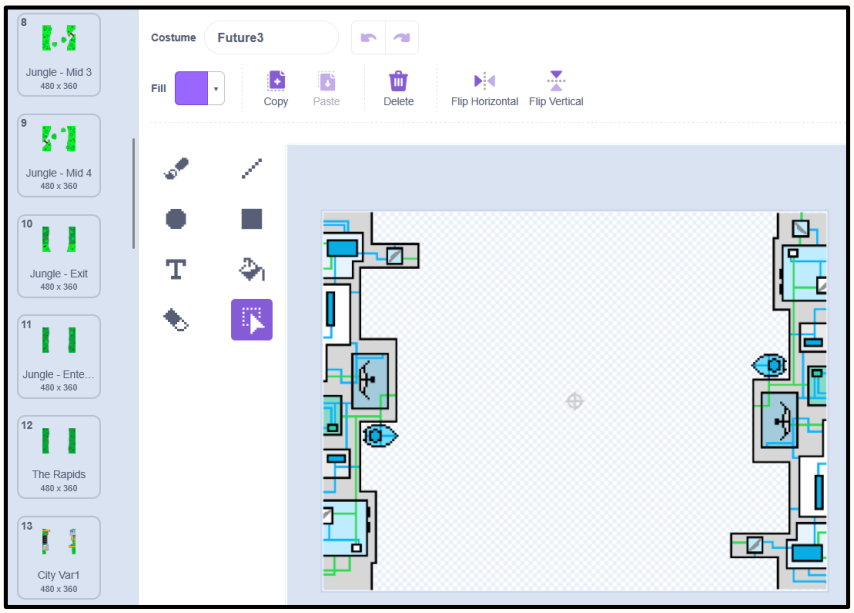


Image above: BGWaN constructed its river by shuffling ~40 handmade segments.

Contrasting BGWaN, Buoyant Voyage's river is fully procedurally generated. We never quite met that high-fidelity detail which handcrafting segments allowed for, but procedural generation brought many other advantages – more on that later.

Inspiration point 2: Build a Boat for Treasure

Buoyant Voyage and *BGWaN* were inspired by a Roblox game we used to play (and still do from time to time) – *Build a Boat for Treasure*.

Gameplay from inspiration source #2: Build a Boat for Treasure

"Patrick, you added this screenshot just to show off that boat you built, didn't you?"
-Daniel

In *Build a Boat for Treasure* (or just *Build a Boat* for short), players construct a boat out of blocks and other parts before sending it down a river in hopes of finding treasure. The idea of exploring a river with multiple biomes, each with unique obstacles, was largely inspired by *Build a Boat*. Although similar in theme, our game varies largely in gameplay – and not just because they have a third spatial dimension, and we don't.

Build a Boat's core gameplay is about building, and then watching as your creation falls apart block by block as it smashes into rocks, falls into giant washing machines, or whatever else. There are some mechanics in *Build a Boat* which give you limited control of the boat, but these are intentionally finnicky. This contrasts our game, which doesn't have any of the building mechanics, but instead a focus on steering your boat; skilfully dodging the obstacles which *Buoyant Voyage* throws at you.

Another point of difference is in the river's shape and generation-*Build a Boat's* river is created by chaining together handcrafted segments and randomly filled them with obstacles (which inspired the generation in *BGWaN*). *Buoyant Voyage's* curvy, procedurally generated rivers are a point of distinction from *Build a Boat.*

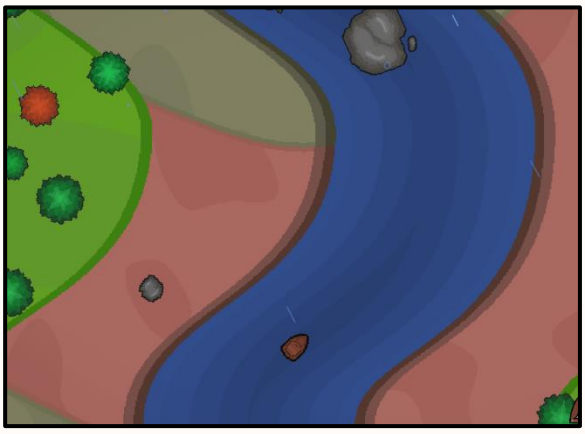
Inspiration point 3: Real-world creeks and rivers

On a camping trip at Belair National Park many years ago, the creek was dried up. We took a trek along the riverbed, following it all the way out of the park and partially into the town area (we probably weren't meant to be there). During this walk, we came up with the idea for *Boat Game With a Name*, a decision which eventually led to *Buoyant Voyage's* creation all these years later.

At one point, the creek stank overpoweringly of what we thought were mushrooms, at another, strange concrete constructions jutted out of the ground. We were young, so this felt wonderous – and we wanted to bring back that sense of adventure something as simple as walking along a dried creek gave us. This influenced our design choices in creating *Buoyant Voyage's* levels. We spent time making the levels feel immersive and atmospheric through particle effects, atmospheric music, and pacing the progression through each level, so that the journey felt immersive.

We could have made the player trek through all sorts of places – desert, jungle, cave, city; a huge variety of distinct but unconnected experiences to cram as many unique features in the game as possible. But, with our creek-adventuring experiences in mind, we felt a continuous experience, where each stage sticks with one specific theme, would better re-create that feel of adventure and journey.

In general, much of *Buoyant Voyage's* world takes inspiration from the beautiful real world around us. Of course, we tried to make the generation feel somewhat realistic, but some zones take more specific inspiration from real-world features.



Left: A glacial river; photo taken by Patrick on a family trip to New Zealand. Right: Buoyant Voyage's 'Gravelly Plains' sub biome, with the different 'gravel' colours inspired by the real-world environment.

'Micro-Inspiration' Points

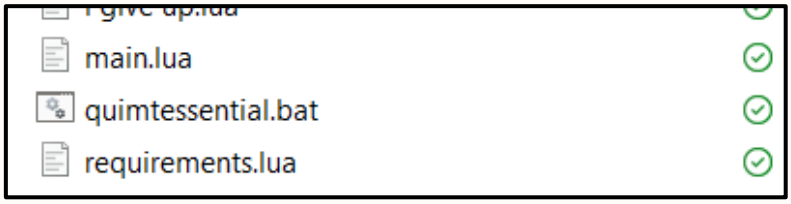
There are a few other inspiration points for *Buoyant Voyage* which didn't inspire the game's core gameplay or theme, but rather small gameplay features or individual artistic choices. These are some of our favourite games (and a book):

- *Celeste – inspired the remixed "storm" levels*
- *Stardew Valley – inspired some weather/particle effects*
- *Risk of Rain 2 – inspired some UI elements*
- *The Hobbit* (book) *– inspired the art style for the level select's map*

Further details on what inspiration we drew from these games will be discussed later in the ***Game Design*** section alongside the feature(s) they inspired.

Development Environment: Love2D / Lua

Buoyant Voyage is written in the Lua framework Love2D. It's called a 'framework' as opposed to a game engine, because there's no interface. Working on the game instead entails editing code files in a separate code editor (we used Microsoft's Visual Studio Code) and placing assets within the game's directory using file explorer.



To give you a sense of how little interface Love2D has, to test the game without compiling, we had to boot the game via a batch file – the 'quimtessential' batch file.

Love2D handles basic functionality – drawing graphics, an update loop, playing sounds, taking inputs, a watered-down physics engine, delta time, and so on. If we want anything more advanced, like clickable buttons, we had to code all that ourselves. It's a very code-heavy environment, and there's none of the scene editor or asset managers which other game engines like Godot or Unity would've given us. Love2D is comparable to something like Pygame (though, Love2D is more sophisticated)

You may ask, "why put yourselves through this?" Answer: because, for some reason, we really enjoy working with it. Both of us are familiar with Lua and Love2D, and we enjoy the huge amount of freedom (and added challenge!) that comes with having close to full control over how everything works. You can definitely achieve that level of control in an engine, but with much more additional work and skills we didn't have.

So, similar to how we retained the core gameplay from a previous project, we decided to stick with what we were familiar with and knew we loved.

Working in an environment like Love2D informed our game design decisions too, like the need for a procedural generation system to make up for the lack of a scene editor. If we wanted to handcraft *Buoyant Voyage's* river, much more manual labour (filling in hitboxes, typing spawn-points in via code, so on) would've been required.

Alternatively, we could've coded our own level editor – something we actually did for last year's STEM VGC project, but that would be a massive time investment. Limitations drive creativity; we were forced into procedural generation in the end, and it ended up being our game's strongest feature.

We did seriously consider switching to a more 'proper' engine, like Unity...

Engine	Unity	Love2D
Scene Editor	Yes	Unfortunately, not.
Documentation	Very extensive	Decent; the Love2D wiki has everything we need
Tutorials	Very extensive	Few and far between, and even less video tutorials.
Personal Experience	Daniel has experimented in Unity. Patrick downloaded it but never touched it.	Both of us are very familiar with Lua and Love2D; we enjoy working in it.
System requirements	8 GB RAM; 16GB Storage. Daniel's computer had occasional issues running it.	512 MB RAM, 10MB Storage. Could probably run on a potato.
Physics Engine(s)	Box2D (2D) PhysX (3D)	A watered-down version of Box2D.
Graphics	2D & 3D	2D only.
Language	C#	Lua
Cost	'Free' with some conditions	Free and open source

A somewhat informal comparison between Unity and Love2D

It may have had its advantages, but there would be a lot of learning to do had we switched engines, and eight months is already a short enough timeframe. We decided we enjoyed Love2D's full control despite its limitations, and with some outside-the-box thinking, it was still possible to make a fully functional project in the end. The main limitation is we didn't have access to flashy graphics and a lighting system, which decreased our game's aesthetic appeal.

System Requirements

As of writing this, *Buoyant Voyage* runs on Windows only. We've successfully exported past Love2D projects to Mac and Android before, but the process is a little more complicated. *Buoyant Voyage* is a 2D game, and whilst it's not super well optimised, players probably won't experience issues on a mid-range laptop or higher. We'd recommend at least the following specs as a minimum requirement:

- Intel core i5 10th gen or equivalent (64-Bit)
- 4GB RAM
- 100MB Storage
- Integrated Graphics

The most resource-intensive aspect of *Buoyant Voyage* is the river 'background' generation; filling in the many thousands of coloured pixels used to draw the river is processing-intensive. If the game cannot keep up with background generation, it will stay at 60FPS at the cost of parts of the river not rendering correctly. As wider aspect ratios need to generate more pixels per unit of river, they are more prone to 'lagging'.

Demonstration of the river generation not being able to keep up. Cheats were used to speed the player's boat up to extreme speeds, thereby inducing the issue.

This is not a bug; it is an intentional behaviour (thanks to multi-threading) to keep the game running smoothly even when the computer can't keep up. Our laptops rarely experienced such issues, though those with an older processing unit might.

"Um, my fan is going like crazy, and the river is lagging again."

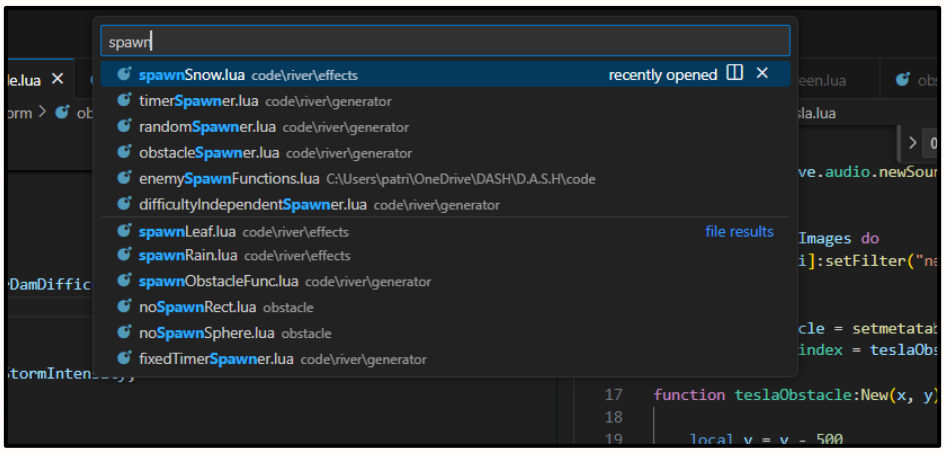
"Daniel, you are screen-sharing AND screen recording. What did you expect?"

Resourcing

Aside from **Love2D**, we used many applications in the creation of *Buoyant Voyage:*

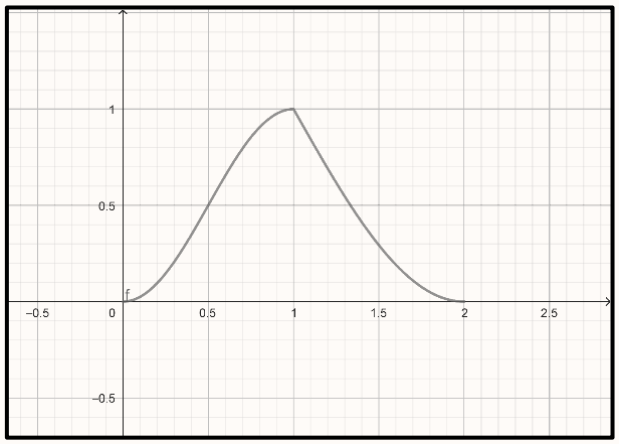
Programming: Visual Studio Code, GeoGebra
Assets, Sound & Music: Aseprite, SFXR, Audacity, Logic Pro
Project Management: GitHub, Trello

Our code editor of choice is Microsoft's **Visual Studio Code** (VSC); a free and versatile code editor that has all the features we need. VSC's 'code workspaces' were especially helpful for working across our project's many files. VSC also has extensions for Lua and Love2D, which can slightly speed up the process of writing and debugging code.



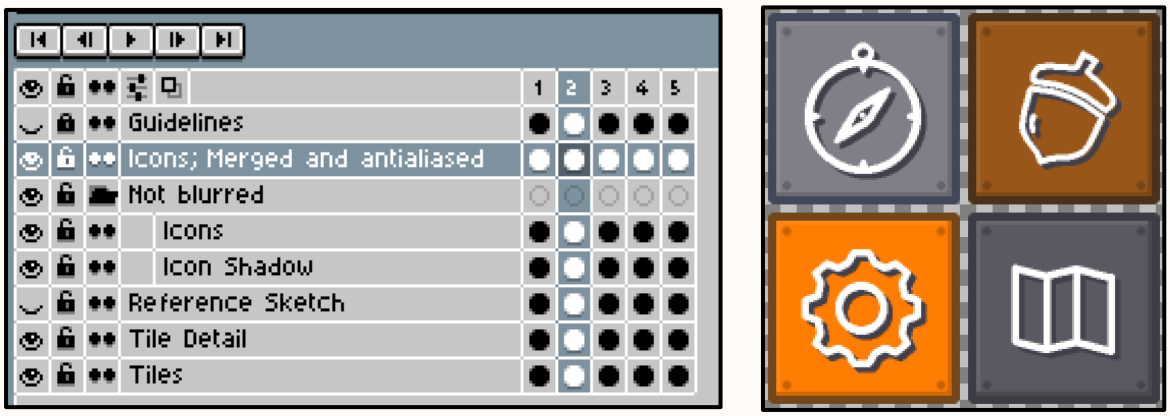
Being able to navigate between our files via a search bar was a huge time save.

Although not created specifically for game development, **GeoGebra** held a part in the development of *Buoyant Voyage*. GeoGebra is a useful tool for graphing the functions that defined gameplay elements, like the leaf particle particle's movement.



A function created in GeoGebra to model leaves blowing in the wind, used to add immersive particle effects to Stage 2, Autumn Grove.

All the assets in *Buoyant Voyage*, both high resolution and pixel art have been created in **Aseprite**, a sprite editor designed for creating and animating pixel art. Patrick is familiar with Aseprite, and its layering system (comparable to adobe Photoshop's layering) was useful for creation of more detailed assets. As much of *Buoyant Voyage* is in a pixel art style, Aseprite was a good choice of image editor.



Use of Aseprite's layers in creating the loading screen's 'tiles'. As we have different tiles for each stage, we stored each set of tiles in its own animation frame. Even though this sprite isn't animated, the animation system still came in handy.

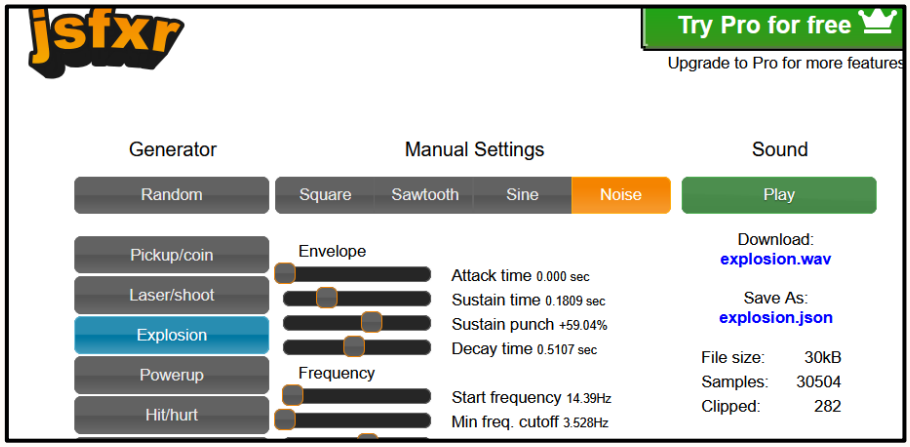
Aseprite is a pixel-art editor but was still useful for creation of more high-resolution assets like the tiles shown above, though the program occasionally slows down when editing higher-resolution images.

To compose *Buoyant Voyage's* music, we used **Logic Pro**, a professional DAW (Digital Audio Workstation) which Patrick was lucky enough to have on his family's MacBook. He used a small midi keyboard to play and input the melodies, chords and drumbeats into Logic, before selecting an instrumental soundfont from Logic's built in instrument library.

Logic lets you layer multiple sequences on-top of each other (much like how Aseprite let us layer sprites), which was crucial for composing *Buoyant Voyage's* adaptive music. Logic also offers more advanced capabilities like 'automation' and EQ, both of which Patrick used to make the music more dynamic (more on that later).

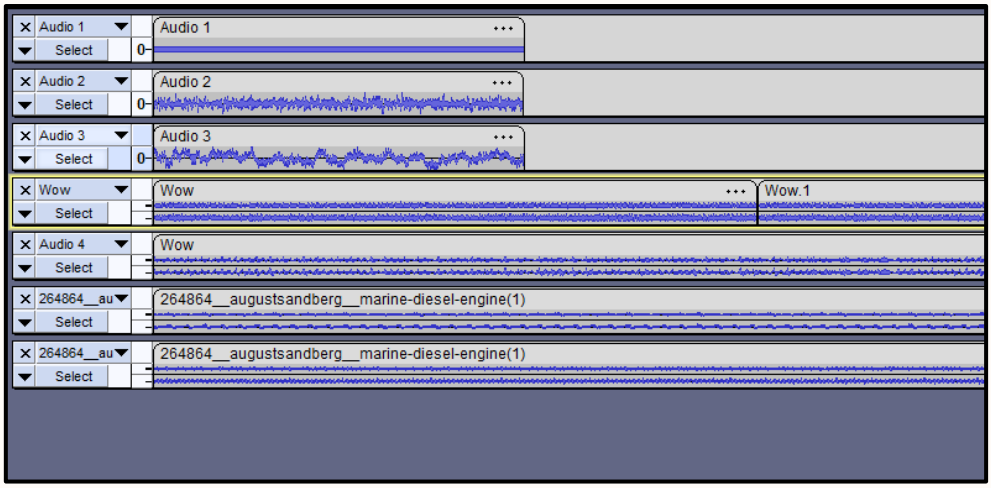
Before trying Logic, Patrick had done a lot of messing around on GarageBand (which is essentially 'Logic but free') so he was already familiar with the program and composition process.

The sound effects were created from a variety of sources - recording, downloading free online sounds, and also creating our own sound effects in **JSFXR**.



Creating sounds in JSFXR, a javascript port of SFXR. Here, we were synthesising explosion sounds to layer into the player taking damage sound effect.

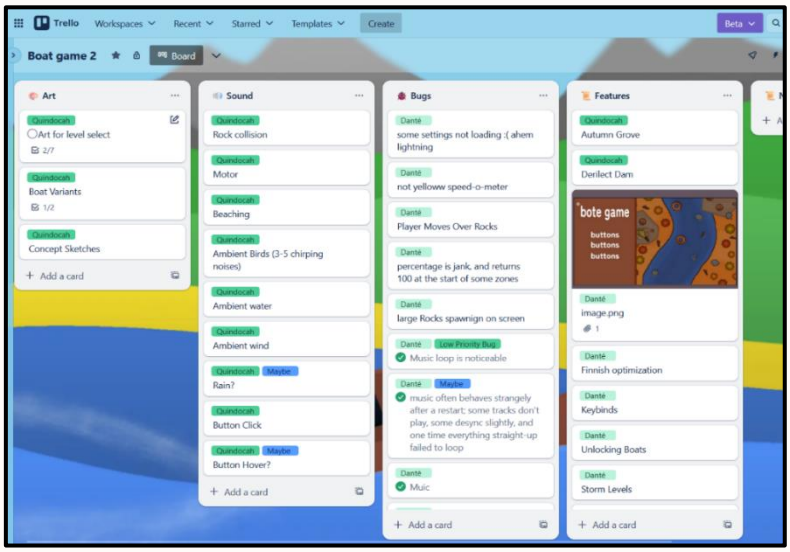
JSFXR is a lightweight sound synthesiser, used by many game designers for creating sounds of the '8-bit' aesthetic or in game Jams. Daniel's familiarity with manipulating JSFXR's parameters meant it was a good choice for this project's sound effect creation, especially considering all sounds created by SFXR are completely unbound to any licencing. However, JSFXR's simplicity meant the sounds it generated often weren't suitable by themselves, so we often layered sounds together in audio editor **Audacity**.



It's become a common theme that layering components is an absolute game-changer. Here, Daniel was layering the 3 components that formed the player engine noise.

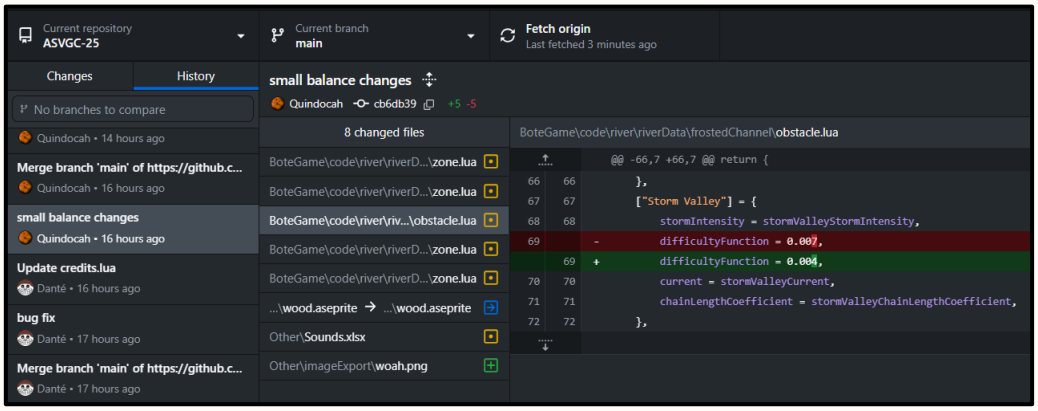
Audacity allows us to manipulate the waveforms of audio files, and add effects like a fade-out. Combining and splicing audio in Audacity is something both Daniel and Patrick were comfortable with, so it was a good choice for sound design in *Buoyant Voyage*.

We don't have a designated team leader, but for this project we needed some sort of management. We chose **Trello** for this, as it was a massive help for our organisation last year. Trello helped us split huge jobs into smaller, achievable tasks. Using Trello's labels, we were also able to keep track of who was assigned to each task.



A Trello Screenshot from early development. Note that at this point in development, we hadn't yet assigned sound design to Daniel.

To store and remotely share our game files with one another, we chose an industry standard, **GitHub**. Previously, we used OneDrive to share game projects and files. GitHub's ability to track and revert changes (something which was sorely missing in OneDrive) which convinced us to switch. This allowed us to create concepts quickly and revert the changes if we didn't like them.



GitHub lets us track and review individual changes down to the specific lines and values changed, vastly increasing our productivity and helping with communication.

New Skills

To us, this project wasn't a time to try huge new things, but instead to create our best ever game by developing and utilising all the skills we'd learnt over the past few years. That being said, we still did plenty of experimenting, and picked up a few new skills:

To make a procedurally generated river run smoothly whilst simultaneously being aesthetically pleasing, Daniel had to learn some of Love2D's more advanced features like **multi-threading** (so that the river generator and main game loop could run independently) and 'canvases' (to efficiently draw the river to the screen). Second, *Buoyant Voyage* was our first time using **GitHub**, and though there was a few initial growing pains (like Patrick always forgetting that you have to commit AND push before his changes could be accessed), we quickly became accustomed to it.

Timeline and Workflow

For a project of this size, it was essential that we do at least a bit of planning on when what aspects of the game need to be done. Year 12 is incredibly busy, however, so we knew we would only be able to tackle large tasks in the school holidays. We created the following **plan**:

- Summer holidays: program the majority of the game's technical systems and create enough gameplay for a 'minimum viable product', a demo version of the game (with far less gameplay and polish) that we would already be happy to submit. This way, we can run early playtests, and later development can proceed quicker given all technical systems are complete.
- School semester: run playtests and slowly work on small tasks (like the level select or high-resolution art) in the spare time we have.
- Winter holidays 2025: finalise the game, write the game dev doc, and submit.

We delegated roles so that many of our tasks didn't overlap, allowing us to work independently of each other. However, when we worked on gameplay features and building together, things got more complicated. As Daniel programmed the systems that Patrick needed to create gameplay with, Patrick would 'lag' behind Daniel, waiting for the technical to be completed before he could start. We quickly figured out that Patrick could do the art and design work while waiting for the technical system, and while Daniel was waiting for Patrick to finish the gameplay, there was always plenty of bug-hunting and side projects (like the settings menu) for him to work on. This way, we were more efficient with our time.

Below is a table showing the approximate timeline of *Buoyant Voyage's* development, beginning at the theme announcement and ending at the submission due date.

White: School Holidays (more productive periods)
Grey: School terms (less productive periods)
Blue: Development milestones

Time Period	Daniel working on:	Patrick working on:
October/November 2024	**Prototyping** various approaches to river generation systems, so that when holidays begin there is a system in place	(**Mainly focusing on school**) Early concepting and design for general gameplay and progression.
Start of December	**Start of Summer holidays** Proper development begins.	
December 2024	**Programming** the river generation system – path creation and background generation. Assembling early menus such as a preliminary settings menu.	Creating concept art and defining the game's **art style**, creating the first **assets** like rocks & player boat.
1st half of January 2025	**Programming** the technical systems to spawn obstacles, scale difficulty, and have multiple sub-biomes in a river. Followed by initial player logic.	Crafting the first level's **environment** and scenery – background generation, trees Creating particle and weather systems to add immersion. Writing the first stage's **music**.
2nd half of January 2025	Creating the game's **UI**; an early title screen and game over menu. Programming **saving and loading** systems for settings and high scores.	**Gameplay**: Using the newly written obstacle systems to create obstacles for the level. Balancing to ensure the game is fun and balanced.
End of January	**Minimum Viable Product Completed.** Development slows down as school begins. *Testing commences.*	
School Term 1 (Feburary to April)	(**Mainly focusing on school**) Fixing and improving the minimum viable product. Early **Playtesting** of the minimum viable product.	(**Mainly focusing on school**) Creating high-resolution art assets, such as the world map. Created a new design for the title screen. Early **Playtesting**.
School Holidays 1 April.	**Reworking** the river generation system based off the flaws found in background generation approach. Including new ambiance system, unlocking rivers. Also changing loading system.	Somewhat **busy** these holidays with family but managed to record some sound effects. Created final high-resolution art assets.

School Term 2 (May to July)	(**Mainly focusing on school**) Making sure all systems and menus are finalised so that holidays can be focused on gameplay programming	(**Mainly focusing on school**) Finalising the design / assets for the second two stages, so that when holidays begin development can start immediately.
Start of July	**Semester Break Begins.** Development picks back up to finalise the game.	
Holidays week 1	Transition zones, setting up endless mode. Assisting in gameplay development through code tweaks that allow for more versatile rivers. Detailing technical systems in the game design document.	Creating the **gameplay** for stage 2 and stage 3. Development for these stages progressed quickly thanks to the preparation work done during the second school term.
Halfway through semester break	**Game is feature complete.** Focus on the game development document.	
Holidays week 2	**Polishing** all aspects of the game, finalising **sound effects** and **bug fixes**, writing the game design document.	**Playtesting**, final additions and **balancing**. Adding the tutorial. Writing and editing the game design document.
End semester break	**Game is complete and compiled on July 20.** Game development document is almost ready for submission.	
Final three days	Final **playtesting** just to confirm there are absolutely no bugs remaining. Creating an itch.io page and submitting the game.	Final editing and formatting for the Game Development Document.

Our time management was generally successful, and allowed us plenty of time to really polish up the game in the last few weeks. Creating a minimum viable product was probably the biggest strength of our timeline, as it allowed us to run playtests super early into development, and ensured we didn't procrastinate essential features until right at the end.

Our biggest regret is not leaving enough time near the end for a final round of playtesting and fixing. We still ran final playtests to check there were no huge crashes, but weren't able to action on any playtester feedback in fear of adding bugs that might slip into the final release. The other thing is that we should've left more time for writing the game dev doc. We felt we'd rather spend time working on the game and that a week for the GDD would be sufficient, but it ended up causing unnecessary stress.

"So are you going to write about how we did everything right at the end?"

"Yeah, but I'll frame it as if we had it all under control."

Game Design: Overview

Theme

This year's theme is "Journey", which we felt was a very broad theme. A journey is 'an act of travelling from one place to another', as defined by Oxford languages. In video games, a journey is often more than the act of travelling, but also overcoming challenges, meeting new people, and maybe slaying a few monsters along the way.

Setting aside the more literal definition, a 'journey' could also be the overcoming of personal or mental challenges. For instance, someone could undertake a 'journey' in accepting themselves for who they are without so much as leaving the neighbourhood. A more famous example of a game featuring such a journey would be *Celeste*, a 2D platformer which cleverly parallels summiting a mountain with overcoming anxiety and self-doubt. Gaining skills, overcoming challenges and accomplishing feats are elements of any real journey, and it so happens that video games often contain these themes too. If you were to push it, you could frame almost any game as a 'journey'.

Buoyant Voyage is a journey from a river's source all the way to the sea - a somewhat unconventional journey, but a journey nonetheless. The player learning how to pilot the boat to overcome the river's many obstacles is a part of the journey too.

We felt that the best types of journeys were adventures, so a big focus of our game's design was really creating that adventure feel. Sub-biomes like the autumn grove (a forest of eternal autumn), the clockwork ruins (an ancient site whose mechanisms are still running), and or the electrical complex (a huge facility filled with turbines and tesla coils) were created to be mysterious and tickle that sense of adventure and exploration. We focused on features like procedural stage decoration, weather systems and atmospheric music to truly immerse players in that sense of adventure and journey.

We could've fit a story into *Buoyant Voyage* to add an aspect of a 'personal journey', but we decided not to, in favour of players creating their own story. After all, this is **your** journey, and we didn't want to impede on the sense of exploration and immersion by forcing players into a narrative which some might not relate to.

Game overview

The Title: "Buoyant Voyage"

As we hope has been obvious so far, our game's title is "Buoyant Voyage". Being the direct successor to an earlier project, *Boat Game With a Name*, our playtesters were initially surprised the game wasn't called *Boat Game With a Name **2**.*

"Patrick, I've got an idea. Why don't we call it 'Boat Game With Two Names'?"

"What's the second name, then?"

"Well... that's not relevant, is it?"

...Two other top ideas for our game's title were "Unboatable" (Portmanteau of 'unbeatable' and 'boat' which Patrick quite liked), or "Another Boat Game". Though, we picked "Buoyant Voyage" in the end for the following reasons:

- *Buoyant Voyage* sounded more professional than "Boat Game With a Name 2"*.*
- The title immediately described the game: *Buoyant* invokes connotations of water and boats, and *Voyage* lets the player know they will be embarking on an adventure or journey.
- It rolls nicely off the tongue, making it memorable.
- 'Buoyant Voyage' is similar to 'Bon Voyage', "a statement used to express good wishes to someone about to set off on a journey" (from Oxford Languages).
- There weren't any other similarly-named games that we could find.

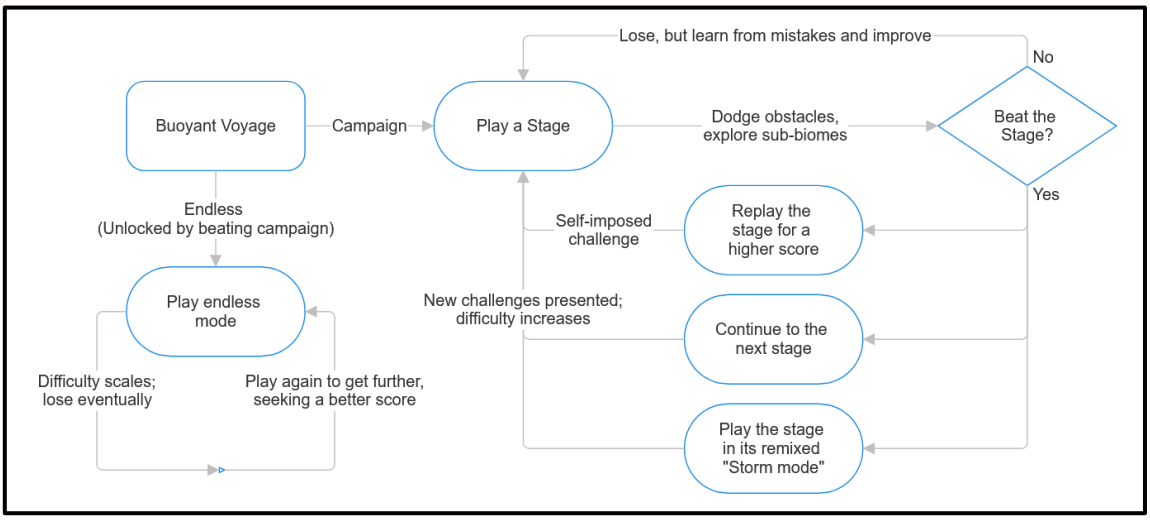
Game description

Buoyant Voyage is a game about steering a boat down a dangerous, procedurally-generated river. The core gameplay is intentionally simple and almost arcade-like: by steering left, right, and throttling up and down, dodge the riverbank and obstacles that fill the river: rocks, giant hailstones, or spinning turbines (a full list can be found in ***Appendix A***). By successfully dodging most of the obstacles in a stage (the player's boat can take four hits before the fifth is game over), the stage is cleared and the next one unlocked. As players progress through the three stages (let's call this the 'campaign'), more difficult obstacles are slowly introduced to scale difficulty.

Buoyant Voyage has one clear goal that follows the natural progression of any river: get to the ocean, thereby completing your journey. Upon beating the third stage ('The Dam') and sailing out into the sea, the end credits roll. This is *Buoyant Voyage's official* win condition, but the game still has more to offer.

Buoyant Voyage offers a "Storm" remix of each level, which is significantly harder in difficulty, so that experienced players are not left feeling unchallenged. There's also an 'endless mode' which cycles procedurally through all the sub-biomes in the campaign, giving the game potentially infinite replayability as players attempt to score as high a score as possible in an infinite river.

Micro-inspiration point: The idea of a "Storm" remix is largely inspired by *Celeste's* 'B-Sides'. In *Celeste*, collecting a tape cassette unlocks a 'B-Side', with remixed music and harder platforming, which builds off the mechanics established in that level's 'A-side'. This gives a challenging extension for more experienced players.



Flowchart: Buoyant Voyage's gameplay loop

The game's progression is visible in the gameplay flowchart above. Notice that there is no 'boat upgrading system' which formed the core loop of *Boat Game With a Name*. We felt an upgrade-based progression system was unsatisfying. Rather than gaining skills and improving as a player, *BGWaN's* focus was on incrementally improving the boat until clearing the level was trivial. Instead, *Buoyant Voyage* focuses on the player's journey in improving; learning from mistakes rather than brute forcing challenges via upgrades. This is why our game has no player progression apart from skill.

One flaw in *Buoyant Voyage's* gameplay loop is that players may be forced to play the same level over and over if they can't beat it. This is where procedural generation comes in – every run of any given stage will be slightly different, so that the game feels less repetitive when players are stuck. Furthermore, as the level changes each time, players are forced to learn how to handle and dodge the obstacles themselves, rather than practicing a set route through the levels.

Intended Audience

Buoyant Voyage is intended for both casual and experienced gamers of all ages. The straightforward controls and low difficulty in the first stage mean that players can quickly learn the game, greatly lowering the entry threshold. As players move toward the third stage, the difficulty increases and learning is required, but we've balanced the game so that casual players should be able to clear the campaign eventually.

At the same time, experienced gamers who seek a challenge shouldn't feel that *Buoyant Voyage* is 'too easy'. Such players can find fulfilment in beating the significantly harder 'storm' levels, scoring high scores or getting deep into endless mode. This is a similar design philosophy to the difficulty in *Celeste;* whilst casual gamers should be able to clear the main campaign, there exists plenty of far more difficult 'postgame' content to keep the hardcore fans entertained.

Due to the implicit randomness in procedural generation and obstacle placement, every run of a given stage could be easier or harder than the previous. Whilst that sounds like a limitation at first, it's a huge strength in terms of appealing to a wide audience. More experienced gamers will find satisfaction in braving highly 'unfair' river generation without taking damage, while a more casual player will eventually get 'lucky' with generation and pass a level with enough tries.

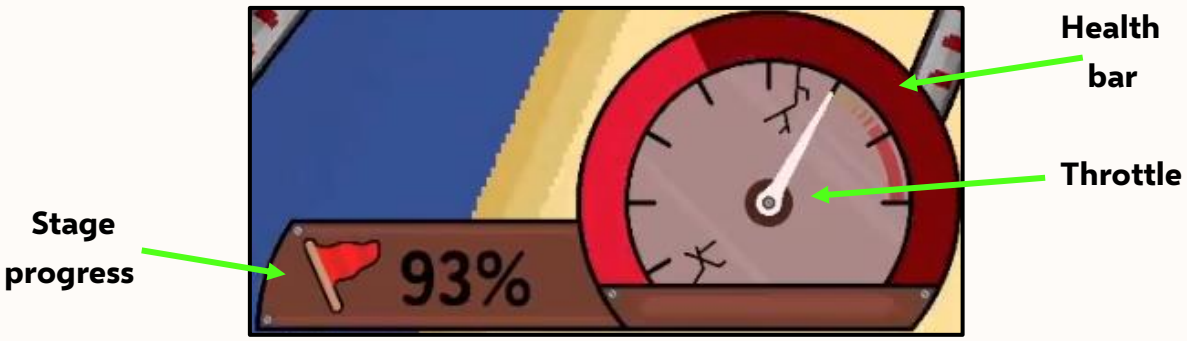
Perspective and User Interface

Buoyant Voyage's perspective is 2D, top down. It's a full bird's eye view, which made it challenging to represent objects like trees. We considered doing a mixed front on / top-down perspective like that of *Stardew Valley*, but we weren't sure it would look right, and neither of us had ever drawn pixel art in that perspective.

Left: Buoyant Voyage, which is fully top-down.
Right: Stardew Valley, which has a mixed front-on / top-down perspective.

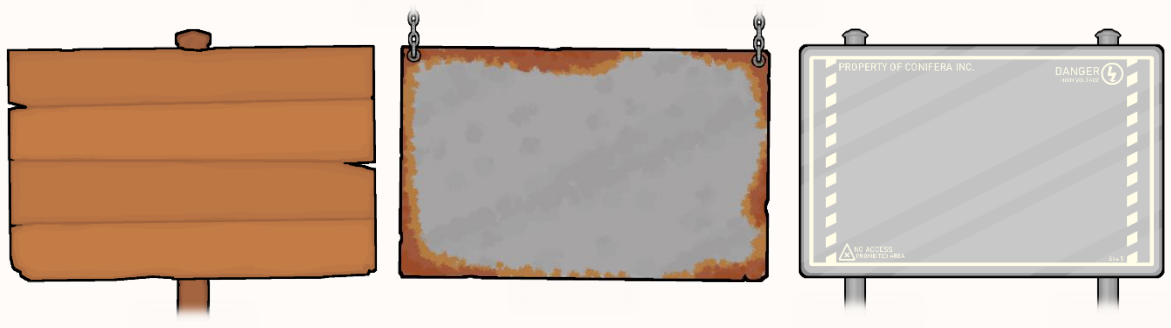
User Interface

Buoyant Voyage's gameplay UI is by design minimal. Our submission to the challenge last year had far too much screen-cluttering UI, so we decided to only display what was absolutely necessary to the player. The UI is styled as a speedometer, with the health bar built into the outer border. There's also a crack effect which appears at low health, as a visual indicator to add some urgency when the player is damaged.



Micro-inspiration point: At low health, the health bar rapidly blinks white and red. This is inspired by *Risk of Rain 2's* health bar, which similarly flashes rapidly at low HP.

The level select user interface consists of clicking on flagposts which open a sign menu with the level's name and play button. Each stage has its own sign which fits with the level's theme, a detail to add just that extra bit of immersion:



We put time into making all the buttons feel tactile and nice to click, improving the user experience. Every clickable button, down to hyperlinks in the credits screen, change colour when hovered over and play a click sound when pressed. The icons next to the buttons on the title screen all have unique hover animations to add some life.

"The play button can be a flag, the settings menu a cog... the quit button?"
"How about an anchor that falls off the screen when you hover over it?"
"Daniel.... Are you insane? That's absolutely brilliant!!"

Movement Controls

Buoyant Voyage has just four simple controls. Their default key binds are:

Steer left/right:	[A] / [D]	or	[Left arrow] / [Right arrow]
Throttle up/down:	[W] / [S]	or	[Up arrow] / [Down arrow].

Steering left or right rotates the player's boat in the relevant direction, whereas throttling up/down changes the power output of the player's engine; giving them some control over how fast they travel down the river. The player's turning speed is not static, but instead calculated by the following equation:

$$Turn\ rate\ (degrees) = 180° \times (1 + \frac{Player's\ current\ throttle\%}{100\%})$$

Which scales the player's turn speed by as they throttle up, increasing the control at high speeds. Secondly, the player's turning angle is clamped so that they can only rotate 75 degrees in each direction, so they can't turn and drive back up the river.

As the current increases or the throttle is raised, the player becomes more manoeuvrable*. There's also a base manoeuvrability of 50 pixels / second so that players have better control at lower speeds in lower currents. This helps players navigate zones with low current but many moving objects, such as the *Clockwork's Core* sub-biome in Stage 2.

**Manoeuvrability refers to how quickly the player can move horizontally across the screen (change their x-coordinate).*

As the river bends, the boat may be pushed left and right towards the river's direction of flow. At higher currents, this effect becomes more apparent. This can feel unintuitive at first but is an intentional challenge of higher-current or bendier sections of the river. A pro-tip: you'll have to counter-steer in the direction opposite to the river's flow to cancel out the current vector.

Lastly, the throttle automatically drops back to 70% if [*Throttle up*] is released. We made this change based off playtesting information (we found this made the throttle feel more intuitive - more on that in the ***Fixing and Testing*** section).

Our game has a keybinds tab, located within the settings menu. This allows players to customise their input scheme. Two keybinds can be assigned to each input.

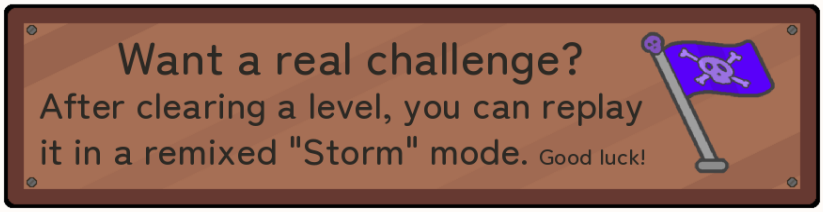
Daniel will be happy if you use his built-from-scratch keybinds menu ;)

Instructions/Tutorials

Buoyant Voyage's tutorial pop-up.

Buoyant Voyage's tutorial is a simple two-line text box in the first sub-biome. Given that our control scheme was so simple, we didn't need much of a tutorial. The beginning of the game presents the player with a wide, empty 'lake' to get a feel for the throttle and steering before being thrown into gameplay. The remainder of the first stage was designed to be quite easy, so the player has time to adjust to the controls and learn the gameplay features themselves, rather than being prompted by an endless stream of text boxes.

The other information feature in *Buoyant Voyage* are the dialogue/info boxes which appear in the level select when the player achieves certain goals. These prompt the player, so that they are aware of all the available levels and features.



Example dialogue box.

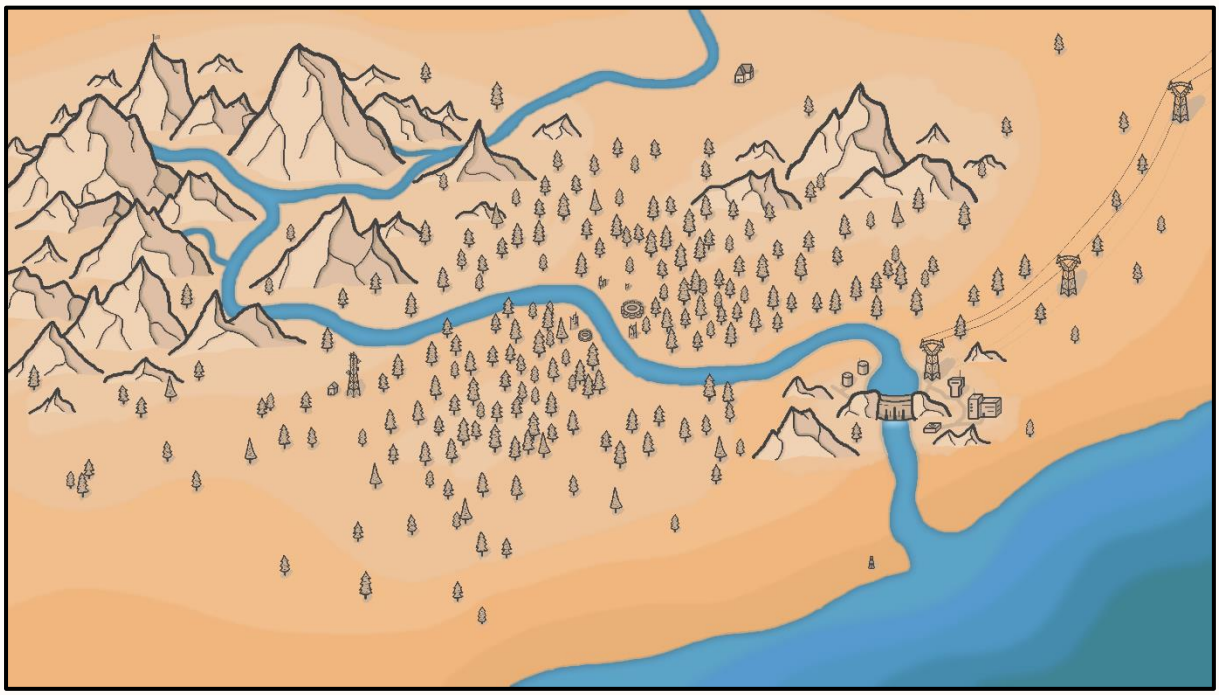
The 'dialogue' boxes which appear in *Buoyant Voyage* are:

- *"Welcome to Conifera! Your goal: sail from the mountains to the sea. Click the flag to begin."* – appears at the start of a playthrough; gives a welcome and establishes the game's goal, as well as how to navigate the level select.
- *"Want a real challenge? After clearing a level, you can replay it in a remixed "Storm" mode. Good luck!"* – appears after beating the first level.
- *"Congratulations! Endless mode has now been unlocked."* – appears after beating Stage 3, 'The Dam', offering them an infinite endless mode.
- *"Thanks for playing! You're a master of the river!"* – secret dialogue which appears after beating all storm levels. 'You're a master of the river!' is a homage to the win screen in *Buoyant Voyage's* predecessor, *Boat Game With a Name.*

World & Levels

As Patrick (that's me!) did the world and level design, this section is written by him. I've used personal voice to make explanations easier to follow.

The world: Conifera



Map of Conifera as appears on the level select.

Buoyant Voyage is set in 'Conifera' a fictional region which, if I had to assign it a real-world location, would probably be somewhere in Canada. Conifera is a beautiful, mountainous landscape forested with pine trees, almost unpopulated save for a huge hydroelectric dam which operates at the bottom of its largest river. *Buoyant Voyage* is set along this river, down which player sails from Conifera's mountains to its oceans.

I would have loved to set *Buoyant Voyage* in Australia, but the main limitation was with the tree type. In a 2D-top down perspective, scraggly gum trees would've been hard to convey artistically, where as pine trees work well in a top-down perspective. Conifers are also my favourite type of tree, so the choice was obvious.

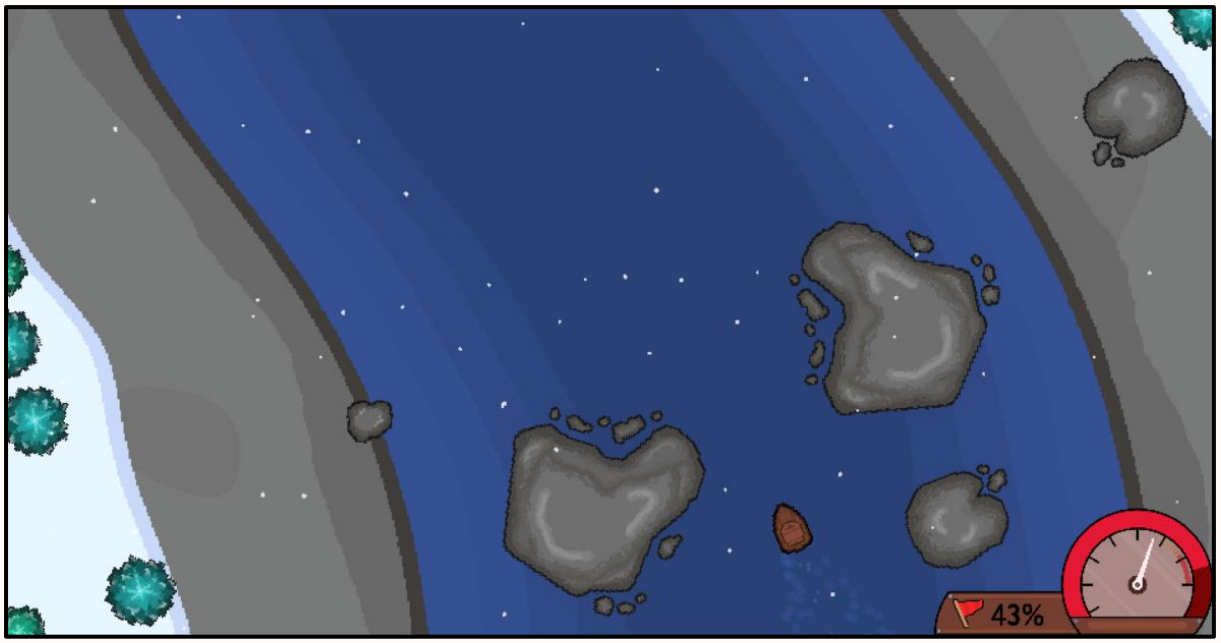
"Patrick. If pine trees are evergreen, then how is there an autumn level?"

"Uhh... I'm sure there's deciduous pine trees somewhere in the world... Would you look at that? They're called 'Tamaracks'. I'm saved."

"Or, we could just put other types of trees in the game."

"No, I like conifers. And they're easier to draw, okay?"

Stage 1: Great Valley

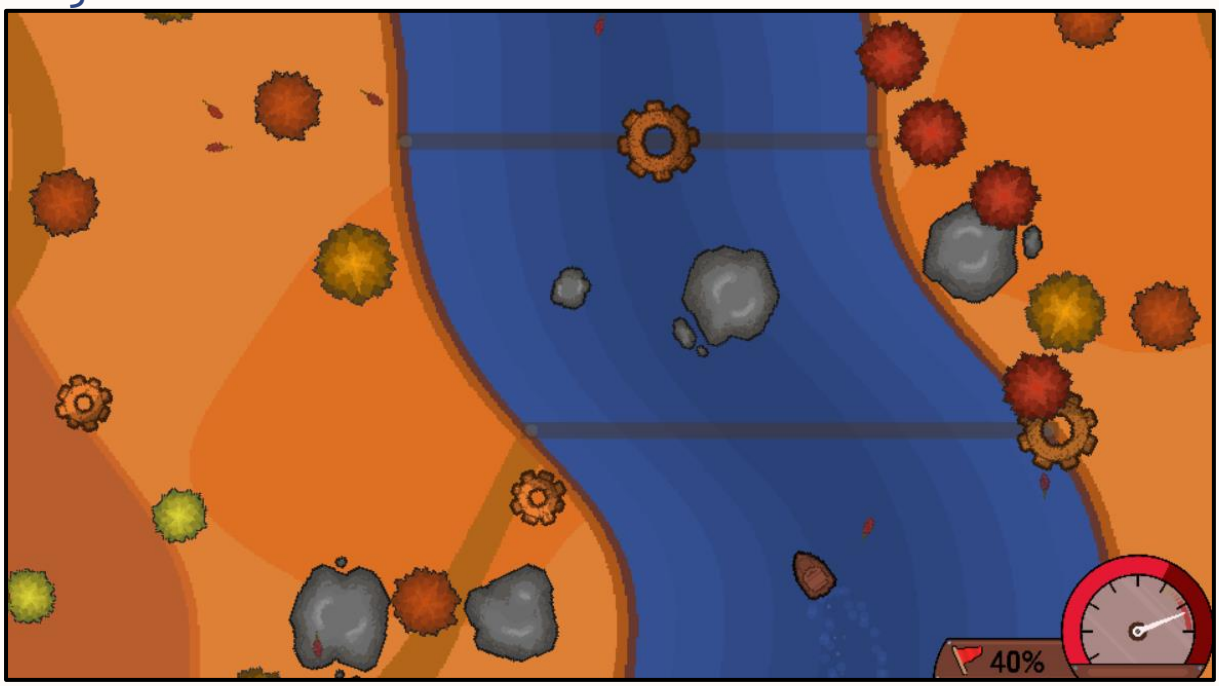


Sub-biome: Boulder Valley

Stage 1: Great Valley is set in the snowy highlands of Conifera. It's an introductory stage which is intentionally quite easy and has few moving obstacles. The entire stage functions as an 'invisible tutorial' which slowly introduces most of *Buoyant Voyage's* mechanics and gameplay over its course. Progression through this stages' sub-biome is as follows:

- The stage begins in the 'glacial lake', a wide, empty section of river for the player to get used to their controls.
- The player then enters the 'ice plains' where river bends and rocks are introduced at a low current.
- As the player moves down the mountain, they enter 'boulder valley'; a wide, stony section of river. This sub-biome contains rock-chains which teaches the player how to squeeze between obstacles.
- The screen darkens, and the snow now blows across the screen much faster as the player enters 'storm valley'. The game's first moving obstacle is introduced: hailstones, small balls of ice which fly across the screen. The current is also increased. We used to have rock-chains in this sub-biome too, but playtesters found these too difficult at higher current – so we removed them.
- As the player clears Storm Valley, they pass the snowline into the wooded hills, a short, calm intermission just before the end of the stage.

Stage 2: Autumn Grove



Sub-biome: Clockwork Ruins

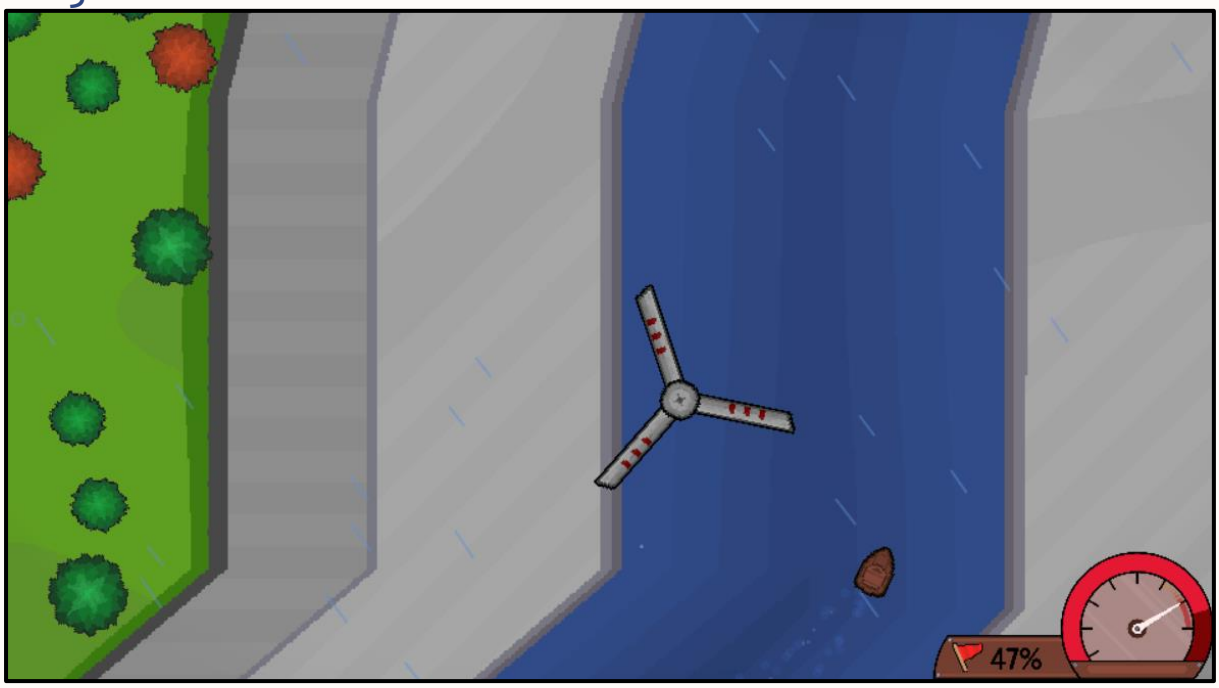
Stage 2: Autumn Grove is set in Conifera's hills, in a patch of mythical forest which is eternally in autumn. Leaves blow across the screen, and the current picks up as the player enters the remnants of what must be an ancient steampunk civilisation. Stage 2 is where the game really opens up; players must deal with moving obstacles, some high current and a bendier river.

- The stage begins in the 'wooded hills', a short, easy introductory section.
- The grass turns yellow and the trees a beautiful read as the player moves into the 'autumn grove'. The river is bendier here, and the current stronger. Here, the effect of the current pushing you side-to-side is much more noticeable than earlier levels.
- The player enters the 'clockwork ruins'. Cogs spin back and forth horizontally along the river, providing a timing challenge.
- Deeper into the ruins is 'clockwork's core'. Huge cogs turn and the player is forced to weave between many different moving parts at once.
- The player exits the autumn grove by the 'autumn rapids', a straightforward, high-current area after the difficult previous zone.

"Shouldn't the pine trees have needles, not leaves?"

"Don't worry about it."

Stage 3: The dam



Sub-biome: The Inlet

The final challenge between the player and the ocean is Stage 3: The Dam; a huge concrete construction of . A light drizzle of rain falls as the player braves the most dangerous obstacles yet.

- The stage begins in the 'gravelly plains', a windy, rocky section of river. We didn't want to start this level in the dam, so this brief intermission exists.
- The player enters the 'upper dam', a rock-filled reservoir with rows of spinning turbines that the player must time correctly to pass unharmed. The river here is far more 'artificial' with straight lines instead of curves.
- The current picks up and the river gets thin as the player passes into 'the inlet', a super high-current, thin section of river. Turbines spawn near the sides of the river, forcing the player to quickly react.
- At the bottom of the inlet is the mysterious 'electrical complex', with more rows of turbines and huge tesla coils that fire spirals of energy orbs.
- After clearing the complex, the concrete fades to sand and the player exits the river through the river mouth. Congratulations!

Micro-inspiration point: The rain effect is inspired by the rain effect in *Stardew Valley*, which similarly has droplets that fall diagonally across the screen even in a top-down perspective. Even if the perspective is technically wrong, the rain effect still fits.

Game Design: Visual & Audio

Visual Design

Buoyant Voyage's primary visual style is pixel art. Pixel art is a versatile artstyle used and associated with many video games, emerging in the late 1970s due to graphical limitations. *Buoyant Voyage's* gameplay is also somewhat arcade-like, which the pixel art style's arcade connotations help convey. The pixel art resolution of *Buoyant Voyage* is [640x360]. As the game's native resolution is [1920x1080], each art 'pixel' is worth 9 'real' pixels.

Pixel art actually wasn't the original artstyle we had in mind, but instead one we were forced into due to the river generator's limitations. We had originally planned a smooth, vector art style with few colours, but generating the river background at fullscreen resolution would've been too slow to be playable. As it would've been weird to have smooth assets on a pixelly river, we decided all of gameplay would follow a consistent pixel-art style.

An early concept 'mock-up' of Buoyant Voyage's pixel artstyle. We actually ended up keeping the cogs and trees from this concept in the final game.

Outside of gameplay, *Buoyant Voyage's* art follows a low-detail, high-resolution artstyle, present on the world map, level select signs, and the game's buttons. We felt that pixel art fonts and menus inhibited readability, so instead went for smooth-art menus. We believe it isn't necessarily wrong to have multiple artstyles within a project. *Celeste* does this too, combining pixel-art gameplay with a 3D-modelled level select.

Process: Gameplay (pixel) Art

This section written by Patrick

To make *Buoyant Voyage's* pixel art, I would start by choosing the colours for the given image. For smaller assets like the hailstones in Storm Valley, I could easily draw the whole sprite at once, but I took a more methodological approach for bigger sprites. I would start by defining the asset's shape, before layering details step-by-step.

I then shaded sections which were 'lower' to give the illusion of depth. As sprites like rocks have multiple image variants for variety, I had to use standard colours, line widths and such so there weren't inconsistencies. Top-down 2D is a somewhat limited perspective when it comes to conveying information (especially prevalent with the tree sprites).

Four screenshots showing the process of creating one of the 'huge rock' assets. Left to right: Shape, outline of height, spaces for main details, final touches.

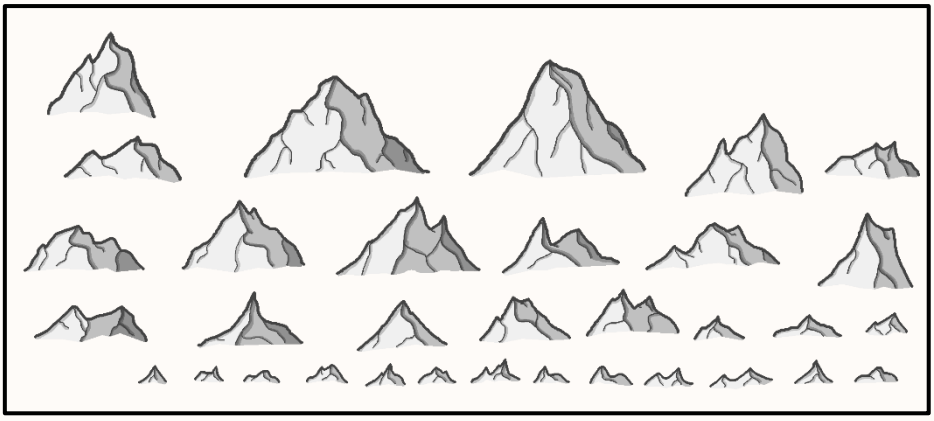
I'm happy with the art in *Buoyant Voyage;* it's consistent, fits the arcade gameplay, and is much more professional than the art in *Boat Game With a Name.* Though, I've had a playtesters say it's a bit un-expressive; I agree. In the future I might consider learning proper dithering, or choose more vibrant colour schemes.

Process: Level select art

This section written by Patrick

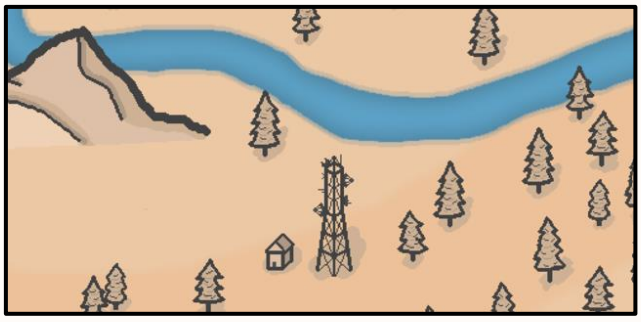
I thought I'd do something a bit different for the level select and make a fantasy-style map. Maps of this style appear across adventure games, fantasy books and DnD campaigns: I picked this artstyle for the world map, as it had inherent connotations with adventure and journey.

Following fantasy map convention, I picked a beige, weathered paper colour scheme for the map, where lighter and darker shades of beige could represent different elevations. As the river is *Buoyant Voyage's* most important feature, I've highlighted by colouring it a vibrant blue, so its salient against the rest of the beige map.



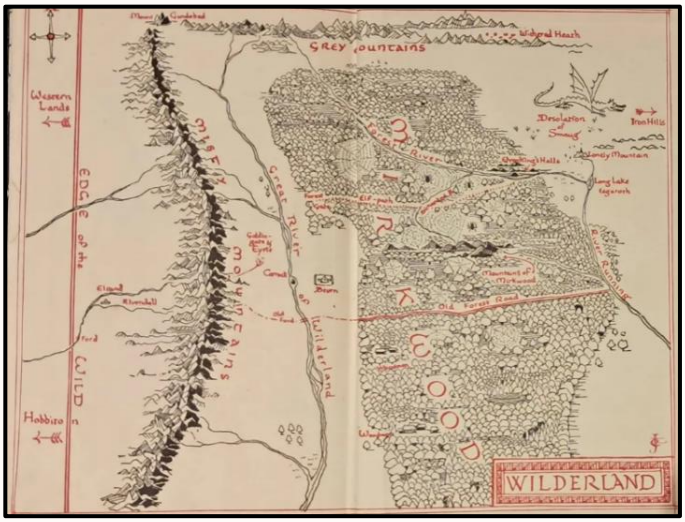
All the mountains used on the world map, drawn separately at first.

To give the mountains depth, I shaded their right side to give an impression of the mountain casting a shadow. Trees and other objects also have translucent shadows pointing in the same direction, so the map feels more consistent and real. I made sure to scatter a few details and man-made objects here. I love little unexplained details like this; they add just that extra bit of adventure feel for those who are observant.



Zoomed in: small details like this radio tower add some life and interest to the map.

Micro-Inspiration Point: The level select was inspired by the wonderful maps of J.R.R. Tolkien, like this one which is in the back cover of my copy of *The Hobbit:*



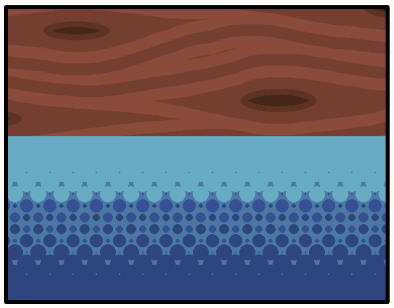
Process: Title art

Left: Old title art, used as a placeholder during development.
Right: Current title art.

This is one of the few art assets in Buoyant Voyage which was made by Daniel. The *Buoyant Voyage* title art is based off the original one which Patrick made for our initial build/play tests. The text is split into two textures, a wood grain for the word 'buoyant', and a water texture for the word 'voyage'. The woodgrain colour comes directly form the player, while the colours chosen for the water are the darkest part of the river in our game, and a brighter blue colour from our previous game.

Not only do the different textures add interest to the title art, but they also represent a boat and a river. The wood grain and stylised O (which looks like a boat's steering wheel) in the word "Buoyant" sits atop the watery blue "Voyage". This helps players deduce the main gameplay loop, steering a boat down a river, from the title art alone.

To create this banner, Daniel started by adding the base text. Next, the wood grain texture was created from oval shapes that have been expanded, with points. The water takes inspiration from the dithering within the original banner. A simple 3D effect was added by cloning the text below multiple times then darkening it.



The textures Daniel made, from which the title art's logo is 'carved' out of.

Our banner needs to pop on any background, so an outline and shadow has been added. The outline is a copy of the text, with a 90% brightness reduction, and 17x blur that has had its alpha channel increased to remove act as an anti-aliased boarder. The shadow is similar, only void of colour, offset, and transparent.

Audio Design

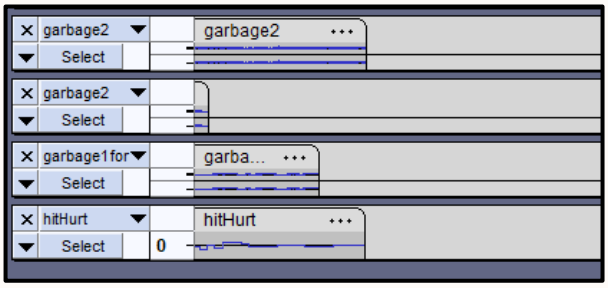
The sounds in *Buoyant Voyage* are often mixes of many sounds from separate sources:

- Recordings taken on our phones. These include some very makeshift foley, and field recordings taken by Patrick of birds chirping and water flowing
- Sounds generated on SFXR. As our game has a pixel art theme, the "8-bit" sounds synthesised on SFXR fit quite nicely.
- Downloaded sounds from freesound.org, because there's some sounds we just couldn't make or capture ourselves (like thunder strikes). We downloaded three sounds, and they're all attributed in the credits and ***Appendix D***.

Much of sounds came from a pile of sticks in Daniels back yard. These were our makeshift foley and were used as components within the beaching and collision sounds. These sounds actually turned out quite well with some editing.

Sticks recorded to create player sounds.

The players collision sound was initially made of three recorded stick components, and an 8-bit hitHurt generated though JSFXR, all mixed in Audacity.

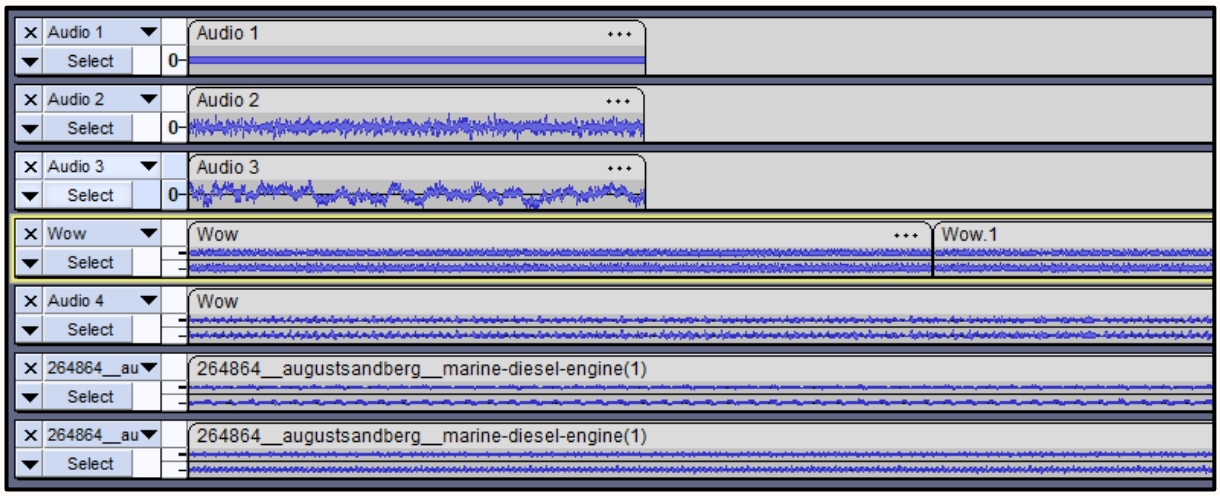


Player collision Audacity file

Since then, an extra 'thud' downloaded from freesound.org has been added on top to give the sound that extra bit of "oompf". After all, each point of damage is 20% of your total health, so it should be a deep, impactful sound.

The player motor sound is comprised of three parts, which are dynamically mixed in-game depending on the player's speed. These three parts are:

- White noise, generated in audacity. This provides some base volume which helps conceal any artifacts with the sounds looping.
- A downloaded 'diesel motor' from freesound.org. The engine sound didn't sound quite complete or real, so we sourced an actual engine recording.
- And lastly, Patrick screaming into a soft drink can, recorded on his laptop microphone. Believe it or not, with some editing, this became an incredibly convincing engine-revving sound.



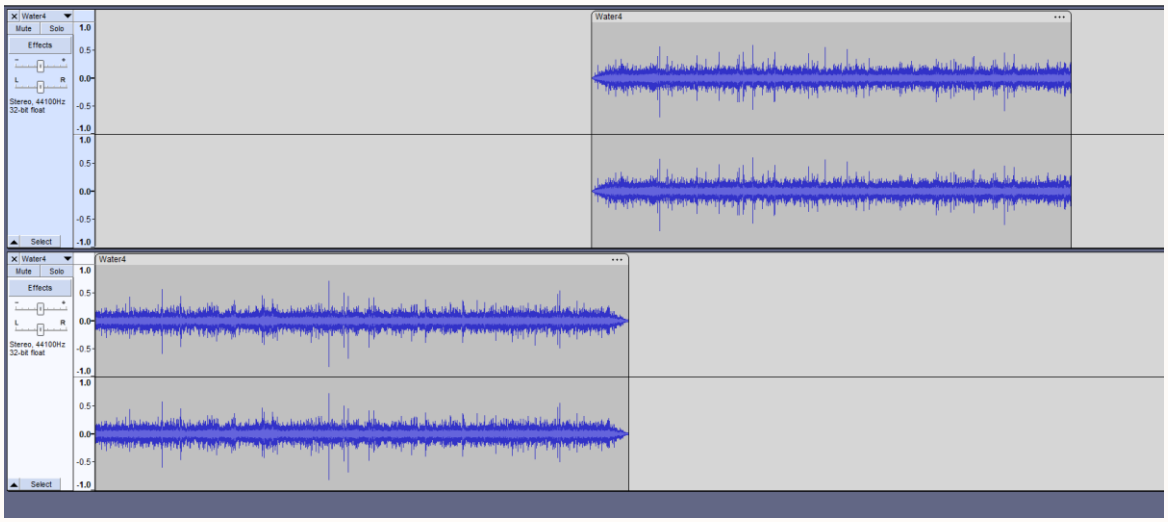
Player motor sound layers in Audacity.

To make the engine sound convincing, it wasn't enough to just export this sound as one file and then change its pitch/volume as one block in-game. Each track's pitch/volume is adjusted differently to make the engine sound more realistic. For example, the engine-rev sounding layer has volume and pitch functions of:

$$volume = \left(\frac{player.speed}{4 \times player.maxSpeed} + 0.2\right) \times volumeMult$$

$$pitch = 1 + \frac{4 \times player.speed}{player.maxSpeed}, \text{ g}$$

Partick recorded the water sounds at a nearby creek. To create the seamlessly looping, the original sounds were split in half and the halves offset so now the sound's beginning and end occurred in the middle. Using the fade-in and fade-out effect, a cross-fade between the sounds provided an almost-seamless loop.



Creating water sounds in Audacity: split, rejoined and faded.

The ambient bird sounds found in the level select were recorded by Patrick deep in Cleland National park, a short drive from where he lives. Originally, we had these play randomly in forest areas, but with the engine sound, water ambiance, and music in the background, there were already sounds. Instead, we moved the bird sounds to the level select.

Similarly, we had considered adding a wind sound to make the storms feel more immersive. Again, this made the soundscape too busy. Considering the storm music is already atmospheric, we didn't feel it was necessary and cut this feature.

Some other audio-related considerations

Buoyant Voyage's audio manager has a system to automatically vary the pitches of any sound played, to avoid audio fatigue. Playing one sound over and over would get annoying quickly, especially if it happens as often as taking damage. For this reason, most sounds within *Buoyant Voyage*, have multiple variants and can play at different pitches.

While editing all sounds in Audacity, we had to ensure each track started and ended on a 'zero' point. Failing to do so resulted in an awful popping sound whenever the offending audio track was played or ended.

Music Design

This section written by Patrick

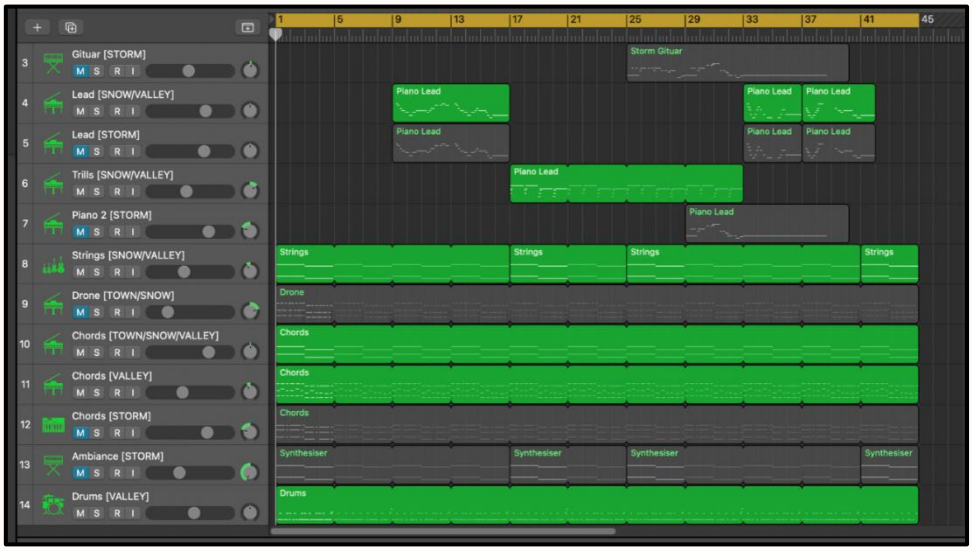
My main goal with *Buoyant Voyage's* music was to create an immersive and atmospheric soundtrack; another design component which works to make the player really feel they're on a journey. Each sub-biome has its own instruments that contribute to that biome's feel. Forest areas have calm and serene piano tracks, whereas the music in Storm Valley utilises synthesisers and muffled drums, and a little bit of messing with the EQ to create a lost, stormy 'blizzard' feel.

Aside from being atmospheric, the three gameplay tracks are also entirely adaptive. One of my absolute favourite things in any video game is adaptive music, where instrumental tracks fade in/out or change volume depending on what situation the player is in. Rather than giving each sub-biome its own music track, the music in *Buoyant Voyage* dynamically adapts based on what sub-biome the player is in. Each stage starts with just a simple piano track with some repeating chords, but more tracks with different instruments are layered to build up to the climax of the level.

Component tracks	**Volume at different zones**			
	Wooded Hills	Autumn Grove	Clockwork Ruins	Clockwork's Core
"Forest chords" – Piano playing the repeating A-G chord progression in shifting triad chords.	**1**	**1**	0	0
"Forest strings" – track containing a simple, repeating backing played by string instruments.	0	**0.7**	0	0
"Clockwork chords" – similar to forest chords with a synth and cymbals to build intensity.	0	0	**1**	**1**
"Clockwork drums" – contains a drum beat (without cymbals) and a clock ticking sound.	0	0	0	**1**
"Forest lead" – Piano track containing the main melody, at a higher octave.	0	**1**	0	0
"Ruins lead" – Piano track containing the main melody, at a lower octave.	0	**1**	**1**	**1**

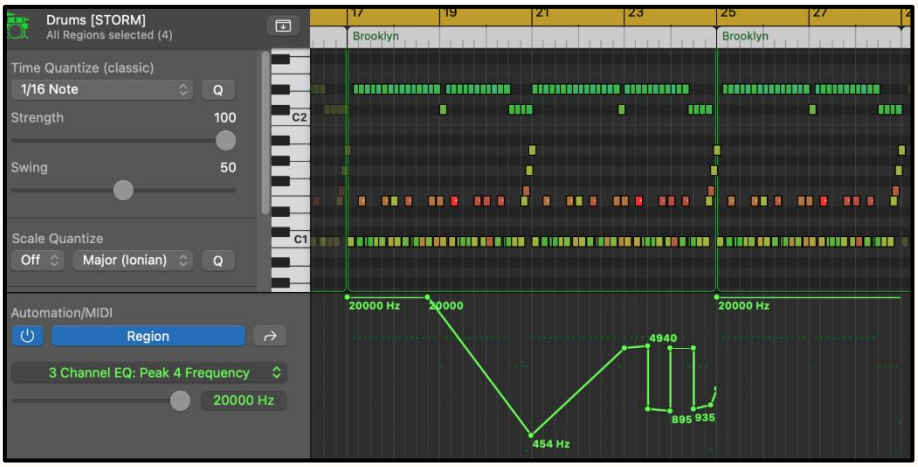
Example: How the volumes of the 6 different tracks in Stage 2: Autumn Grove change as the player progresses through the first 4 sub-biomes, to create a dynamically shifting music track.

I composed the music in *Logic Pro* using a mini (2-octave) midi keyboard for all inputs, including drum beats. I composed melodies within the minor key; this gave the calmer sections a serene, quiet feel, and the more intense sections a desperate, somewhat anxious feeling. To give continuity between levels, all of the gameplay tracks follow the same repeating A/G chord progression, though they all have different melodies.



Logic Pro screenshot for the music from Stage 1: Great Valley. Not all the tracks are used at once, as the game dynamically swaps between them.

I mixed tracks so that different instruments are panned on different ears, as this makes tracks feel more 'real' and thus atmospheric. When I wanted to more advanced effects, I dipped into logic's EQ and 'automation' systems.



Automation nodes used in the drum track that plays in Storm Valley.

This automation channel controls the frequency at which EQ peak 4 operates. As the EQ peak moves, different frequencies of the drum track are amplified and distorted. This is one of my favourite effects and gives a spacey, somewhat disorientating effect which fits that blizzard which I'm trying to encapsulate perfectly.

GAME DESIGN: RIVER GENERATION

As Daniel did the river generation work, this section is written by him. Like the level design section, he's used some personal voice for ease of understanding.

The river is one of the fundamental aspects of *Buoyant Voyage,* giving the game its shape. Unlike in *Boat Game With a Name* (*BGWaN*), we wanted our backgrounds and river to be entirely procedurally generated, from how the river curves to what obstacles go where, all at 60 FPS. To achieve this, I have made use of Love2Ds multithreading. *Buoyant Voyage* runs on two threads, a main thread and a generator thread, something neither of us has done before. The generator thread is used to generate the river segments, the background images, and zones in the case of an endless river, while the main thread handles everything else.

The Procedural Generation Thread

The limitations of multithreading mean that generator can only communicate with the main thread though certain channels that we created. These channels are as follows:

- **background_closeThread** – A flag received by the thread to notify it to stop what it is doing and close.
- **background_closeThreadReceived** – Confirms that the thread is closing to prevent any double river bugs that have occurred in the past.
- **generator_playerY** – The player's y position, tells the thread how far ahead to generate zones, segments and background data.
- **generator_riverData** – The name may allude you however this only sends the river name, as sending all the data would be more resource intensive than it has to be.
- **generator_seed** – A random seed value to prevent the same river from generating twice. The "os.time" function is also used when setting the seed however that does not provide enough variance for our use case.
- **generatorThread_backgroundImageData** – Sent to the main thread, containing a table of pixel colours to be re-constructed by the main thread. (Images cannot be directly sent though channels.)
- **generatorThread_minZones** – A list of zones generated or not for the main thread to use for distance calculations, zone titles and other general information.
- **generatorThread_requestBackground** – Received on a window resize, to re-generate the entire background to remove any missing or removed sections.

- **generatorThread_riverSegments** – The next segments river points, sent to the main thread to be added onto the main river.
- **generatorThread_scale** – The y scale required to create the initial river points below what the player can see on their screen.
- **generatorThread_screenWidth** – The width of the screen, used in creating background images that precisely fit the screen and not spend extra time on unnecessary calculations.

Perhaps, not all of these channels are necessary, scale and screen width could pass as one channel, however, these all work together to form a robust first attempt at multi-threading, leaving no room for unexpected errors.

Zones

One of the first things the thread does is get the zone (the internal name for a sub-biome) list for the current river. These lists can be found at:

"code/river/riverData/[riverName]/zone.lua",

and contain the internal zone name, display name, subtitle, distance, and zone order. In the case of an infinite river, the zone file also controls a transition variable along with a function to select the next zone. Upon loading a zone, if the river is finite, it is simply sent to the main thread and zone processing stops there. Endless rivers only generate zones 10000 pixels in advance, re-sending the zone list each time a new one has been generated.

The Endless Function

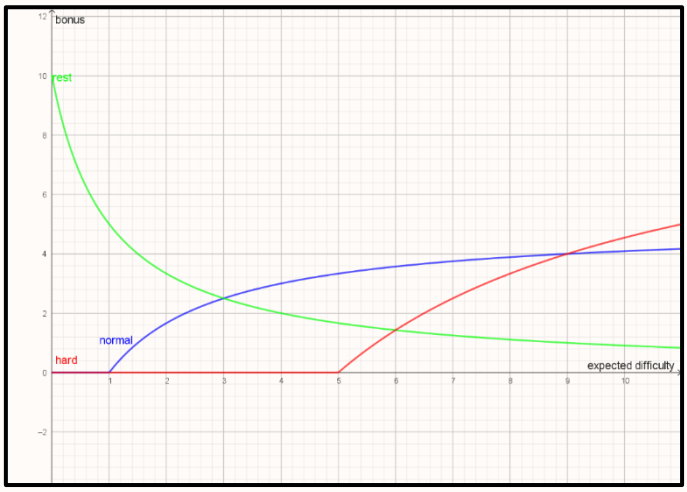
The endless function selects the next zone within endless mode. Simply picking the next zone results in unfair chaos, with a chance to spawn right next to all the most difficult zones. To solve this, the endless function was created, increasing the difficulty as the river continues. This function also has a continuity bonus, meaning similar zones are more likely to spawn next to each other.

When selecting the next zone, a number of factors are taken into account including zone difficulty, expected difficulty and a continuity bonus. If the previous zone was a "rest" zone, the expected difficulty increases by three points, a "hard" zone decrease it by five, and any other zone provides a +0.2 to the expected difficulty.

```lua
local expectedDifficulty = 0
local continuityBonus = 7/5
local function calculateWeight(self)
    if self[1].difficulty == "rest" then
        expectedDifficulty = expectedDifficulty + 3
    elseif self[1].difficulty == "hard" then
        expectedDifficulty = expectedDifficulty - 5
    else
        expectedDifficulty = expectedDifficulty + 0.2
    end

    for i = 1,#zoneDefaultWeight do
        local z = data[zoneDefaultWeight[i].name]
        local w = 1

        -- difficulty bonus
        if z.difficulty == "rest" then
            w = w + 10/(expectedDifficulty+1) - self[2]/25
        elseif z.difficulty == "hard" then
            w = w + math.max(-60/(expectedDifficulty+1) + 10 + self[2]/20, 0)
        else    -- should default to normal
            w = w + math.max(-10/(expectedDifficulty+1) + 5 + self[2]/15, 0)
        end

        if z.area == self[1].area then
            w = w*continuityBonus/(count[z.area]/count.total)
        end
        if z.zone == self[1].zone then
            w = w*0.1
        end

        zoneDefaultWeight[i].weight = math.max(w, 0)
    end

    return zoneDefaultWeight
end
```

Code snippet: The endless difficulty function

Each zone starts with a weight of 1 that is modified based on the expected difficulty with 3 separate functions, one for each difficulty, the previously mentioned continuity bonus (multiplier) and a 0.1 multiplier if it is the current zone. This reduces the chance that the same zone appears twice in a row, without preventing it, and encourage zones with similar colour pallets and obstacles to appear next to each other. Below is a graph of the weight bonuses given to each difficulty based on the expected difficulty value.



Geogebra graph showing the distribution of sub-biomes of different difficulty as the player progresses farther into endless mode.

Both normal and hard have been clamped to a minimum of 0, so that the bonuses never fall below zero and cause issues with probabilities.

Segments

The river is formed from many segments, that can be generated through various parameters; minWidth, maxWidth, segLenMax, segLenMin, maxDeviation, offsetX, and "chunky". The first four are rather self-explanatory and maxDeviaion is how far to the left or right the river's centre can change by per segment, only limited by the screens size. The offsetX parameter is only used in the title screen river, moving it to the right. The newest addition is chunky, that lets the river generator know the depth to render each curve at. By default, chunky is set to 4, however zones such as The Inlet use a value of 1 to create the pointier sections. Each of these parameters are based on the current zone at the start of the segment and are found at:

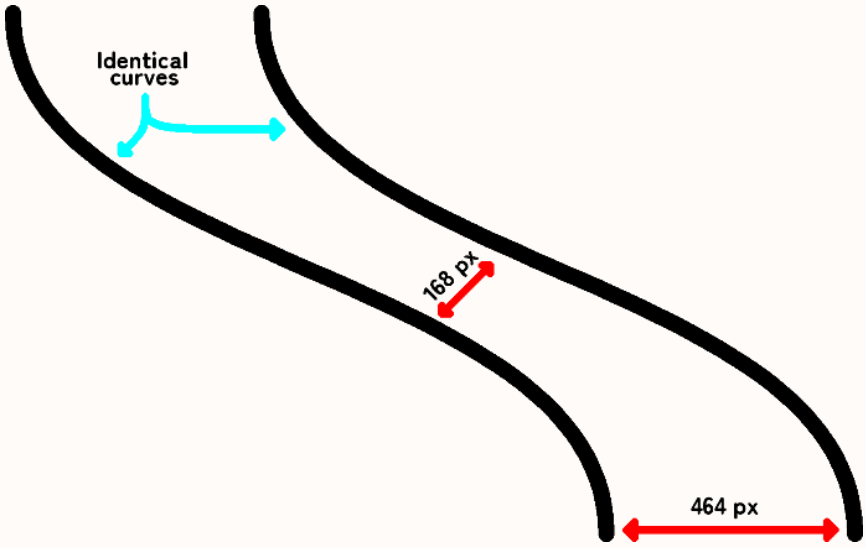
"code/river/zone/[zoneName]/ pathGeneration.lua".

Each segment is sent to the main thread and kept on the generator thread to be used for background generation.

Not all as Planned

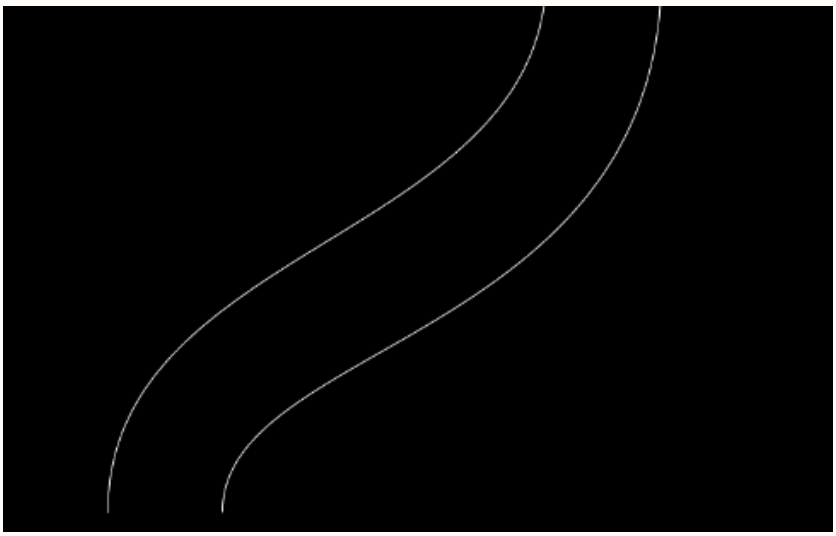
Each segment is currently generated through a series of bezier curves, however originally random noise values were used. Each side of the river had a slight offset to change the bend. This was rather limiting; constantly swinging from one side back to the other with no rest in between.

It was only when I was looking for an alternate option that we came across bezier curves. Each bezier curve has four "control" points, the last position, center 1, center 2. and the final position. Even so, an obvious 'shearing' was obserbvable, where steeper sections of the path were noticalby narrower, as depicted below.



Issue: sections of the river that bend sharply experience shearing and are thinner in width than straight river sections.

My next thought was to adjust the points manually. This worked for the most part, creating a river with almost no shearing, however some points would end up in the river, and even after attempting to delete such points, some remained (bottom left). Eventually I landed on a solution that worked for us, adjusting the control points on each curve directly based on the average angle of each segment. Although it does have some shearing, it is mostly unnoticable at the scale the player observes.



Improved river generation by adjusting Bezier curve's control points.

```
-- adjust the middle sections to reduce the effects of squihsing
    -- imagine the while river is rotated (traveling right) so we flip the x and y axis in atan2
local angle = math.atan2(startX - endX, -segLegnth)
    -- trun the angle into a percentage (0-1)
local anglePercentage = angle/(math.pi*2)
    -- finally change the y values

local minAmount = math.min(midWidth*anglePercentage, -midWidth*anglePercentage)
midLeftY = midLeftY + midWidth*anglePercentage + minAmount
midRightY = midRightY - midWidth*anglePercentage + minAmount

endLeftY = endLeftY  + minAmount*2
endRightY = endRightY + minAmount*2
```

Code to adjust the control points on our curve.

Background

So now we have a river, but we have to put an image to it, so it isn't just a line in a black voice. Each zone has its own background generation code, located in the zone's folder that has access to a couple general helper functions and the generated percentage though the zone. This lets us create custom transition zones, like the one between Ice Plains and Boulder Valley where the mountains "move" onto the riverbank. If a zone contains its own transition distance, a generic noise transition is also used.

```lua
local function GetColourAt(x, y)
    local colour = {0,1,1}

    if getDistToEdge and getDistToEdge(x, y)/100 > 0 then
        return {0,0,1}
    end

    return colour
end

return GetColourAt
```

Example background generation used in testing, fills with cyan unless the position is outside the river, where blue is used instead.

The river generated by this function is very simple (and quite ugly). The functions used to generate Buoyant Voyage's backgrounds more complicated than this, utilising an 'elevation' parameter and multiple noise values to create riverbanks and hills. Oh, and they have a lot of if/elseif statements too, because Patrick wrote all that code.

Our background generation approach, which uses the same noise values each time, has the byproduct of every background would be the same if not for the changes in river points. By default, the background is generated 25 layers at a time unless a "background request" is received.

On the Other End

When a background segment is received though the river generator, it makes a call to the river, telling it to add the segment to its list of images.

```lua
function River:addBackgorundFromData(imageData)
    if resizing >= 0 then
        resizing = -1
        self.backgroundImages = {}
    end
    local tempImage = love.image.newImageData(imageData.width, imageData.height+1)

    for y = 1,#imageData.pixles do
        for x = 1,#imageData.pixles[y] do
            local c = imageData.pixles[y][x]

            tempImage:setPixel(x-1, y-1, c[1], c[2], c[3], 1)
        end
    end
    local finalImage = love.graphics.newImage(tempImage)
    finalImage:setFilter("nearest", "nearest")

    table.insert(self.backgroundImages, {y = -imageData.y, image = finalImage, x = -(imageData.width/2)*pixlesPerPixle})
end
```

First, we check if the player is resizing their window, if so, the entire background is removed as we are re-generating it to fit their new resolution. Next an image data is created and feed all the points and colours required, before being converted to an image. It is then added to the river's image list along with y and x positions. When an image goes off screen it is promptly removed to save on ram and other resources. If the player were able to go backwards, this would be a problem.

Improvements

Currently, each pixel must find its own distance to edge through a rather complicated function of loops and if statements (see below). A better solution would be to feed in either the two edge points or the river's width and center. This would have made all these calculations redundant appart from a simple subtraction of the pixles x position and one comparison. However, with prospects of multi-channel rivers, this was not as viable as it would be now with only one channel.

```
-- functions for generating the background
local function FindHighAndLowPoints(channel, side, yPos)
    local high, low

    for point = 1,#tempRiver[channel][side] do
        if tempRiver[channel][side][point].y < yPos then
            low = tempRiver[channel][side][point]
            high = tempRiver[channel][side][point-1] or tempRiver[channel][side][point]
            return high, low
        end
    end

    --just guess at this point
    return tempRiver[channel][1][1], tempRiver[channel][1][2]
end

function getDistToEdge(x, y)    -- global so we can acsess in generating colours scripts
    if tempRiver and #tempRiver > 0 then
        for channel = 1,#tempRiver do
            local leftHight, leftLow = FindHighAndLowPoints(channel, 1, y)
            local rightHight, rightLow = FindHighAndLowPoints(channel, 2, y)

            local leftPercentage = (y - leftLow.y)/(leftHight.y - leftLow.y)
            local rightPercentage = (y - rightLow.y)/(rightHight.y - rightLow.y)

            local leftX = leftLow.x + (leftHight.x - leftLow.x)*leftPercentage
            local rightX = rightLow.x + (rightHight.x - rightLow.x)*rightPercentage


            return math.max(leftX - x, (rightX - x)*-1)

        end
    end
end
```

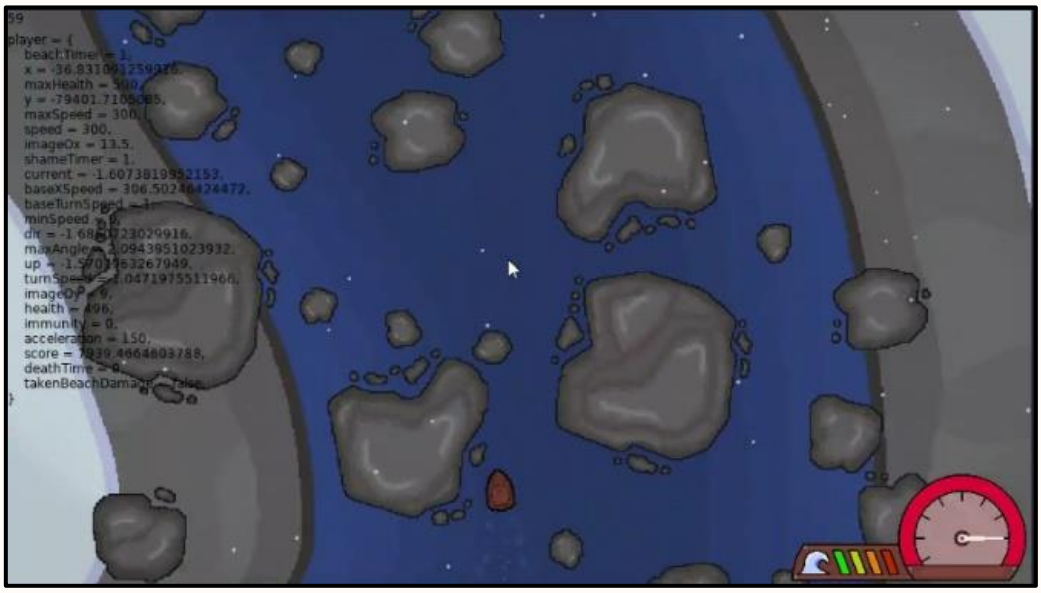
Obstacles

We've done it, a procedurally generated river. But... there's nothing in it! What fun is that without obstacles in your way?

When first loading a river, an obstacle spawner is created for every zone. It is passed a table of all obstacles that can appear, this contains a weight, weight change, noise and a noise divider for each obstacle. The weight controls the base chance for it to spawn, that is added to a noise value multiplied by the weight change. The noise value is calculated with the noise entry in the table and the noise divider. Giving two obstacles the same noise and noise divider will result in their weight scaling at the same rate, if you wanted two things to spawn with similar frequencies. However, we never ended up using any of the noise functionality in obstacle spawning.

Limitations of the Random Spawner

The obstacle spawning system went through a variety of iterations. The initial process of generating obstacle was having random number generator pick when to generate them, y value the player passed and comparing it with a "difficulty" function, defined in the current zone. If an obstacle was to spawn, it would calculate the weight of each one, including a variance then select one randomly based on the calculated weight of all possible obstacles. This random generation caused sometimes where the river became almost impossible to traverse and so needed to be changed.



A stroke of terrifically bad luck made this section of the river all but impossible. Clearly, we needed to improve the spawning algorithm.

The solution: 'Multi-Spawner'

We still wanted some obstacles to spawn completely randomly, however larger ones had to be put on a timer. The solution we came up with was giving each zone the possibility to have multiple spawners and creating different types of spawners for each one. This way we can have smaller obstacles spawn randomly, and larger ones spawning on a close to set timer. This prevents the chance that an obstacle that should be a smaller rock turns into a large one by pure luck. All spawner variants take into account weight when selecting an obstacle to spawn, allowing us as developers to fine tune the experience for players.

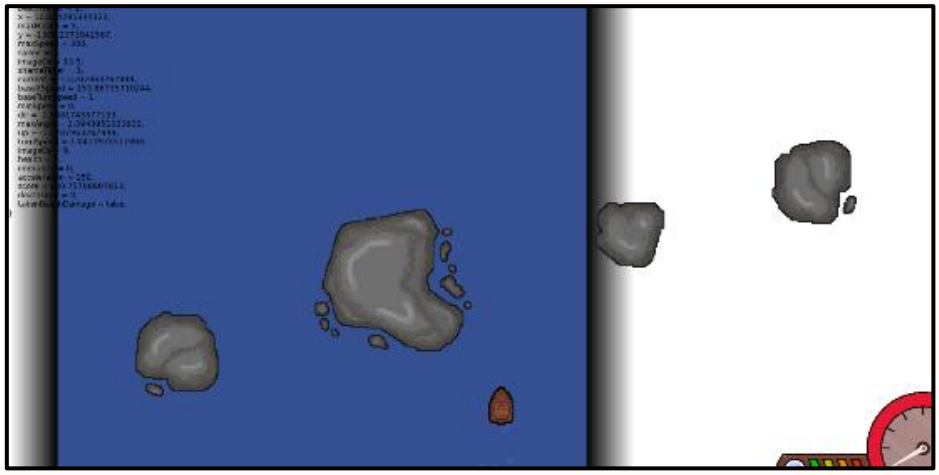
Theoretically, in any stage with random generation, a terrific stroke of bad luck can still create an 'impossible' river like the one shown above. One of Patrick's jobs was to balance until the chance of this issue occurring was negligible.

Other things

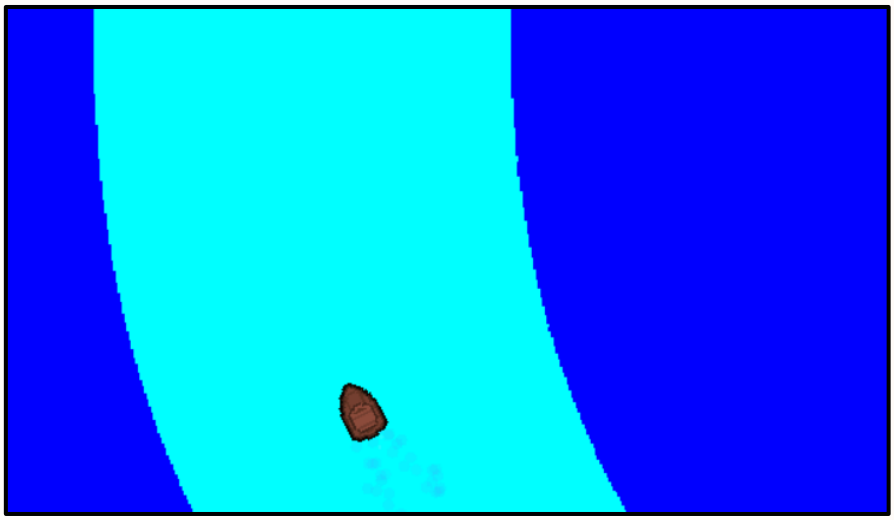
I gave obstacles the ability to run functions when they spawn and update, allowing for Patrick to create obstacles such as 'rock chains.' When a leader rock spawns, it spawns other rocks to create a chain, spanning across the river.

One other use case is moving obstacles, that we have taken advantage of within the Autumn Grove. Each obstacle has a physics body that allows it to collide with other obstacles and players. If an obstacle that spawned in is colliding with another, it will despawn unless explicitly told not to, so that we don't have ugly overlapping rocks.

Finally, I created a layering system to allow us to decide which obstacles would draw on top. This was useful for trees and the energy orbs fired by tesla coils in Stage 3, as it looked odd when these sprites appeared under the player..



Rock chains in early development. Wow, that prototype river is ugly.



The 'dev river' where backgrounds and zones were first implemented. I take it back... I think this river is uglier than the last one.

Other Technical Aspects

Player

Movement

Buoyant Voyage has two player controllers, one for when players are playing and the other for the end of the river. The main controller takes players inputs and converts them into movement. Similarly, the automatic controller converts information about the players position into movement. The auto controller simply attempts to seer the player to the centre of the river. The player's rotation has been capped, preventing any reversing and forcing the player to move forward at all times. The player gets additional movement from the river's current, determined by the current zone and general river direction. The current is not and does not affect the players rotation or visual rotation. Another factor acting on the players movement is a base x speed. This is to help the player feel in more control at higher currents. This gets added to the players horizontal movement and is calculated with cos(10*sqrt(current)).

Collision

The player has a circle collider, allowing it to interact with other physics objects in the world. Because the riverbank does not have a collider, we use a 'beach timer' to slow down the player when they are on land, preventing the player from gliding over it.

Game Over

At the end of each run, the player gains a score based on how well they played.
The score calculated by a variety of factors. The largest of these is the players displacement at the end of the run, in pixels, then divided by 10. In the Frosted Channel, the highest possible base score is 10,000. The player's run time (in seconds) it then deducted from this score, encouraging players to go faster and play more riskily. Finally, a bonus of 1,000 times the players health gets added to the score. This bonus score can only be added when the player beats a level as dying, restarting, or exiting sets the player health to 0, preventing any form of funny business. The scoring system increases the replayability as earlier described in the ***Game Design: Overview*** section.

```
function PlayerBoat:UpdateScore()
    self.score = math.abs(self.y/10) - self.runTime + math.max(self.health, 0)*1000
    UpdateHighScore(self.score)
end
```

The simple function to calculate the player's score.

Camera

The camera in *Buoyant Voyage* allows players to seamlessly resize their game window to any desired size. While Love2D does have functionality for resizing the window, there is no automatic scaling that comes with it. *Buoyant Voyage* is designed around a 1920x1080 screen, where a screen scale is calculated though the following function:

```
screenScale = math.min(love.graphics.getWidth()/1920, love.graphics.getHeight()/1080)
```

This scaling factor ensures that our desired resolution will always be visible. This alone does not provide the effect we want as the Love2Ds default camera origin is top left aligned, simply scaling up our images by the screen scale would leave the excess on the bottom or left regions of the game window instead of evenly around the edges. To combat this, an additional screen offset x and y (sox/soy) is calculated.

```
local sox = ((love.graphics.getWidth()/screenScale) - 1920) /2
local soy = ((love.graphics.getHeight()/screenScale) - 1080) /2
```

All graphics are transformed by this the screen scale and offset variables before anything can be drawn to the screen before being drawn.

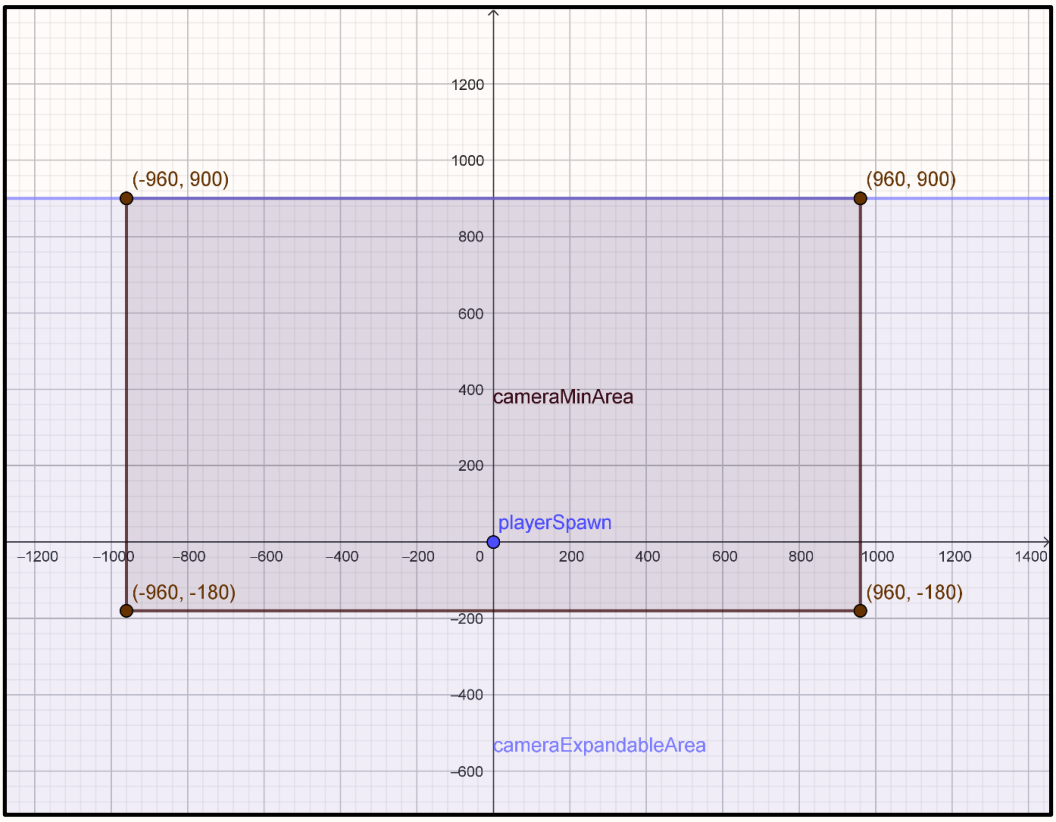


Diagram of default camera position (above) with expandable area. Please note that y axis values are flipped within Love2D. The player may spawn at any x position; however, it is set to 0 when first created.

This works well for the level select menu, however while in a level, the camera has further offsets based on the players y position, and the river's centre. The camera is moved left half a screen (960px) so the river, whose centre x is 0, remains in the middle of the screen.

The y position of the camera is locked to the players position, plus an extra 900 pixels to allow players to see ahead. Furthermore, the soy value is ignored, unless drawing a user interface, so every player can see the same distance in front of themselves no matter their screens resolution.

There is one more camera scaling option withing *Buoyant Voyage*, used for the title screen river. This camera scales so the width is always1920, in order to keep the rivers x location the same, relative to the rest of the screen.

Dynamic Loading

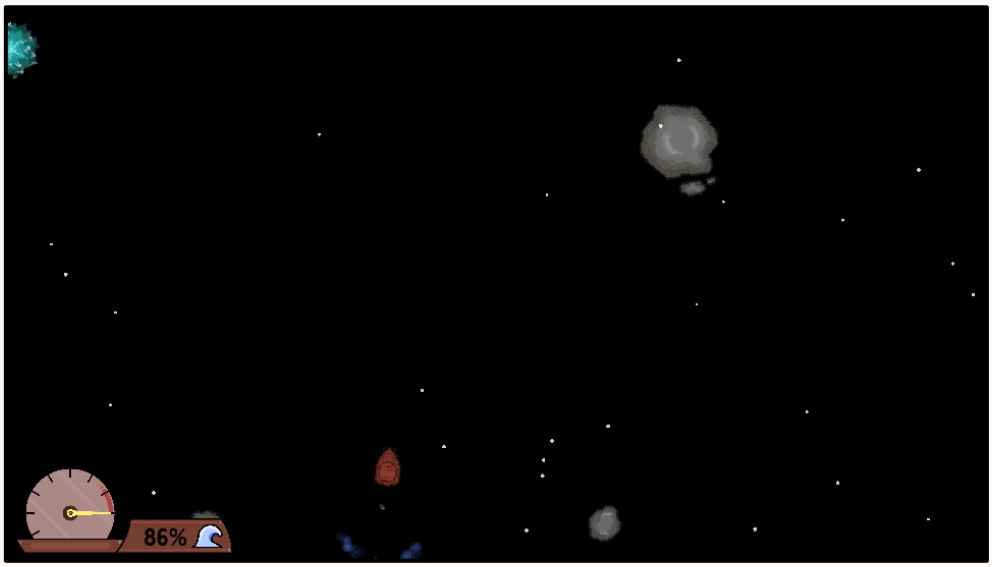
Love2D does not automatically load assets for us, and if we were to load *all* of our assets at once inside the "love.load" function, Love2D would freeze up – as love.load dedicates all processing power to loading whilst no longer processing other events. A Dynamic Loading (let's call 'DL') library was created to load assets while still allowing Love2D to process events. A modified version of the default Love2D 11.0 love.run function (https://love2d.org/wiki/love.run) allows events to be processed. Another bonus is that it allows us to draw and animate a loading screen.

DL first loads the "load list," a list of all items to load, then systematically loads each item, into a similar place similar to its file location into our asset able. For example, the players image, stored at "image/player/default.png" can be accessed through "assets.image.player.default." All the while, managing events without freezing up and continuously responding to player's inputs like quitting, fullscreen, and other window resizes.

To help save resources, DL unloads the previous gamestates files before loading the new ones. In cases where a file must be permanently loaded, a function can be added to DL's load list.

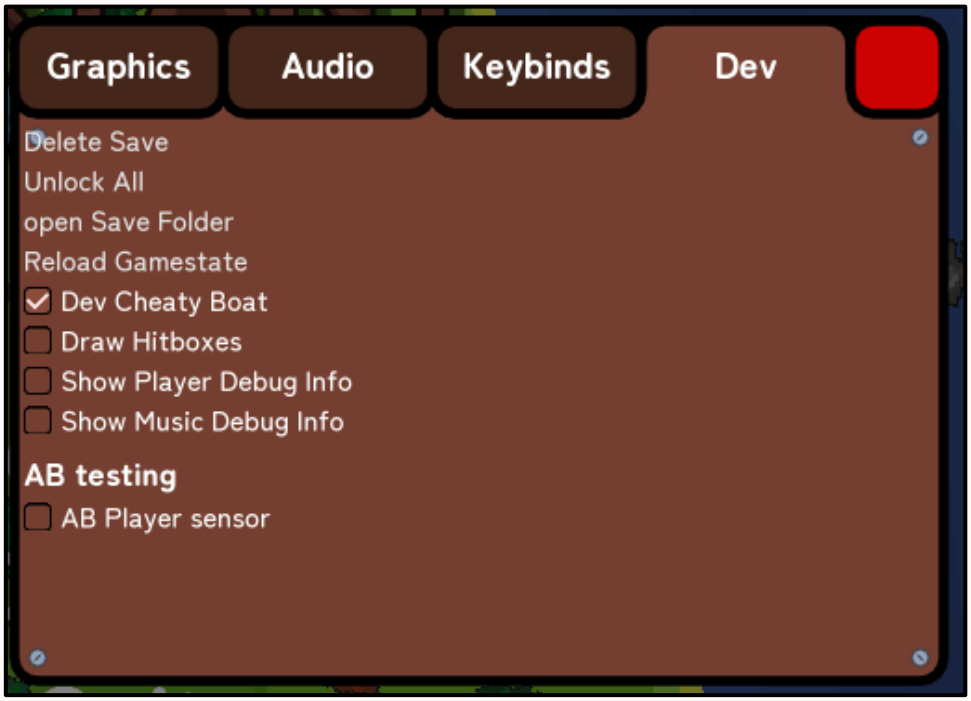
Other Tweaks & Quality of Life

Developer Cheats



"'Dev Cheaty Boat', my beloved"

Testing and debugging a game with no scene editor is a tall order, so to make our lives easier a "Dev" menu was created. Tacked onto the settings menu, this provides abilities similar to Gizmos in Unity.



Almost nothing in *Buoyant Voyage* is "required," meaning any script that is edited can be reloaded by reloading the gamestate. This one feature, allowed us to quickly iterate code without having to re-load the entire game, being especially helpful to me when creating the custom "transition zones" to smoothly blend sub-biomes together.

Viewing hitboxes is especially useful for debugging colliders, esoterically on objects that don't have any visual assets like the no spawn areas. Seeing exactly their size and location helped with adjustments without having to 'feel' around. A shortcoming of how "Draw Hitboxes" was written prevented them from being useful when debugging the "Ghost objects," mentioned later. This occurred as each object in the object table draws its own hitbox instead of the world drawing all of them.

And then there's the debug info. These print an entire table, or in the case above, the player or music table, onto one the side of the screen. The debug info works like the inspector panel in many game engines, displaying all variable in real time. The "AB testing" section, gave us the opportunity to play with ideas, allowing testers to turn features on and off at will without needing two different builds.

Library

Our 'proprietary' template library, which we prepared before entering the challenge, helped speed up development time. This library includes a custom game state manager, somewhat similar in functionality to unity's 'scenes.' Our small tweening library is the basis for most aminations within *Buoyant Voyage*. easings.net (https://easings.net/) was used to help visualise and find logic for easings, which we used to add life to some UI features. Our game saves directly to lua files that can be loaded directly into Love2D, thanks to the saving and loading library. This library can convert any table into a string, with the exception of functions, and nonnative lua types. Although, it's not the only remaining library, the final one to be mentioned here is the UI/Button system within *Buoyant Voyage.* This was re-written thought development to support our growing needs including features such as sliders, image support, colour tweening, and update functions. The only buttons not using our library are the level select buttons due to their circular hitbox shape and animations that were not available at the time of implementation. The rest of the library can be found on the GitHub reop within the "boatGame/ templateLib" folder.

Delta Time Cap

The delta time, or the time taken for the last frame to process, used in framerate independent calculations has been programmatically capped to one tenth of a second. This does not affect the processing rate of *Buoyant Voyage*, instead giving players with slower computers a maximum amount of time that can pass in game each frame. This simple addition prevents players seemingly teleporting into obstacles or the riverbank during any possible lag spikes that may occur.

TESTING & REFLECTING

Testing

Most of the testing we ran internally between one another. When Daniel programmed a technical system, Patrick would mess with it to see if any bugs or crashes occurred. When Patrick created gameplay and assets, he would run it across Daniel to get a second opinion. However, at various points we ran testing with friends and family to get outside opinions on gameplay. We made sure to attribute testers on our credits screen.

'Playtesters' section in the credits menu.

Play testers were asked feedback about how the player felt to control and whether the difficulty of the game was sufficient. They were also keen to provide extra comments and feedback on certain systems they did or didn't like. Below are some examples of playtesting feedback which we actioned upon. There's one example where we implemented a change, and another where we actually ignored the playtester feedback for other reasons.

Automatically lowering the throttle when W is released

When playtesting the minimum viable product back in February, we observed a common habit that many players were making:

1. Players would push the throttle up to maximum, and release the W key.
2. They would then subsequently forget about the throttle.
3. Players reach a more difficult zone, and quickly perish as they're at maximum throttle without even realising it.
4. Playtester then proceeds to complain profusely. ugh.....

We found that forcing players to hold the acceleration key if they want to stay at top speed (and dropping it back to 70% throttle if [*accelerate*] is released) made them subconsciously more aware of the throttle, and nobody has forgot its existence since.

"Add a health pickup"

A common request from many testers was that the game needed some sort of 'repair system' or 'health pickup' so they could regenerate health. This would most likely take the form of a box which moves down the river. We did get pretty far into implementing it, adding a system for pickups and new functions so that obstacles could run custom functions on collisions with the player. We stopped halfway through because there were a few things we didn't like about it.

- It would reduce the gameplay impact of taking damage.
- It's counter-intuitive that, in a game about avoiding everything in the river, players have to collide into this one for benefit.
- If it's flowing down the river, how does it interact with rocks?

There were just too many issues and so, we went against tester feedback and scrapped the idea. Instead, what we took was that testers had perhaps found the game's learning curve too steep, and in some parts unfair. We balanced by reducing current (so testers had more time to react to upcoming obstacles) and lowering the overall learning curve.

Final testing phase

On our final build, we played with a wide range of testers and made sure to run the game on as many different computers as possible just to see if anything would break. Unfortunately, because this testing phase was so close to the submission date, we weren't able to action on some of the feedback (such as "the current pushing the player left and right is unintuitive,") but may do so in a post-challenge release of the game.

Fixing

Over the past few months, we have become a custom to patching small bugs and crashes from simple syntax errors all the way to obscure bugs within Love2D itself. Each bug presented its own challenges as after all,

"It's always the last thing you try"

Throughout the debugging process a lot was learnt, like the fact that we have no idea how the key binds are being updated while in a river, even though the system is entirely coded by Daniel from scratch. Our play testers were relentless when it came to finding bugs and general issues/improvements that could be made. Some of the most egregious bugs and issues have been placed below, along with how we went about fixing and overcoming the challenges they each provided.

Water Trees & Other Tree-related issues

Tree generation was causing many issues within our game. They weren't spawning at the start of each zone, spawning on top of mountains (still an issue), and occasionally inside the river. In *Buoyant Voyage*, trees spawn when the background colour at their position are within an accepted list. If the red channel at that point matches their accepted colour list, then the tree can spawn. The trees were using the stored zone, calculated each frame based on the players position instead of the trees position. This caused trees to check the colour based on the wrong zone, frequently returning different a colour than it should. This bug was patched by actually using the trees position.

```
-       if assets.code.river.zone[zones.zone or zones[1].zone].GetColourAt(x,y)[1] == autumnTreeAcceptedColours[i] then
+       if assets.code.river.zone[riverGenerator:GetZone(y).zone].GetColourAt(x,y)[1] == autumnTreeAcceptedColours[i] then
```

```
-       until river:GetLastPoints()[1][1].y < riverBorders.up
+       until river:GetLastPoints()[1][1].y < riverBorders.up - 1500
```

This didn't stop trees from spawning in the river, so there had to be something else at play. Eventually, the root cause was discovered, when loading the riverGeneraor would wait for the length of the river to extend above the top of the screen before generating trees and other obstacles. If the last segment was short enough and its x position deviated from the starting position enough, then trees could generate slightly off screen where there was no river. When checking the background colour, the river would not exist at that height and instead use the 'guess' option intended for the player, selecting the first river points as the river's location. This in turn made the background believe it should be ground and not water, allowing the tree to spawn. This bug was fixed by changing the required height of the river to 1500 pixels above what is necessary, providing enough clearance for obstacles to generate.

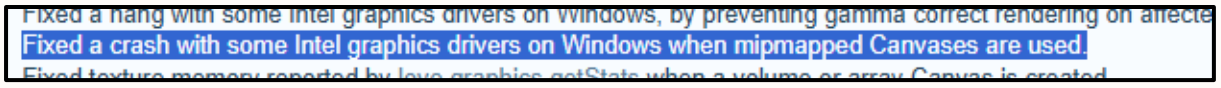
Ghost objects

"Ghost Objects" are physics bodies that don't get destroyed properly before the table they are contained within are reset or set to nil. In such cases, the physics engine still believes they exist and edge cases within the river loading/unloading made getting rid of them painful. These pesky objects are almost undetectable unless players happen to collide with one while playing increasing the difficulty of eradicating them. Since we can't have the player colliding with an object that was once there but now invisible, keeping them was not an option.

Implemented solutions include proper user data management, only creating bodies where necessary, and proper unload procedure when changing or reloading gamestates. The problem only grew as we added layers to objects as removing an incorrect object was made possible through one mistyped Boolean. Removing the wrong object leaves behind a ghost object while appearing to remove the correct one. After much combing through the code, I (Daniel) believe we no longer have to call the Ghostbusters.

'Big Crash'

Upon making our first test build for windows, *Buoyant Voyage* was occasionally crashing when loading into any river that was not the first. This crash seemingly traced back to the background generator however gave no error message. When debugging in an uncompiled version, we were unable to recreate the issue, no matter how hard we tried. And so, we turned to the internet, still finding nothing. As a last resort, we had the thought to check the Love2D changelog and issues page, due to the older version of Love2D we were using. Turns out, there was an issue with canvases, and some windows drivers when compiled with the version of Love2D we were using.



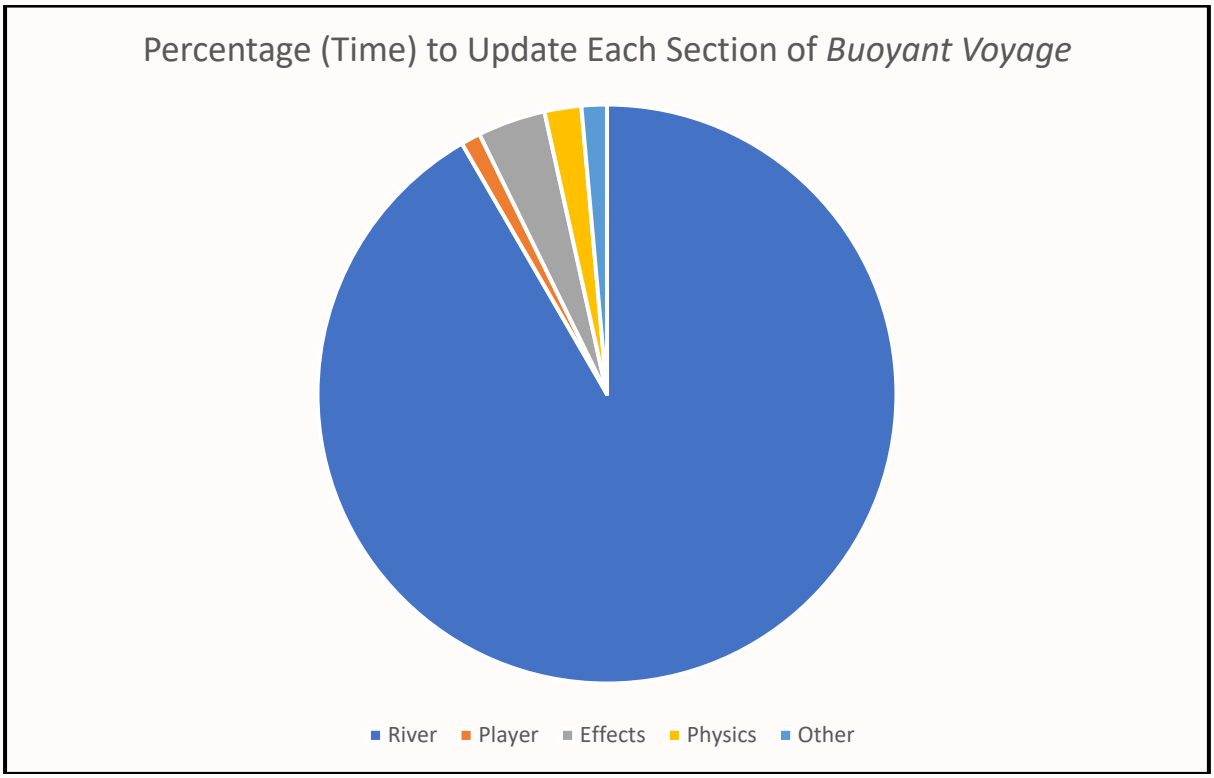
As a result, we updated to love 11.5, the latest version of Love2D. Eventually, when implementing multi-threading, we moved away from canvases entirely, opting to use Love2D's ImageData object, that is converted directly into an image. This change also acts as an extra layer of redundancy to the initial error.

Lag

After a few months of intense feature additions (around May 2025), *Buoyant Voyage* started to slow down. Love2D had no built-in method for figuring out what functions were taking up time, so Daniel endeavoured to create a small library of functions to help with our performance issues. The functions he created are as follows:

- **log.start** – Starts the logging process, resets current logs and sets the time.
- **log.jump** – sets the time to the current time without logging the section. Useful for skipping areas of code that you don't want to time.
- **log.point** – marks a point in time and how long it has been since the last point, starting point, or most recent jump.
- **log.stop** – sorts the logged points from most time taken to least then prints to the console.

From a first log, we found that the update function was the caused most of the lag, and further logging provided the results below.



Graph of percentage time taken to update Buoyant Voyage's initial playtest build.

The river was taking up most of the processing time, in fact, about 92% of the measured time, so something had to change. There were two options that we thought of; multi-threading and shaders, both of which are supported to some extent in Love2D. The obvious choice was to create a shader that looked at the river points and generated a background based on that. This would have been much faster however, neither of us knew GLSL and Love2D uses a slightly modified version of GLSL, meaning not only would we have to learn another language but a more obscure version of it. Furthermore, the prospect of constantly sending river point to a shader was not something we was fond of.

This left us with the multi-threading option and although neither of us had any experience, the fact that we both could use a language we already knew made it much easier to learn. Instead of sending river points to the generator thread, we opted to generate points within the thread and send them to the main thread, saving more resources for other tasks. Although this approach is vastly slower than a shader, the accessibility made it worth it, giving us more time to focus on polishing.

[Not Responding]

While testing *Buoyant Voyage,* one of our testers came across an excruciatingly exotic bug, where the game would cease to respond upon returning to the title screen. we were unable to re-create the bug constantly, or even almost at all, during our testing when repeating what the tester did. Eventually, Patrick was able to re-create it though the following steps:

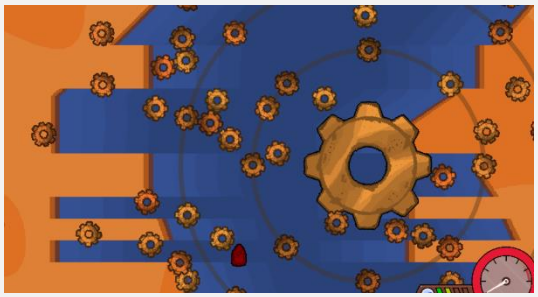
1. Enter the level select
2. Return to the title screen
3. Watch the credits from the title screen
4. Enter the level select again
5. Play Stage 3: The Dam, beating for the "first" time
6. The credits play automatically upon beating the stage (watch until the end)
7. Enter the level select again
8. Return to the title screen
9. The game freezes with no error message.

Even with all those steps, it was not consistent. Daniel's initial guess was that the issue lay on the generator thread, because the crash was a freeze rather than a runtime error. Adding debug printing into the thread, in an attempt to catch the bug, did not provide the results he wanted in fact the thread was as smooth as ever. After scouring the dynamic loading, river generator, river, and title screen loading files, we still had no leads. Nothing was changing and the bug was still as elusive as ever, creating a horrible situation to be in where code tweaks had to be tested over and over in the off chance that the unknown issue had been resolved. As a last resort, Patrick suggested we look at the code form start to end systematically. Eventually Daniel came across the problem and the unsatisfyingly simple fix.

A single line- and it all goes to show that the hardest bugs to spot may have the simplest solutions. The problem had nothing to do with the credit sequence like we originally thought but simply staying on the title screen for an extended period of time, leaving, then returning to it. The rivers y value was not resetting, causing the river generator to go into overdrive generating all the way from y = 0 to a very large y value. The river could not continue to update until it got points above its own y, causing *Buoyant Voyage* to freeze up until Windows gave up and quit the program for us.

Other Prominent Bugs and Issues

We've included a table of other worthwhile bugs here. For the sake of conciseness, we won't go into too much detail about them.

Bug	Fix
Various crashes related to performing actions while loading (Fullscreen, clicking, tabbing off, keypresses).	Added checks to ensure all accessed tables existed.
Max deviation & River Offset X parameters not working as intended.	Re-did the math, ensuring all scaling factors effected calculations.
Player being unable to spawn in the canter of the river, instead opting to be wherever it pleased.	Fixed update order, player loads after initial river segment generated.
Player could obtain a negative score when colliding with the riverbank at 1 health.	Clamped players health to 0 or above
Two rivers at once (image below) 	Add a thread channel to ensure the generator has closed before creating another one.
The player could die after completing the level, displaying game over instead of river complete.	Player can no longer take damage after passing the river length
Transition zone (boulderValley-coniferousMountainside) replayed the entire storm quickly.	Added support for zones that don't update title when zone changes. boulderValley-coniferousMountainside is in this list.

Project Reflection

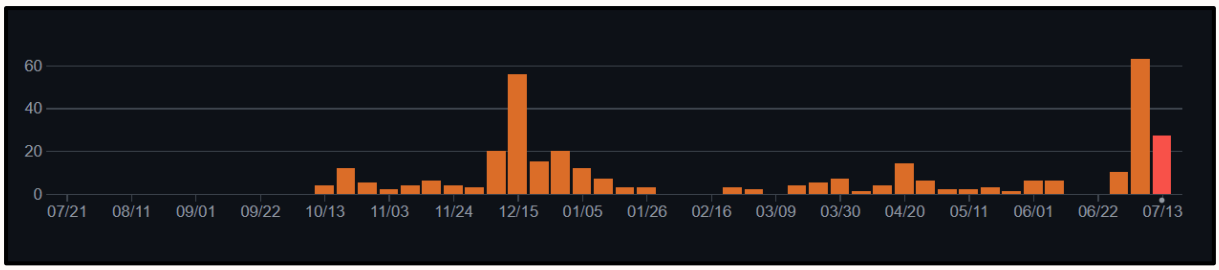
Written by Patrick, as we thought this section would do better in personal voice.

Buoyant Voyage is one of the two game design projects we've seen through from beginning to end. Most other projects we've made are abandoned somewhere along the process. The only other 'finished' project we've ever made was *Buoyant Voyage's* predecessor, *Boat Game With a Name* (which is almost a bit poetic). Last year's submission to the STEM VGC was terribly unfinished in comparison.

We've wanted to make this game since 2022, and while whatever ideas we had back then are long forgotten, we came up with a new vision and firmly stuck with that one image throughout the entire project. That's a first for Daniel and I, and so I'm proud. For once, the game really is at a point where I'd call it complete.

That being said, there's always plenty more things we'd like to add, even if they aren't core to gameplay. We'd both like to add a system for unlocking different boats, whether that be cosmetic only or have gameplay changes. I'd like to make a 'journal' menu with some of storytelling on *Buoyant Voyage's* world, and Daniel won't sleep until every single tiny bug and crash is hunted down (trust me, we found the most exotic, un-replicable crash on the final day of playtesting and it's haunted us since.)

Since we're year 12s, there won't be a 'next' Stem VGC for us, but I'm sure this isn't the last project we'll make. We've learnt next time to leave time after the "final" playtests for all the new feedback we thought we wouldn't get. Our time management skills are always a work in development, and this time was no exception. Still, I think we managed to set enough just enough time to polish the game and write a (very long) game design document – something we didn't do as well last year.

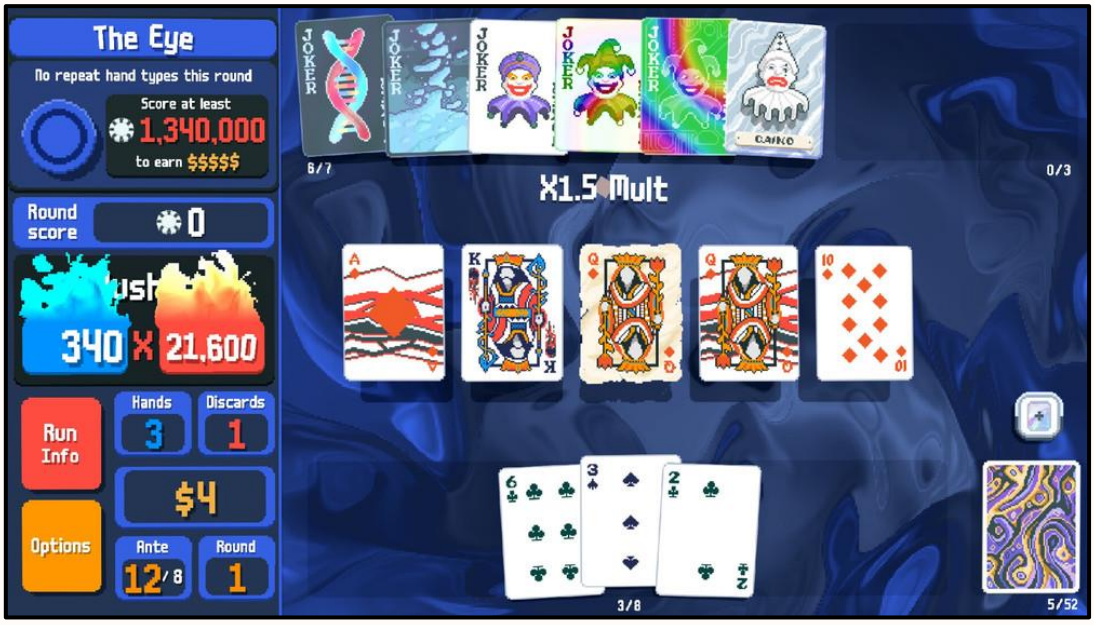


The commit history graph of Buoyant Voyage. Funnily enough, the all-time peak is just two weeks before the submission date (which tells you a lot about how we work, haha). A total of ~500 commits were made across the 8 months we worked on it.

My main disappointment is that *Buoyant Voyage* isn't very aesthetically impressive. This is a natural side-effect of us using Love2D; we don't have any lighting, 3D or a particle system as good as Unity. On the technical side, I think Daniel deserves to be proud of his procedural generation system... but, it's really unoptimized, and if you take a look at the code, there's a lot of spaghetti.

Moving forward in our game development journey, I think there's two options for us furthering our game development skills. The first is we overcome our fear of good things and pick up a 'real' engine like Unity; this would speed up development and give us more aesthetically impressive outcomes.

The other, more difficult option is we commit to Love2D and learn how to properly use GLSL shaders. It's absolutely possible to create a good-looking game in Love2D. *Balatro* was made in Love2D (with heavy use of custom shader plugins), and the graphics in that game are phenomenal. Oh, and it almost won game of the year.



Balatro was also written in Love2D. So, we have no excuse really.

Either way, *Buoyant Voyage* is the most technically impressive project Daniel has written and the most artistically cohesive game I've made. We're both proud of it.

"I would say it's our Magnum Opus, but it's really starting to sound like I'm trying to sell our game, so I'll stop there."

"How about you say, it was truly a "journey" to make this game?"

"No."

REFERENCE LIST

Help / Documentation / Tutorials

https://easings.net/
https://love2d.org/wiki/Main_Page
https://www.lua.org/manual/5.1/

Programs Used

https://code.visualstudio.com/
https://github.com/
https://love2d.org/
https://lua.org/
https://sfxr.me/
https://trello.com/
https://www.apple.com/au/logic-pro/
https://www.aseprite.org/
https://www.audacityteam.org/
https://www.geogebra.org/

Inspiration

https://hopoogames.com/risk-of-rain-2/
https://scratch.mit.edu/projects/545055939/
https://www.celestegame.com/
https://www.roblox.com/games/537413528/Build-A-Boat-For-Treasure
https://www.stardewvalley.net/

Other

https://box2d.org/
https://www.google.com/maps
https://www.playbalatro.com/
https://www.stemgames.org.au/shared/files/stem-classification-guidelines.pdf

"Can I add the Library of Babel as a source?"

"No Daniel, please, don't."

APPENDICES

Appendix A: Obstacles

A.1 Trees

Snowy Tree
A decorative tree obstacle; this one has snow on it. Spawns throughout most of Stage 1. To make sure they don't spawn on the river, or riverbank, all trees run the terrain generation code to check that the colour underneath them is

Highlands Tree
Functionally identical to a snowy tree. Dark green in colour, without any patches of snow. Spawns in the "Wooded Hills" sub-biome at the end of Stage 1.

Hills Tree
The generic tree. For some colour variety, one in four of these trees are a slight orange like the one pictured on the left. Spawns in the intermissions at the beginning and end of Stage 2, as well as throughout Stage 3.

Autumn Tree
Again, functionally identical to other trees, but in a range of beautiful autumn colours. Spawns throughout the Autumn section of Stage 2.

A.2 Rocks

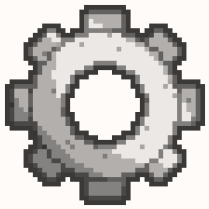
Rock
A generic, small rock. *Buoyant Voyage's* simplest obstacle, fitted with a circular hitbox. To add variety, there are 11 alternate rock images which each rock randomly picks at spawn. All variants are identical in functionality, with the same hitbox size. Appears in many places throughout all three stages. It's hitbox has a radius of 30 pixels.

Big Rock
A larger version of the rock, with hitbox radius 75. Functionally identical, with 5 image variants. Also appears intermittently throughout the entire game.

Huge Rock
An even larger version of the rock which spawns only in the 'Boulder Valley' and 'Storm Valley' sub-biomes within Stage 1. This rock's hitbox has a radius of 150. Has three image variants.

'Leader' Rock (Spawns a 'rock chain')
Spawns in 'Boulder Valley' within Stage 1 only. When created, this rock spawns a group of other rocks, in a chain that spans across the river, forcing the player to weave between the gaps.

A.3 Cogs

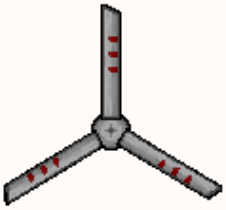


Moving Cog
A cog which slowly oscillates left and right between the borders of the river, on a sine wave. Spins as it moves and has two image variants. Appears in the 'Clockwork Ruins' and 'Clockwork's Core' sub-biomes in Stage 2.

Huge Cog
A massive cog which spins in place. Spawns rings of Subordinate cogs which orbit around it, creating a menacing, spinning mess that the player must manoeuvre through (it also looks *very* cool). Spawns in the 'Clockwork's Core' sub-biome in Stage 2 only.

Subordinate Cog
When spawned on its own: Functions identically to a rock. One in ten slowly rotate to add some variety. Has two image variants.
When spawned by a huge cog: Slowly rotates around the huge cog in an 'orbit' with many other subordinate cogs.

A.4 Spinners



Spinner
Spawns throughout Stage 3, rotates in place. Has three blades and a complicated hitbox made of multiple polygons. The small red stripes were added to make it look more menacing. Spins faster when specifically in 'The Inlet' sub-biome of Stage 3.

'Leader' Spinner (Spawns a spinner row)
On spawn, creates a straight row of spinners which spans the river. For variety, rows of spinners vary in direction and synchronisation; some rows spawn with every other spinner spinning the opposite direction. Spawns in 'Upper Dam' and 'Electrical Complex' within Stage 3.

A.5 Tesla Turret

Tesla Turret
An incredibly menacing rock which shoots energy orbs in a spiral pattern. Utilises a custom draw function so that its 'coils' can rotate, giving the turret an animated appearance. Spawns in the 'Electrical Complex' sub-biome of Stage 3. Plays a loud mechanical sound.

Energy Orb
Spawned by the Tesla turret, moves in a straight line for ten seconds before fading out. Renders on-top of other obstacles, and uses a 'sensor'-type hitbox so that the player takes damage without physically colliding with the orb.

A.6 Other

Hailstone
We felt the first stage needed at least one moving obstacle, so we added the hailstone. A small ball of ice (radius 15 pixels) which flies horizontally across the screen within the 'Storm Valley' sub-biome of stage one. Spawns a trail of harmless particles behind it.

Beach Junk
Believe it or not, this is functionally identical to a tree. Spawns in the 'River Mouth' sub-biome in Stage 3. It's a statement on humanity's widespread pollution due to by the rise of rampant consumerism... uhh... no, it's just there because the beach looked empty otherwise.

(no image)

'Obstacle' (template)
A template obstacle so we didn't have to rewrite all the boilerplate code every time we made a new obstacle. Doesn't show up anywhere in game, and has no sprite.

(no image)

No-Spawn Rectangle
A large rectangular hitbox which blocks other obstacles from spawning within its confines. Does not interact with the player. These are spawned by Moving Cogs to ensure their entire movement path is kept clear of other obstacles. Has no sprite.

(no image)

No-Spawn Circle
Functionally similar to a No-Spawn Rectangle, but circular. Spawned by Huge Cogs and Tesla Turrets so that the nearby area near these dangerous obstacles are clear, as to give the player a fleeting chance of survival. Has no sprite.

Appendix B: Game Systems Tree

Daniel created this tree as an informal way of keeping track of all of *Buoyant Voyage's* technical systems. Over time, this tree has grown to be quite large, so we thought we'd put it in the appendix.

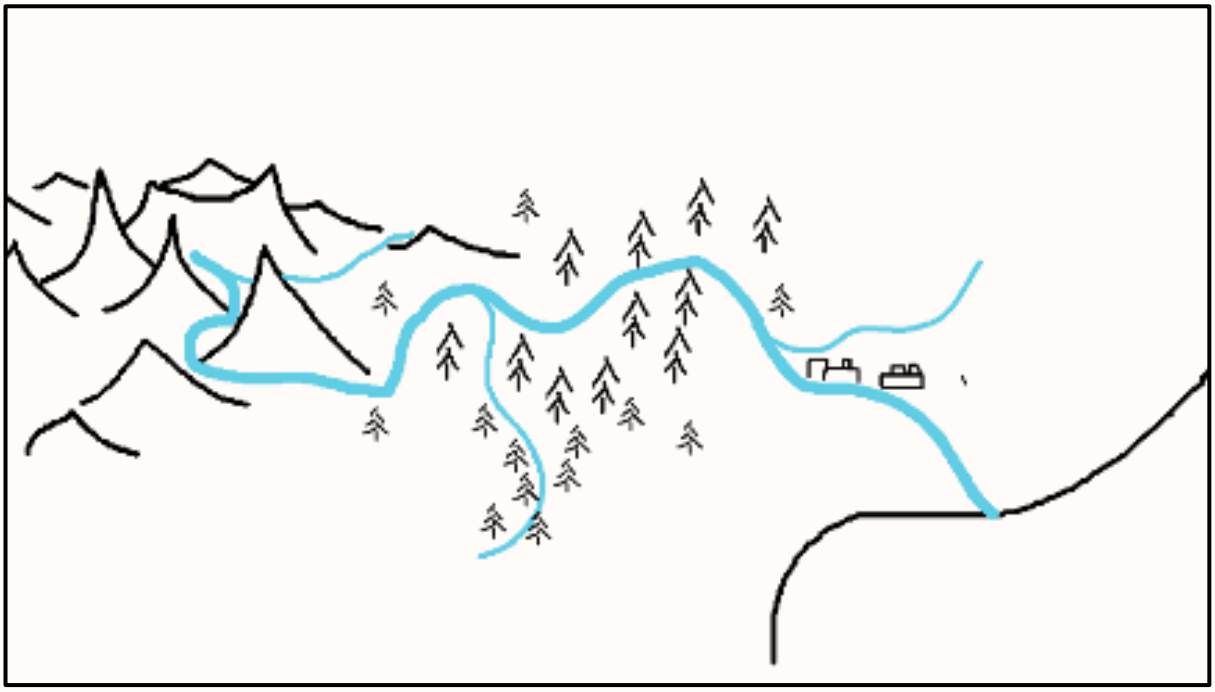
```
Buoyant Voyage
├ Splash
├ Credits
├ Template Library
│ ├ Audio Manager
│ ├ Dante (Saving)
│ ├ Font System
│ ├ GraetUI
│ ├ Particle
│ ├ Quindoc (Math functions)
│ ├ Gamestate Manager
│ └ Tweens
├ River
│ ├ Zones
│ │ └ Files
│ │   ├ Background generation
│ │   ├ Obstacles
│ │   └ Path generation
│ ├ riverData
│ │ └ Files
│ │   ├ ambiance
│ │   ├ music
│ │   ├ obstacle
│ │   └ zone
│ ├ Generation
│ │ ├ Threads
│ │ │ ├ Zones
│ │ │ │ ├ loading
│ │ │ │ └ Picking in infinite river
│ │ │ ├ Path
│ │ │ │ ├ 10k buffer
│ │ │ │ ├ Points for curve
│ │ │ │ ├ Curve
│ │ │ │ └ Warping curve based on angle
│ │ │ └ Background
│ │ │   ├ Buffer
│ │ │   └ Default noise transition
│ │ └ Obstacle
│ │   ├ Two layers
│ │   ├ Spawning
│ │   │ ├ Spawner types
│ │   │ ├ Random
│ │   │ ├ Timer
│ │   │ └ Difficulty independent
│ │   ├ Obstacle (base)
│ │   ├ Trees
│ │   │ ├ Fix to ensure trees didn't spawn on water
│ │   │ ├ Fix when trees were spawning on 'mountains'
```

- Fix for trees not spawning at start of zone
- Rock Chains
- Cogs / moving obstacles
- No Spawn Area
- Blades (Multi Fixture)
- Despawns off screen
- Other
  - Current
  - Despawns behind player
- Player
  - Health
    - Beaching
    - Take Damage
      - Camera Shake
    - Game Over Logic
  - Movement
    - Input manager
      - In game
      - Game end of level (game takes control of player)
    - Clamp angle
    - Visual direction
    - Current
  - HUD
    - Speed-o-meter
      - Shake
      - Crack
    - Health Animation
    - Wave icon
- Camera
  - effective 16x9 lock (1920x1080) in river
    - Center on X axis
    - Lock to top of screen for Y
  - Resize Pauses Game (re-generate the background)
    - Not necessary on title screen as zoom in
  - Shake
- Title screen
  - Buttons + animations
  - Title screen river
- Menus
  - Settings Menu
    - Keybinds
    - Audio
    - Graphics
  - Game Over
  - Pause
  - Level info
- Save and Load
  - Settings
  - High Score
  - Separate score for storm levels
  - Level Unlocks

Appendix C: Some concept art and design sketches

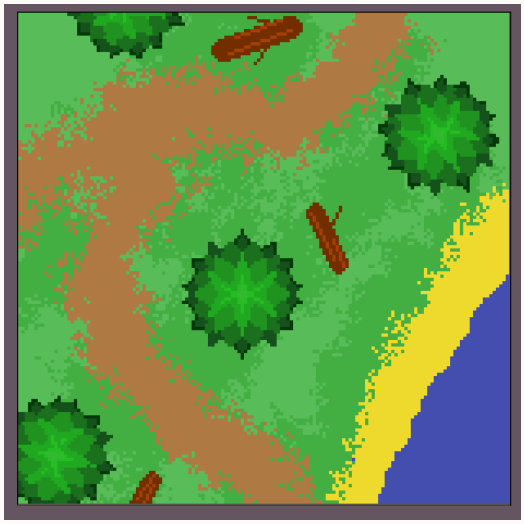
The concepts, mock-ups and sketches below are shown in no particular order.

Concept art for the loading screen splash. Initially, this was going to follow a pixel art style like the river.

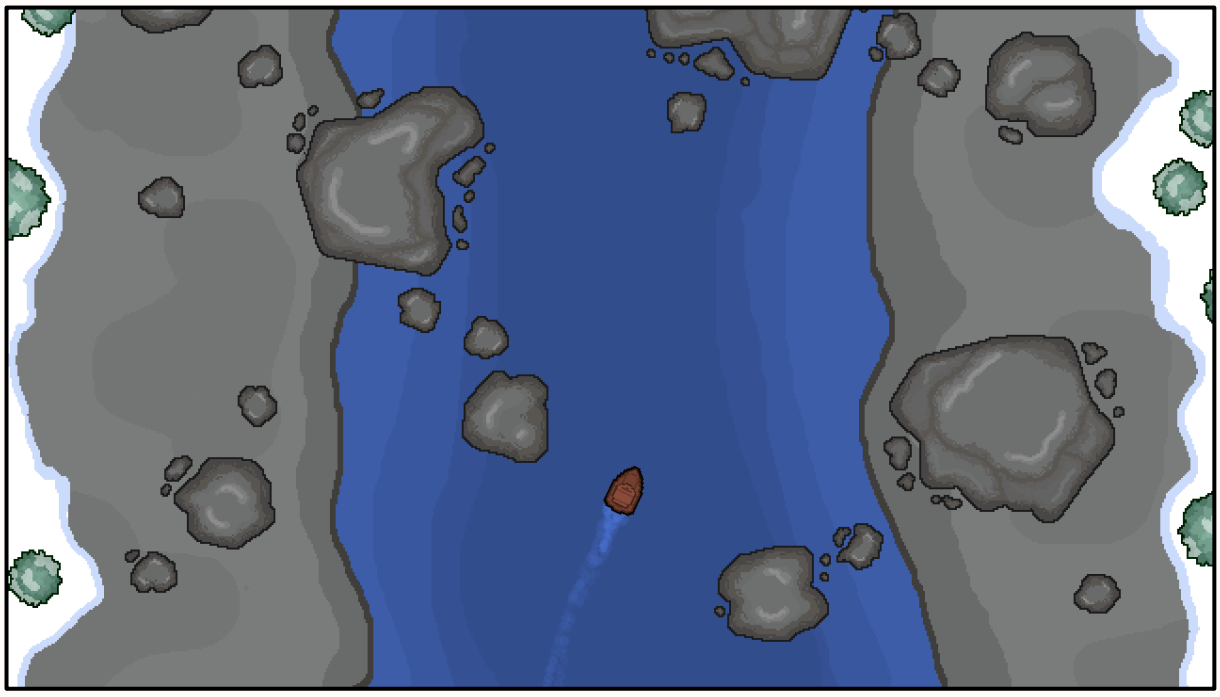


Digital sketch of world map's design. This was quickly thrown together to communicate ideas, so the quality isn't particularly high.

Title banner concept, used as a placeholder for initial playtesting.

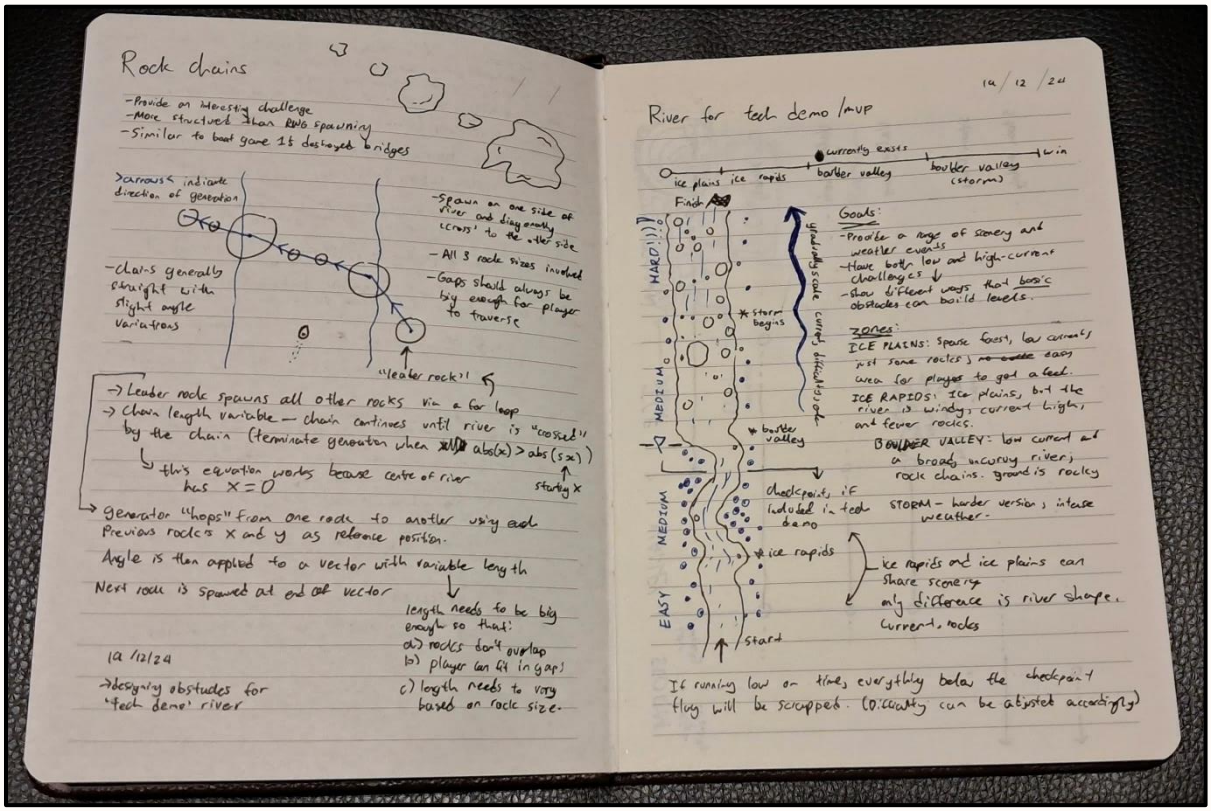


The first art style concept, made all the way back in November 2024. It's very noisy.



Boulder valley mock-up, used to showcase the newly made rock sprites.

Autumn Grove mock-up; was the concept for the game's artstyle



Two pages from Patrick's notebook back when he was first designing gameplay.
Left page: concept sketch / design notes for the rock chain obstacle.
Right page: concept and notes for the river included in the 'minimum viable product'.
This later became stage 1: great valley, though after a lot of changes.

Appendix D: Sound Attributions

Three sounds were downloaded from https://freesound.org for specific sounds we weren't able to create ourselves:

Diesel engine sound (A component of the final player engine noise)
Marine diesel engine by AugustSandberg
License: Creative Commons 0
https://freesound.org/s/264864/

'Thud' sound (A component of the player collision noise)
Wind On Door Short.wav by Benboncan
License: Attribution 4.0
https://freesound.org/s/264864/

Thunder sounds (Plays when lightning strikes in stormy sub-biomes)
Thunderstorm_Lightning_RX_No_Birds by johnnydekk
License: Creative Commons 0
https://freesound.org/s/686207/

A Closing Note

"Patrick, should we write a letter to the judges? The winning team last year did it"
"No, that's too pretentious. Also, I can't be bothered"

...No hate to Team Redshift who, if they're entering this year, will certainly do very well once again. Either way, thank you, judges, for playing our game and reading this GDD. We had a blast making *Buoyant Voyage* and putting this document together.

- *Jaraph*