Table of Contents

1. Executive Summary

In the academic field of Virtual Reality, with the current state of hardware-based bipedal locomotion, current tools are at a price point where only researchers benefit from acquiring them, but they are too expensive for most consumers to be interested in, this also causes less implementation with consumer software. Because of this, there is a focus for researchers to find and study alternative locomotion methods, or improve upon already existing ones, such as teleporting, the use of a virtual joystick, or even physically swinging one's arms in a walking manner. The problem with locomotion testing is that when a user deviates from an expected path, it almost makes the virtual environment data unusable without having to do an extensive amount of filtering; there is also the issue that there is not much documentation in other testing systems for this same purpose, or even one that covers all commercially available headsets. This project aims to create a platform-agnostic testing environment and toolset which can be parametrized for any locomotion systems through the creation of a pseudo-maze with section loading on the fly to give the illusion of choice, and thus removing erroneous data, while also making it easy to modify to import future systems or change already existing features; it will collect raw user data in realtime in which the researcher will then be able to conduct any kind of statistical analysis. We will create our systems using the Unity game engine, a favored amongst academics, the use of Meta Quest headsets for hardware, PlasticSCM as version control, and all while being done using Agile methodologies for the software development cycle.

2. Project Description

This project is meant to aid Virtual Reality (henceforth referred to as "VR") researchers by providing a standardized toolkit to make virtual, maze-like environments with reduced / eliminated user-experience variability. Throughout VR research—and frankly any form of research—it is common for researchers conducting user studies to desire standardized *yet random* conditions in their study. Examples of this include randomized survey questions, randomized sequences of patterns, or even the inclusion / exclusion of entire parts of the study. With our toolset, we specifically aim to provide a virtual environment that *feels* random to the user by being in a "maze", but in reality this environment is highly dynamic and does not succumb to the variability of a true maze. The Project Motivations section delves further into *why* normal, pregenerated mazes have such high variability in regards to user studies. This reduced variability allows

research implications to be more statistically significant, reinforcing conclusions and making the study more effective overall.

One side effect of how this maze-like environment will be generated is the inclusion of what is referred to as “impossible spaces”. These are spaces that are physically impossible, but by the use of clever tactics can *seem* to be plausible. While these spaces occur, again, as a side effect of how the maze will be generated, the implications are still very interesting to researchers and curious minds alike. Below is a figure that depicts an impossible space in the context of our toolkit:

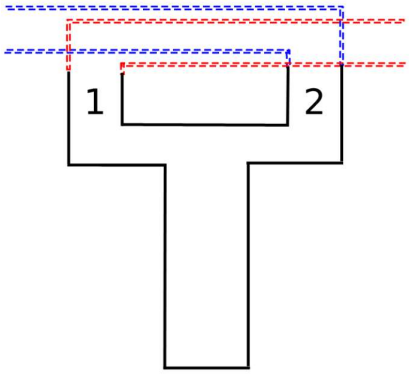


Fig 1. A visualization of an impossible space

Explanation: The solid, black outlined corridors of the maze are “static”. That is to say, this part of the maze does not change. The dotted, colored corridors are “dynamic”. That is to say, these walls do not permanently exist and rather are only physically active when in the user’s view. Clearly, both of these prospective walls existing at the same time would not be physically possible since they overlap the same place in space. However, the user can only make one choice at this junction - meaning both corridors *don’t have to exist at once*. If the user chooses corridor **1**, the red walls will become fixated into the maze and the user will proceed down the corridor, to the right side of the figure. Inversely, if the user chooses corridor **2**, the blue walls will be fixated and they proceed to the left. In the end, the *prospective* space is impossible, but the *experienced* space is sensical.

When developing this toolkit, one of our main foci is to provide all the “knobs and whistles” that researchers can use to configure a useful virtual environment. The implications of research using this toolkit are broad, so as such we aim to provide options to researchers that can adapt to each studies’ needs. Firstly, this involves exposing the “maze” generation algorithm to be highly customizable. This includes, but

is not limited to: the distance the user will travel in the maze, the time the user will be in the maze, the density of “turns” throughout the maze, the frequency of junctions (the intersection of more than one “segment” of the maze that requires the user to make a choice from), the size of junctions (2-way vs 3-way vs 4-way), and the inclusion of “dead ends”. Secondly, how the user interacts with the environment should also be highly configurable. This includes: various movement techniques, navigational aids, and environmental set dressing. Overall, these transparent settings will make the toolkit more versatile and robust for a wider array of research studies.

3. Broader Impacts

The aVRage paths project aims to provide researchers with a tool to analyze how humans interact with impossible environments. In the field of research, the goal is to minimize the number of confounding variables to be able to draw general conclusions from the study. However, in reality this can be very difficult to do as some variables are impossible or too expensive to control. This virtual reality project aims to solve that problem by providing researchers with an environment they control. The aVRage paths project will be a tool that researchers can customize and fine-tune scenarios for their participant’s experience as well as collecting extensive data throughout the experience.

On a larger scale, aVRage paths hopes to help spark interests on the use of virtual reality in research. More interest in the VR field brings more people and cash flow allowing for the space to expand at exponential rates.Virtual reality in turn could become the next big powerful tool not only for entertainment but for education and research projects as well.

4. Project Function

The toolset which this project aims to create functions in two ways: User side, and researcher side. In high-level, the researcher will be using a desktop environment to set some parameters for the virtual testing environment; there are three categories for parameters. The first category is the generation method for the maze in which the user will traverse in, such as the frequency of junctions, how many rooms to reach the goal, and the frequency of turns between rooms. The second category of parameter is the locomotion method that the researcher wants to test, such as teleportation, arm swinging, or joystick controls. The last and third category would be the environment’s

visuals, while we are first starting with a corn maze, we want to implement different kinds of visuals styles so that researchers can test an user's performance based on different moods and ambience.

From the moment the user appears in their virtual environment, the program will start writing two types of telemetry data about, this first kind is tracking telemetry, such as user position, velocity (if they are using a locomotion methods that allows them to traverse continuously), hand positions, head rotation, what the user is looking at, hand velocity, hand rotation. The second type of telemetry data is what we call task telemetry, which is how long has the user spent in a room, how long did they stay in a room, how long did it take them to traverse between rooms, and how long did it take for them to complete the overall test, etc. Depending on where the user is looking at, and where they are heading, the parametrized maze algorithm will start generating the next section or room on the fly for the user to traverse to, without them noticing.

Once the test is completed, all the gathered data is written into a file—such as a CSV—only locally, in which from the researcher's side, they can filter out whatever data they do not need, and apply any model of statistical research they want. Aside from written data, one want-to-have feature for this toolset is a deep replay system, where researchers can replay tests from users and have controls similar to a video editing tool, these would be pausing, fastforwarding, and reversing.

5. Legal, Ethical, and Privacy Issues

Overall, there are no legal or privacy issues that are brought up in this project. All the work will be open source in nature, with little to no user-identifiable data being captured. Ethical concerns that come along with using this toolset would succumb to the same ethical issues that come with any form of research. This would include minimizing user risk, being transparent with users as to what the research is, and debriefing users after the research is finished. However, it is again worth noting that these ethical concerns are more overarching in research and that this toolset does not inherently come with these stipulations.

5.1. Legal

The possible legal issue of concern is surrounding the licensing for the use of unity. Currently unity offers a free tier of development which allows use of the unity

engine for games that have less than a $100,000 gross annual revenue. The aVRage Paths project is a non-profit research project and has no current plans of expanding into a paid game. As such, there are no current foreseeable legal concerns in the near future; This can change if Unity ever decides to switch their stance on the free tier down the line.

5.2. Privacy Concerns

Next are the privacy issues that could arise surrounding the project. The purpose of this project is to collect data surrounding the player/user to research how the player interacted with the generated environment; As such, there is a potential point of concern regarding the handling of user data. To address this issue the aVRage Paths team will only store non-personal identifiable information from each session, avoiding things such as player's name, race, age, etc. and focusing more on aspects such as total time till completion, how many steps taken backwards, etc.

Lastly, there is a need to address possible ethical concerns regarding the research portion of the project. In the area of research there is a need to inform participants about the research they are participating in. The purpose of this is to ensure that all participants are able to give an informed consent and to avoid being taken advantage of.The current plans to address this possible ethical concern is to have researchers who use this tool to provide an explanation to all their participants. However, if needed the aVRage paths team could provide a quick welcome screen explaining the research topic to the participant and requiring them to agree if they wish to continue in the research.

6. Financing

The sole portion of this project that requires funds, while not inherent to its functionality, is for the availability of VR headsets - both to use the toolkit and to develop for it. Dr. McMahan has lent headsets to each team member that does not already own a personal headset to use for development. Furthermore, since this application is serverless, there are no operating expenses or other likewise finances - such as licenses.

7. Goals

The primary goals of aVRage Paths are:

I. Generate mazes that include *impossible spaces*.
    A. Provide parameters to tweak certain aspects of the maze algorithm.
        1. The number of two-, three-, and four-turn *junctions*.
        2. The length of *sections*.
        3. Whether or not to roof sections of the maze.

II. Players traverse the maze in virtual reality.
    A. Provide at least two methods of movement, teleporting being one of them.
    B. Gamify the traversal: add time limits, objectives, etc.

III. Collect data on player behavior throughout the maze.
    A. Number of *backtracks*, time spent per *junction,* and time spent per *segment.*
    B. Support real-time view of these statistics for the user.
    C. Create a tool to export or save these statistics.

IV. Provide an interface for users to:
    A. View player movements through the maze space, through a top-down or first-person perspective.
    B. Trigger sound and action events.

8. Objectives

Create a virtual reality research tool using Unity Engine

- Create a modular program that can have features added as needed
- Streamline the data collection and storage process
- Create a seamless UI system for both the researchers and participants

Implement a successful software development cycle

- Create a deploy and testing cycle for the code base
- Collaborate with teammates on difficult implementations
- Version control for roll-backs for major bugs
- Peer code reviews to ensure quality control on code

- Implement sprint cycles
- Attend weekly stand-ups

9. Project Motivations

The goal of aVRage Paths is to provide a virtual reality research tool that eliminates all variability in the structure of mazes.

When generating mazes, there is often a single path to the end of the maze. This implies that the subject needs to find a single path out of many to reach the end. Not only is it difficult to find this path, but the path itself may have an extreme amount of junctions and segments in comparison to the other paths. This variability in junctions and segments, and the difficulty involved in finding the right path, introduces problems for researchers when trying to understand how subjects orient themselves in a virtual reality setting. When attempting to study how subjects behave *in general*, a highly variable environment reduces the effectiveness of the study, as it could impact individual behaviors.

So, aVRage Paths aims to reduce the variability by dynamically generating the maze environment as a subject traverses it, allowing for a path to the end to be created *for the subject*. It will also allow the researcher to parameterize that path, i.e. how frequent junctions occur, how many options a junction will have, and the length of each segment.

Solving this variability in maze generation is important for the future of virtual reality. Though aVRage Paths is primarily a tool for aiding research through mazes, the research conducted is applicable to all virtual reality spaces. Understanding of how individuals behave and interact with virtual reality spaces will surely inform many decisions in the design and implementation of future virtual reality tools and games.

10. Individual Motivations

Armin Malekjahani

My motivation for developing this project comes in two parts: a) I have a huge passion for VR, and b) to help move VR research forward.

For the first part, the ability for such a compact piece of hardware to break the barrier of human-computer interaction and allow us to utilize the power of modern computers in a familiar, 3D space is beyond just interesting to me. Ever since computers were invented, there has been a raging war over how we translate the way humans work into the way computers work, and vice versa. The invention of the mouse to have a virtual pointer, using keys on a keyboard to move a character around in a game, shaking a phone to undo text—these are all ways we have tried to adapt our human abilities into 1s and 0s. By using VR, however, we can cut out this middle-man. Moving in a game no longer requires key presses—you just move! Instantly you can see users that put on a VR headset start to reach their hands out to grab something, duck at the sight of an incoming projectile, and push off (virtual) walls to move themselves. We were born into a world and taught how to interact with it. VR replicates this world and makes our user experience more than just intuitive; VR makes our user experience feel *familiar.*

While these benefits of VR may hold true, VR applications still need to be developed for the world to use. Especially for such a (relatively) young technology, this is exactly why VR research is so important. Never before have we been able to so easily “trick” the mind into taking a virtual space and making it feel realistic. As such, there are many positive and negative implications of this discovery that are still largely unknown to us. Researchers provide the evidence and empirical data that help bring light to potentially harmful, scary, or even useful effects of using VR. Furthermore, they pave the way on how best to use VR: “what navigation technique should be used?”, “how long can users use a VR headset before experiencing fatigue?”, “how can we create intuitive controls to use in VR?”. All of these questions have their answers aided by the work done by VR researchers. It is my goal, then, to help the researchers do their work and continue to bring advancements in this ever-growing field.

Jose Mendoza

Working on this project motivates me in the field of Computer Science in which I am passionate about—Computer Graphics.

When putting any test subject into any virtual environment, there are many factors that can either disorient or disinterest the user from tasks which are crucial for researcher's data collection. Most of these factors could mostly be attributed to confusing or repetitive visuals. These visual aspects could correlate into the performance of the user in a test caused by psychological ones. Computer Graphics is a pillar for VR research due to the mathematics that go into projecting virtual 3D environments into a flat screen with two images slightly shifted, or some times, two flatscreens that are placed right next to each other that also has to distort the presented images to accommodate for the natural curvature in which the human eye sees because of its spherical-like nature.

By using more graphical techniques, I believe that there is more of yet to be tested for VR, by implementing simple methods from changing the visuals of a testing environment to alter the mood and ambience to test a subject's performance, to using something more complex such as *impossible spaces* to remove repetitive tasks from the subject's perspective. These implementations can help solve current problems for research that perhaps are not being looked at from the right angle, and offer better alternatives to a research topic that is niche in the current day.

Alexander Peterson

The aVRage Paths project motivates me in various ways: my curiosity for the game development process, the core maze generation algorithm, and the uniqueness that VR brings when studying *impossible spaces*.

Game development interests me because of the combination of technical software engineering and artistic game design. I am passionate about the technical side, appreciate the artistic side, and want to learn more about both. There are numerous skills on each side that go into building an experience that a player can enjoy, and I think aVRage Paths will give me a chance to learn many of those skills.

The maze generation is another unique and interesting challenge to aVRage Paths. Using an algorithm to create a maze structure that contains *impossible spaces* is a new concept and thus will likely be challenging to develop. How can you create a structure that changes dynamically around you as you traverse it? How do you ensure the paths you create are all relatively fair? How can we make sure the maze is *impossible* but *feels* real? How can we parameterize the maze so that certain properties of the maze are controllable? These are just a few of the problems that we might face when developing the algorithm, and I'm interested in helping solve each of them.

Patrick Mesquita

My motivations for the aVRage Paths project are around the psychological aspects that this project will explore and test as well as learning about game development and VR.

Game development is an interesting area because it is an area where you can have so many interesting optimization algorithms occur due to hardware limitations. An example is the famous “fast inverse square root” algorithm from the Quake III (1999) game, which used some fancy hardware manipulation that decreased the needed calculation time of an inverse square root by a multiple of 4. That is not all I’m interested in game development either; I grew up on video games so being able to participate and learn what goes into designing and making a game is something I have always wanted to do.

I also have a lot of aspirations to learn about the VR portion of the project. I think VR is a tool that continues to improve drastically and will be an integral tool in education for many fields in the future. I can envision things such as VR being used as a practice tool for things such as surgeries, driving/piloting simulations, and even rescue mission simulations.

VR will also be an important part of research in the future and being able to participate in a project in the early stages of VR is very exciting. The human mind is interestingly very complex and ever since we have begun to study it we discover things that continue to show that we still know so little about it. I think VR as a tool for research is something that can help us understand the psychological aspects of the human brain. A virtual reality simulation would allow researchers to eliminate/control variables that would be impossible to do in the real world; This would allow for studies to be conducted in a perfectly controlled environment which would allow for more accurate conclusions to be drawn as most environment variables would be eliminated. The aVRage Paths project is one of these projects that I hope we can develop to show just how powerful VR can be in the world of research.

Pierce Powell

My motivation for aVRage Paths falls primarily behind getting to explore how navigating through an impossible space within a virtual environment affects people

psychologically. Specifically, if we were to run a control on a pre-configured maze, versus our standardized maze with impossible spaces, would people even be able to realize that they were traversing an impossible space? To what extent could we utilize impossible spaces before it is evident to a typical user that the space inherently doesn't make sense? I believe the implications of this work and the studies to be done using it could massively impact the way we think about virtual landscapes as a whole. I feel like in the current structure of making realistic virtual environments, we attempt to make it as indistinguishable from the real world as possible. However, this typically makes development a lot harder for things that are seemingly simple as the real world has a lot of things going on, so I believe our project could be the stepping stone into treating virtual environments as separate from the real world, as maybe they affect people differently than traversing in the real world. Another big motivation that comes with this project is giving people a solid testbed to further locomotion studies. I personally know quite a few people who conduct trials for locomotion studies but constantly complain about there not being a solid virtual framework that they could use to test all of their nifty new technology. I believe our project will fill this gap and make the lives of my friends a lot easier, as well as giving them a more rigorous and immersive space to extract a lot more meaningful interactions. Finally, as a personal growth motivation, I have a passion for game design but haven't had a lot of long term projects that I can really use to flex the capabilities of unity. I have always been interested in impossible spaces and procedural generation as a whole, so getting a project that can tie all of those individual aspects together is really refreshing.

Jonathan Giat

I started using virtual reality in 2017 with the Oculus Rift CV1 headset. I was introduced to it by my friend and was immediately hooked. Initially, I was captivated by the immersion the device was able to capture, but as time went on, I was able to fully grasp the possibilities with the space.

I have a few motivations for being a part of this project. With VR being a great tool for simulations, I wanted to be a contribution towards something I feel is genuinely impactful and helpful within the space. Additionally, with the software we hope to implement, I feel the blend between VR compatible experimentation and desktop friendly experimentation will be a great way to introduce people, and bring more people, to the world of virtual reality. Furthermore, I'm intrigued by the psychological impacts that researchers will investigate.

Many pieces of software are already taking advantage of the tool. In a similar space of mixed and augmented reality, there is a large amount of research based projects, and I would like to bring more of the research genre, and more researchers, to the virtual reality community. I have had personal experience with individuals favoring the use of MR and AR over VR. While each device has its application, I feel virtual reality is much easier to develop, and it has more utility than its counterparts.

I mentioned above how I would like to bring more people into the sphere of virtual reality, and I feel it is helpful to understand why many have not entered first. Motion sickness is a great problem for many, as well as the price point for the hardware. If we can have a desktop application alongside our VR application, we can have people unable to use the hardware to still be involved with the process, which I think is a great compromise. Additionally, these headsets are not cheap, but relative to other mixed and augmented reality hardware, it is a bargain, which I hope can be appealing for individuals when considering their budget for research.

Lastly, I hope this project can be a great utility for psychological research. I have always had a great fascination with the field and have desired to merge software development with psychology, and here, I have the opportunity. I am excited to see how the software our team creates impacts the world of psychology, hopefully helping people in the future.

11. Project Requirements and Specifications

- Explicit definitions of prefabs used for maze generation, accompanied by proper documentation such that a researcher would easily be able to recreate them in a different look to obtain similar goals.
- Being able to define the total number of segments and joints within a maze and generate those dynamically as a user progresses.

- As many metrics as possible (total time of completion, right turns, left turns, time spent at each junction, time spent at each segment, position and rotation, hand placement, and whatever else we can think of)
- Arxiv publication with documentation so it's intuitive for researchers to tweak the framework

12. Consultants, subcontractors, and suppliers

Since aVRage Paths was a student-driven senior design project there were no subcontractors or suppliers involved in the project. The project is entirely software based and no necessary hardware is needed for developing the project itself. Although the toolset does require a VR headset, the team leaves that up to individual researchers to acquire the necessary equipment to run the program themselves.

The aVRage Paths team did have some consulting work during the development of the project. The consultant for this project was Dr. McMahan, a researcher in the field of VR. Since this tool set is built for research in the VR space, having Dr.McMahan as the consultant for the team was essential for the project. Dr. McMahan provided insights into what researchers look for during an experiment and some of the headaches and problems that occur during that experiment. As such a lot of the tools developed in this program are based on aiding researchers in this field with as much customization as possible to alleviate such problems and headaches and allow the research to fine tune the experiment to their own individual degrees.

13. Budget and financing

Since aVRage Paths was a student-driven senior design project there was no budget or financing required for the development of the project. The main reasoning is not only is this idea sponsored by students but also because behind this is the aVRage Paths program is a software program that aims to be a “tool-box for researchers”, as such most of the software is developed by the students on the team.

Although an important note is that since this is a VR project the use of VR headsets is needed to develop and test the software. Dr. McMahan did inform the team that if headsets were needed that they would be provided, although this was not needed as all team members already had headsets.

The project also requires no servers to host currently as Dr. McMahan's current plans are for local use throughout the university. This also means that no database will be used to store information from participant sessions, such files will be stored on the local machine. If the project is successful and were to be hosted online in the future Dr. McMahan would handle the costs associated with that endeavor.

14. Group Member Ideas

Jose Mendoza

In our first meeting with our sponsor, we discussed the concept of changing the visual style of the virtual environment. Since I have a background in Computer Graphics, I have thought about a few implementations that could be used for this suggestion. The ideas that I have so far all start with the same base. We first would have plain geometry that is called a white box, these basic shapes would be of a single color, and could be altered in different ways:

1. A specific folder would be interpreted as something similar to an atlas or color palette, in which it would apply different textures or materials to the white box in order to change the look of the object itself.
2. Similar to the first approach, but instead, there would be 3D models that could possibly even be animated. This approach would change the overall visual geometry of the white box, but it would not do so for the collision geometry.

Through the use of shaders, which are small programs, which are scripted in specific programming languages that the GPU runs, and are varying from graphics API to one another, that are run in real-time on 3D or 2D objects to process graphics primitives—such as either vertices, indices, or lines—they could be procedurally animated. However, simply because the look of an object changes, it does not mean that it would change the mood of a scene, or alter the feelings of a user. Lighting also contributes to the overall visuals of an object or scene:

Fig. 2. Renée, V, “10 Ways to Create Different Moods with Lighting.” No Film School, 6 May 2017, nofilmschool.com/2017/05/watch-10-ways-create-different-moods-lighting.

So I came up with the idea that in the same reference folders where we have premade asset packs that we will come shipped with our toolset, but the researcher also will be able to set their own textures and models for visuals of the maze that they might desire, there could also be some kind of data file that holds lighting information pertaining to the mood of the scene. This data file could contain an arbitrary number of lights, all of which can contain their own color, position, and luminance.

One final addition to my idea was the addition of post-processing. This is a Graphics concept where the modification of pixel data is made after a frame has been created, hence its name. Some post processing effects that could be added could be Bloom, which is an illusory effect where light pixels bleed out to surrounding ones, creating the visual effect of a very bright light; Vignette, this effect darkens or desaturates pixels towards the edges of the screen, this helps focus the attention of the user; lastly, Color Grading could be used, this technique is used to alter a rendered image’s color and luminance, this is typically done by applying a tone mapping curve, which is what alters pixel color values, by clamping them (usually from 0 to 1 in standard dynamic ranges for colors, but it can go above 1 in high dynamic range color spaces); an alternative to this would be the use of a Look Up Texture (LUT), and it borrows the same concepts that color that Color Grading uses, in that it it also could alter a rendered image’s color and luminance, but rather that using a curve to adjust this values, it uses an assigned texture that looks like a color palette that is used to affect the final look of pixels. All of the aforementioned effects do not have to be used individually, they can be used like a queue of what effects have to be done first, for the render to apply, for example: by applying Vignette first and then a LUT or Color Grading, the color of the Vignette would be altered since the the latter post processing technique

would be receiving the previous image without recognizing if anything else has been done to it; this goes for any post processing technique, they are agnostic.

Some of these post processing techniques might not be as effective, or they might even hinder the user's performance during tests, but we have not enough pre-existing information about this kind of implementation, so these are things we will end up testing for ourselves in very early phases of development.

Jonathan Giat

Companion Website

During our meetings, our group has many discussions of what specifically we want to implement for the researchers. I feel a great utility would be a companion website which allows researchers to view their data in a more convenient way. A few of the key components we discussed were as follows:

- A map of the whole maze which the researchers can "step through"
- Statistics of our data which show some of the following:
    - Statistics about the current maze section
    - Information about the individuals hardware transforms
    - Overall data on the maze
        - Time to complete
        - Average time per section
        - Counts of each binary decision made
    - Each of these should be displayed with a good UI

Since this is still early within the project, this list could be expanded on heavily in the future, but let's discuss more deeply our key points.

I would imagine that since we are utilizing Unity to develop our project, it would be convenient to develop this companion website in an embedded web application also made through Unity. There are many games which utilize this feature, so ideally, we would be able to do the same. Using Unity would allow us to take advantage of the plethora of utilities and services offered in easily constructing a nice user interface, which is a large component of why this website would be developed in the first place.

14.1. Step Through

Having the ability to step through a maze would be extremely helpful. Currently, the team has discussed having the desktop counterpart application showing a top down view of the maze. The website should show a similar view, allows a large grasp of the whole area traversed. There are a few issues with this approach, which leads into the “step through” solution.

The way our maze would be developed involves having many paths crossing over itself, hence the name impossible space. It would be impossible to show the two sections shown at once, unless they overlap and that would look extremely messy and unintuitive, which is the exact opposite of the purpose for this website. So we would provide researchers the ability to step through each part of the maze. This way, there would be no overlap, as only one section would be shown at a time. Ideally, it would be nice to show some path taken over the course of the whole program, perhaps showing overlapped sections in a different color, or perhaps showing it in a subtree where it can be expanded to show the different paths taken. If that is too much for the scope of the project, which I imagine it would be with this being a subcomponent of the main assignment at hand, it can be reduced to just the step through feature.

On a more technical level, if we have already exported the data into something like a JSON or CSV file, it can be reconstructed based on each segment, and we would simply iterate over the segments based on what the researcher would like to see.

I would like to expand more on the subtree concept, as I feel there could be something more intricate and interesting to explore. A path can overlap on itself, but if it does, we would need something concrete to tell us that there was an overlap. Would each segment be made within a single tile, and we could then conduct a binary search over all tiles used to see the overlap? Assuming that there is a solution to find whether a tile has an overlap, the way we show the overlapping sections, as I imagine, would be a list of all the different sections, and the researcher can click through it until they choose to go back. Theoretically, each tile can only be overlapped at the tile level, so there would not be any form of recursion to worry about. This makes our lives a bit easier when it comes to implementation, since there would only be the main screen, and a sub-screen which shows our overlapping tiles.

14.2. Statistics and Data

The other important aspect we need to present to the researchers is statistics. We are expected to export the data into a CSV file, and provide that as our output for

researchers to do with however they would like. Theoretically, there are a few common values that all statisticians look for, such as mean, standard deviation, median, etc. If we are able to process those values for the researchers prior to them arriving, it could be a great time saver for them, as well as make our product more appealing and easy to use.

This concept of preprocessing is not necessarily specific to the website discussed here, but it can be applied, as we would not simply upload a raw CSV file to a website. That would look terrible and would not be consumable to anyone attempting to decipher patterns and significant parts within the experiment. Consequently, the statistics should already be calculated and nicely packaged for the researchers use and for our use within the website.

Furthermore, it is vital that the data is presented nicely. This should be done in a few different ways based on what is relevant to the researcher is viewing at that time. If the research is viewing a maze segment, data about that segment should be visible. Alternatively, if a home page of sorts is created, some type of overview of all the data would be helpful to show, as it is not specific to any one maze segment. Doing so would likely involve having our data separated into different sections prior to arrival at our website, which would both help with the website creation and the data given to the researchers independently.

There is much discussion about how this website would present a lot of the data, and the application of statistics within the experiment. It is important to note that this is not intended to replace or get rid of any utility provided within the original output files from the VR experiment. This website is simply intended to provide a better consumption of data in case the researchers are not feeling enticed to show the data in a manner specific to this software. Since we are the ones creating the program, it would be easiest for us to craft this tool, since we are most familiar with the data being inputted and outputted. Leaving this in the hands of a researcher, perhaps unfamiliar with the underlying coding, could be problematic and an inefficient use of time. Providing a blanket software that can be applicable to most researchers would save the people involved within the research a great amount of time and effort, assuming it is done to the standard expected by the researcher. If this is expanded on later, further research shall be conducted to determine what is the best and optimal way to present information to the researchers, as our expectations as the software developers may be different from the people actually utilizing our software.

14.3. Website Implementation

We also have to be concerned with how we implement our website. We want to ensure it is easy to provide the data to be viewed, which leaves us with a few options:

- A service which connects with our main application and uploads it to a server once the experiment is completed.
- The ability to upload the data exported from the software.

If we expand on a server approach, that would definitely be more streamlined and easy to use, however, with the extent of this project already in the limited time available, it seems unlikely that would be feasible. However, being able to simply upload the data should not deviate too much from the already intended system. If we export the data from the initial VR software, it would be convenient to then simply upload it directly to the website, where it could then be parsed and generated.

Throughout this, I have mentioned many times that this will be a website, and not simply a standalone application, and there is a good reason. Standalone applications can be stressful and cumbersome to download and install. There can be issues with differing operating systems, or perhaps a limited amount of storage space available, not to mention that simply downloading an executable file from the internet is not always viewed as the most safe option. Having a website would avoid any of those issues, and be much more appealing to a researcher. Being more accessible means we can produce a product we are sure can reach more researchers, and thus, be more helpful and impactful.

Although a website seems like a great idea, there are a few issues which are important to mention. Running a program created by Unity on a website can be problematic. Performance issues are a main concern, with also the issue of not being able to develop or utilize the software offline. There could also be an issue of not having the space on a server, or the budget to upkeep a site with the storage for a full program. To solve these issues, we can provide our executable file in addition to the website version in the case of these problems arising. This will be a sort of safety net in case we cannot continue to support the project over time, or any individual team or researcher is unable to meet the requirements of the program on a web browser.

Over the course of this project, I hope to find time and an opportunity to create this utility, perhaps developing our main software with the idea of expanding. This would allow us to easily create the website, or any expansion in general, in the future.

1. **Impossible Space Detection**

I believe being able to detect impossible spaces could have a huge impact when using our platform to conduct studies. In our maze, it would technically be possible to make four right turns and end up somewhere completely different as opposed to making a circle. I believe that if we were able to keep a small artifact of previous maze paths then we could check when a new path has been overlaid on top of a previous one. With this information stored, the researcher could then do something such as (raise your hand or press this button if you feel disoriented). And with that data recorded, the researcher should be able to statistically back whether people actually realize that they're in an impossible space or not.

1.1. Additional Metrics and Tools for Collection Regarding Impossible Spaces

To coincide with my previous recommendation, I believe having additional metric collection around impossible spaces could also have large ramifications for studies conducted using our locomotion framework. One thing that I think would be super useful is time spent and turns made before entering an impossible space. I think this would have interesting ramifications as it's one thing for a user to not realize they're in an impossible space after making 20 turns, but I believe it is a whole other category if they were to not realize they were in an impossible space after making say three turns. Gathering this metric and metric alike (time spent between impossible spaces) would then allow a researcher to gauge a participant's threshold to not realizing impossible spaces and lay the statistical groundwork to allow future VR researchers to exploit impossible spaces with more confidence.

Finally, I believe allowing the researcher to pre-define button behaviors on the user's controllers could be largely beneficial for quick data collection. Something along the lines of "press X every time you feel disoriented", would allow for the researcher to gather real-time emotional responses to our environment. I believe this would lead to more accurate and interesting data, as the user would not have to "ballpark" a time and area to talk about after completing the study in a survey. Furthermore, if we made a UI for the control layout for the researcher to fiddle with, it would make for an effortless experience and allow them to switch what emotions they were looking for with ease between studies.

1. **Options for changeable soundtracks**

I think that in the world of VR not only is what you see important but the sounds that you experience in the virtual world are also very important. I think being able to provide the researchers abilities to upload specific sounds they want to play in the maze is going to be a very important addition that can contribute a lot to the research. My reasoning for this is for example if you are just placed in the corn maze but there are not any ambient sounds as you would expect there to be when outdoors then there could be a disconnect which could affect the research.

It would also be interesting to explore things such as how do people react to being presented with an impossible space when there is suspenseful music playing rather than no music at all. Would the participants feel freaked out? Would they backtrack? Would they start running? There are so many questions that adding soundtracks or specific sound effects can add to the research.

2. **Options for objectives**

Creating objectives or tasks that require completion before finishing the maze could be an interesting option for our tool set. For example, an individual might be able to find the exit before completing their task, and as such they have to return to the maze. Upon returning to the maze, there are many topics to explore like once they find the object how many participants are able to quickly find their way back to exit?

It would also be good for research to explore how a sense of urgency affects memory such as if a maze has a time limit how is the participant's decisions and path taken affected.

3. **Options for obstacles**

Obstacles is something I recommend the team implement into the tool set. The purpose of the tool set is not only to explore how participants interact with things such as impossible spaces but also how they interact with other methods of movement in VR games. For example, a common use case for our tool is to research how participants react to walking vs teleporting short distances vs crawling as forms of movements. Including obstacles that can block the way would also be an important data point to explore.

An important thing to note is that obstacles do not have to just be limited to things such as short walls or boxes blocking the player's path, but things such as "enemies". For example, we could include an option to include a malicious AI that targets the player and attempts to kill them and ends the experiment.

Some interesting obstacles could also be traps that the player could visually see such as spikes they can try to jump or navigate through, or swinging spike balls that could hit them and end the experiment. This would be interesting to explore as it gives insight to decision making when faced with a “life” ending threat, do players attempt to go through the current path or do they turn around and attempt to go around via some other route.

Fig 3. A laser hallway as an obstacle the player could try and navigate through.

Another set of obstacles could be in the form of path terrain. An example of this would be a player has to perform a jump down from a small cliff like path, signaling that they could no longer return. Would the player take the path essentially trapping them if they make a wrong choice or would they turn around until this is the last possible option.

Alexander Peterson

1. **Biometrics**

An interesting datapoint to collect would be heart rate throughout the maze traversal. While various stimuli are occurring, a physiological response is likely to occur. For example, when a frightening stimulus occurs we could expect to see a rise in heart rate. There are likely more relevant scenarios where this datapoint would be relevant, so I think it would be worth collecting.

Something else worth considering is eye tracking. To study the effects of *impossible* spaces, it would be useful to know what exactly the player is looking at. If we were able to collect data on how long something is looked at, whether something is looked at at all, etc, it could be useful for understanding how we react to said spaces. This could be implemented with what is within the viewport of the player, but a more precise tracking would be interesting as well.

2. **Chase Scenario**

We could take the idea of a frightening stimulus to its extremes and introduce a chase scenario. This would have the player find their way through the maze while being chased by some type of frightening figure. This would allow for studies to be done on how individuals under stress and adrenaline perceive *impossible* spaces and what those spaces do to their understanding of getting away from a pursuer.

3. **Memory Test**

Some type of memory test where players are asked to repeat the path they just traversed would be an effective way at understanding if we find difficulty in traversing *impossible* spaces. We could introduce things like time limits, varying lengths of paths to memorize for further control on the study.

Armin Malekjahani

1. **Live User Notifications**

Users can be notified during the simulation about some “event” that they just partook in. For instance, if the user just left an impossible space during a simulation, they would get some sort of prompt about it in their view. The implications of this serve that the user can then try to “learn” how to navigate the maze, use their travel techniques, etc. There would be an interesting dynamic with how the user would interpret and utilize this insight into their actions in real time. Furthermore, this same notification system can be hijacked to further “trick” the user. For instance, users could be told they made 5 correct turns in a row! But in reality, the user has no influence on their eventual outcome in traversing

the maze. This positive reinforcement could be studied in context of how it affects the users ability to navigate, overall mental state, and more.

2. **Minimap**

Providing the user with a glanceable minimap could result in interesting dynamics. To start, this minimap would have to be dynamic since, by nature, the space itself is dynamic. This could give the user insight into the "behind-the-scenes" nature of creating the impossible spaces, possibly altering their pathing decisions. Furthermore, parameterizing the minimap could let researchers study how effective minimaps are under certain conditions and with certain capabilities. Again, this is an especially curious thought since the space is *impossible.* It should be noted that making this minimap would be rather difficult to implement correctly. Since the maze is generated dynamically, a true top-down view of this space may be too revealing of what goes on behind the scenes for the user. One way to prevent this revelation is to only show parts of the maze that have been "traversed". That is to say, only parts of the maze that the user has stepped foot on or sufficiently traveled on would show up on the map. Other parts of the maze that can be seen by the user but do not yet render on the minimap could appear in a sort of fog, showing that that part of the maze has not yet been "discovered".

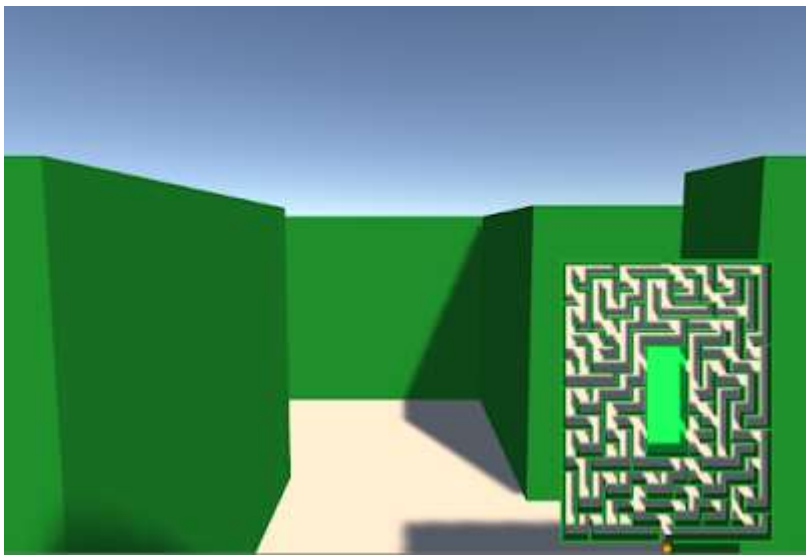


Fig 4. HalloNea, "Maze with minimap" *Maze Generation in Unity, 19th June*

2015, http://mazegeneration.blogspot.com/2015/06/more-decoration.html

15. Use Case Diagram

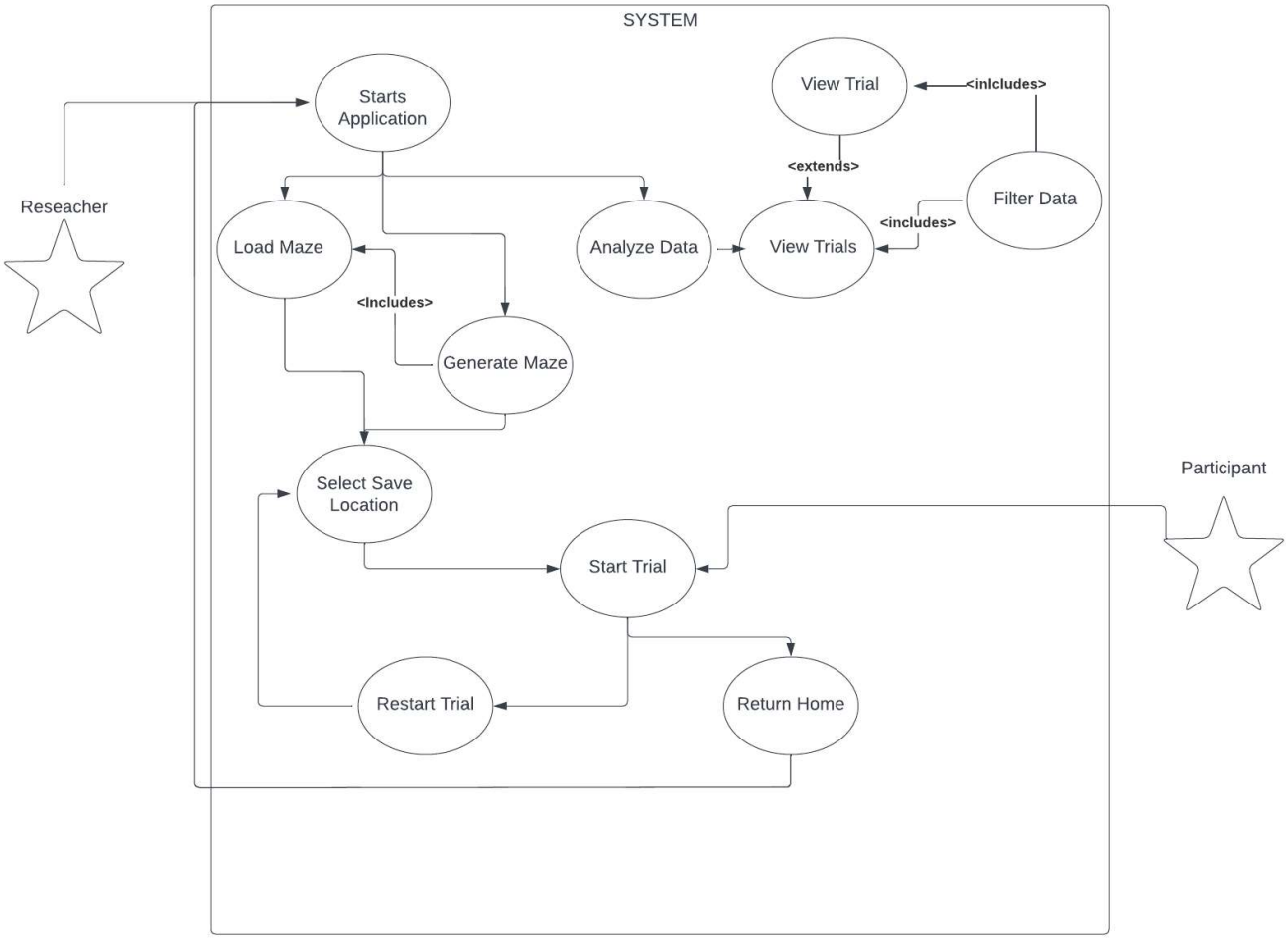


Fig 5. UML Use Case Diagram

Made via LucidChart.com

16. Block Diagrams

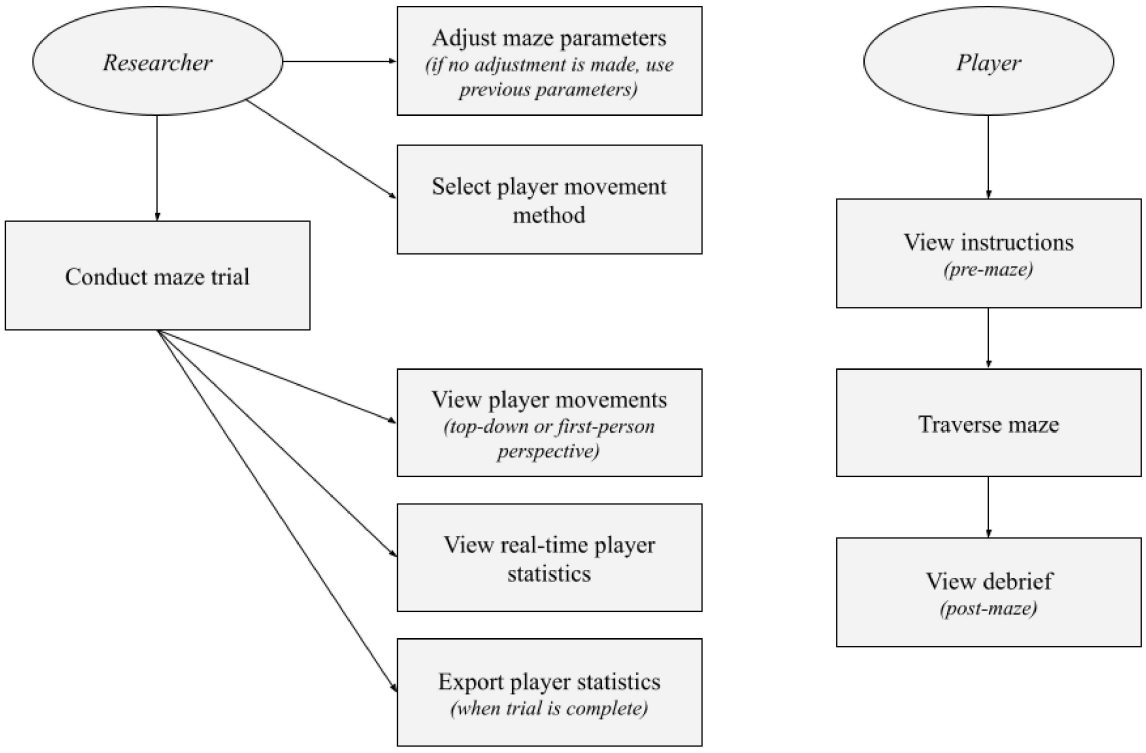


Fig 6. UML Use Block Diagram

Made via LucidChart.com

17. Telemetry

17.1. Overview

Telemetry is the recording and storing of data relevant to a system. While data capture is often useful to users such as developers, the nature of our project and the intended consumers make it especially important. Researchers dig through, plot, analyze, filter, collate - researchers do a lot with this data. Data is the backbone of drawing conclusions that hold statistical significance and show hard truths about what at

hand is being studied. It is important that any and every piece of data that could be used to show correlations is at hand and ready to use.

For these reasons, a major pillar of this project is to create a robust yet easy-to-use system for recording and storing data about the simulation - a system for telemetry. It is important that the modular design of the project is similarly backed by a modular system to record data from additions to the project, whether that is made to the base toolkit or through additions made by the researcher / “3rd party” users. This data will be recorded, stored, and exported in fashions that are cognizant of how researchers currently treat their data.

17.2. Research data: a breakdown

In constructing a robust system for data capture, it is important to understand how this data is going to be used. In this way, we can ensure the system is built with the end-user in mind: researchers. Since the start of the scientific method, studying an experiment or making an observation has always been met with some sort of data capture to go along with it. As such, the ideas around gathering data, cleaning it, and analyzing it for research have been refined over many years.

In modern-day research, almost everything is digital. This is especially true for the pipeline that data from studies go through. In general, data from research goes through a three-step process:

1. Data Capture
2. Data Cleanup
3. Data Analysis

Fig 7. A representation of the pipeline for data in research studies

17.2.1. Data Capture

In the first step, everything and anything that can be quantitatively or qualitatively observed in a study is recorded to some medium. From paper surveys to human emotions to GPS tracking data, researchers aim to get as much data as they can down on some sort of hardcopy memory. It is not so important in this step that the data is ordered or sorted in some way - just simply that it exists and can be looked at after the studies are finished. The way in which this data is captured completely depends on the type of study, what is being researched, the scope of the project, the capabilities of the researcher, costs, etc. Our toolkit will provide data capture mainly / exclusively for quantitative data, such as maze generation and user interactions.

17.2.2. Data Cleanup

In the next step, researchers make an effort to overall make the data more useful by “cleaning it up”. This involves getting rid of meaningless data, organizing data, and converting into formats that can be utilized for the final step. Meaningless data can come from information that is nonsensical in a specific study, redundant data that can be condensed to be more succinct, or even data that was once considered useful but now is no longer. Overall, this process helps hone in on what data needs to be focused on. Next, organizing data involves adding structure to how the data is grouped together, adding labels to various sections of the data, etc. Since so much data is captured upfront when the studies are conducted, it is important this data be made efficient to look over and work with. Finally, data needs to be formatted in ways that can be used by various systems, programs, and research standards. In digital formats, this can include formats like JSON or CSV, which are (currently) commonplace for popular data programs and systems.

17.2.3. Data Analysis

Lastly, data is taken and churned through formulas, machine learning models, statistical analysis, regression models - data is analyzed. While the first two steps are focused on *having* the data, this last step is focused on *using* it. Researchers plot various information against each other to find correlations (or the lack-thereof it), go through standardized tests described by large scientific bodies, and use advanced AI to extract features all to help come to statements about how the overall nature of the study led to information that produces some sort of meaningful conclusion. There is an

endless amount of ways this data is analyzed, which shines importance on why this data capture must be agnostic and moldable to the widest range of uses. One specific use case kept in mind while creating this project is the use of **R**, a programming language popular for statistical analysis. It is important to our toolkit that there is a clear pipeline for information to go from our data capture into analysis done in **R**.

17.3. What data to capture

There is almost never an excess of data that can be captured in research. In the best case, the researcher just has to spend more time filtering and refining the data for use. In the worst case, the researcher fails to create a statistically significant conclusion that would otherwise have existed if further data was captured from a study. This worst case leads to failure to produce a useful product, a waste of time and resources (both from the researcher and the involved users), and more development time on the "backend" of research to reiterate and again isolate which data need to be captured. This is all to say: more data is better.

In this project (but also generally), there are two major types of data: quantitative and qualitative. For now, we will just cover quantitative data capture as it is the most relevant type present in this project. It should be noted that qualitative data can still be found throughout this toolkit, such as in noting when users realize they are present in an "impossible" space.

We will further break the category of quantitative data into subcategories, but this time in the specific context of this toolkit. The three categories are "tracking" data, "task" data, and "maze" data. Lastly - before breaking down each one of these categories - it is important to explain the difference between *continuous/polling* data capture and *event* data capture. The former described data capture that happens at some consistent rate, no matter how that data has changed since the last capture or what has happened since then. On the flip side, the latter described data that is captured only when there is a specific event or "trigger". For instance, if the user completes the maze, capturing the time it took to complete the maze would fall under the *event* data capture category. Below, each datum will have a qualifier for one of these types of data capture. NOTE: These are only the preliminary data that will be captured as shipped with the toolkit. Any other data can be captured as researchers extend the toolkit with their specific needs, as discussed in Extensibility.

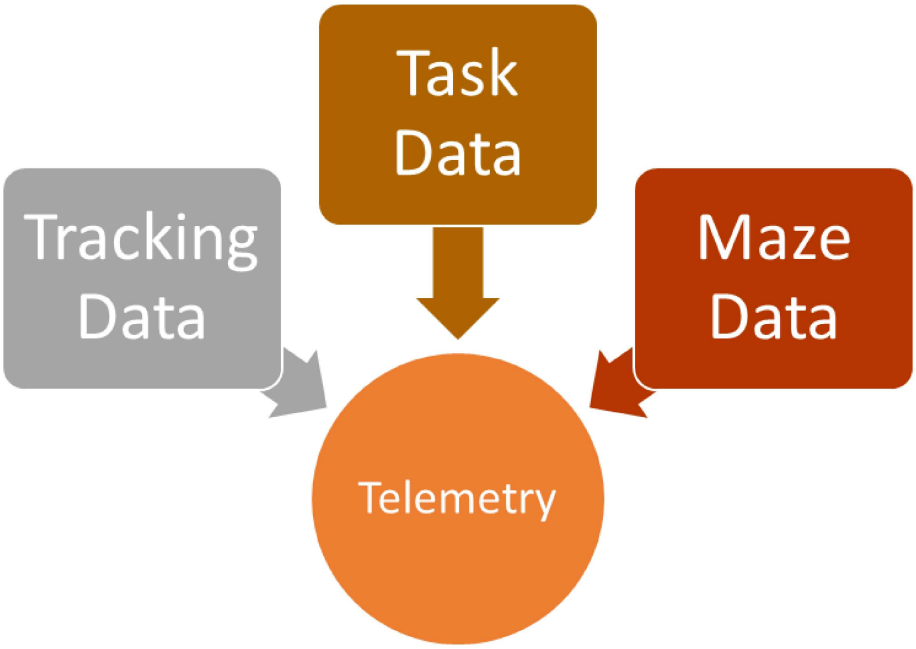


Fig 8. The culmination of these three datum make up the telemetry system as it comes with the toolkit

17.3.1. Tracking Data

Tracking data is that which is more agnostic to the project specifics. While this data does not reveal much about the project specifics, it can still be analyzed with various methods that may show useful down the road. This data includes:

- User Position (continuous)
- User Rotation (continuous)
- VR Hand Position (continuous)
- VR Hand Rotation (continuous)
- Button Presses (event)

17.3.2. Task Data

Task data is specific to what the user does in relation to the simulation. This data is much more “useful” at face value as compared to tracking data because it is honed specifically to data that would respond to what researchers are studying. Furthermore,

the data captured under this category is more likely to change per study / use case, since the study being taken can highly influence what categorical data is relevant. This data includes:

- Time spent at a maze segment (event)
- Movement Uses (event) (notes below)
- Time spent at junctions (event)
- Average speed (continuous)
- Time to complete maze (continuous) (notes below)

Notes: “Movement Uses” is a general term meant to record data specific to the used movement technique. For instance, if teleportation is being used, relevant data would include: teleportation rate (teleportations per second), average length of teleportation, amount of teleportations, etc. Other movement techniques such as joystick movement would have their own respective data.

“Time to complete maze” is not always a worthwhile datum to capture. One way to define the “end” of the maze is for researchers to choose a certain distance they want the user to traverse before “reaching” the end. In this case, the time to complete the maze is variable and directly correlated to the speed of the user. On the flip side, the researcher could instead make this end condition to indeed be time-based. For instance, the user might be put in the maze to traverse for five minutes, no matter how fast or slow they traverse the maze. Once the time threshold (five minutes, in this case) is met, then the next junction will lead to the end of the maze. In these scenarios, the time to complete the maze is practically set in stone before the simulation even starts.

17.3.3. Maze Data

Maze data is specific to the generation of the maze throughout a simulation. Since the maze is random (but still standardized), researchers may want to know the specific conditions that the user was put through which influenced their experience throughout the simulation. While researchers will play a role into how the maze is generated, most of the time there is still some level of variability to the maze. This data includes:

- Frequency of each type of junction (event)
- Average length segments (event)
- Average “Straightness” of segments (event) (notes below)

- Number of impossible spaces (event) (notes below)
- Per-segment data (event) (notes below)
    - Time spent on segment
    - Straightness of segment
    - Movement Uses
    - Length of segment

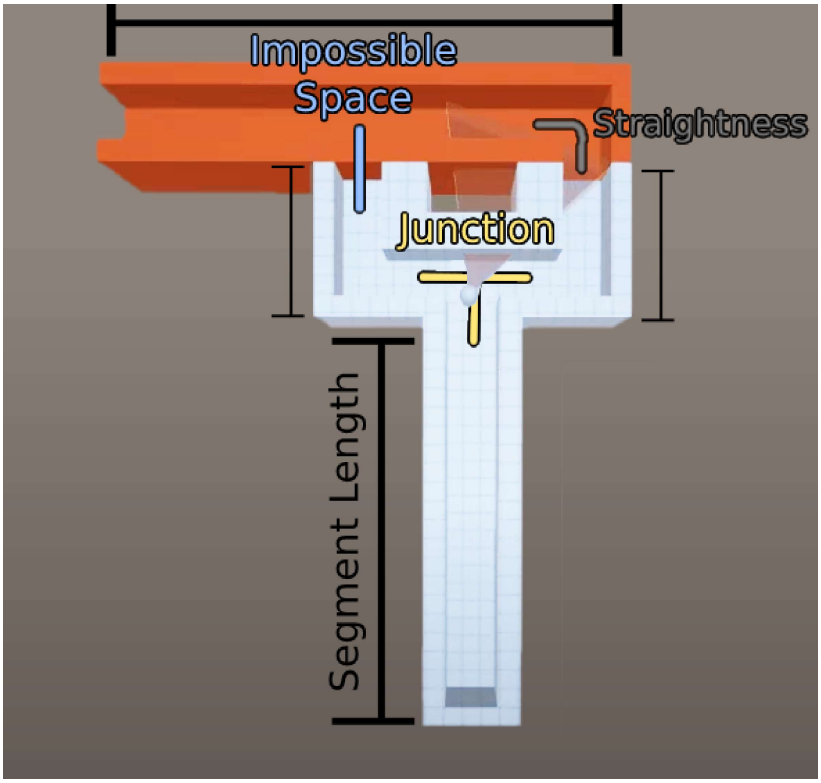


Fig 10. A visualization of maze data

Notes: “Straightness” refers to the nature of segments in the maze. While segments do not pose any options for the user to make, they can still play an influence on how the user acts. For instance, “jagged” segments might be easier to traverse with teleportation vs joystick movement. Furthermore, they can make the maze feel like it is longer, shorter, more complicated - they can add qualitative “personality” that is then perceived by the user, possibly changing how they traverse the simulation

“Number of impossible spaces” refers to how many sections of the maze are impossible. While not all segments have a possibility to overlap, some do. The

frequency of how many possibilities for impossible spaces there were in the maze can give insight into the relative “difficulty” of the maze.

“Per-segment data” is data captured for each individual segment in the maze. This provides more granular insight into how the characteristics of a single part of the whole simulation might have stand-out features that are useful to the researcher. For instance, if segments with a low “straightness” number sport considerably larger amounts of time spent as compared to the straighter segments, conclusions can be made from these data

17.4. Our Approach

Considering all the details and caveats of research data, we will implement a system that takes these points into account to ensure researchers have the tools necessary for useful telemetry. As such, it is important that any and all aspects of the system cohere to standards that are in line with these caveats, producing an efficient workflow for researchers. Our system will stand on the following pillars:

1. Subscriber-Publisher model
2. Unstructured Data
3. Extensibility

17.4.1. Subscriber-Publisher Model

Firstly, the subscriber-publisher model is a system design we will use heavily throughout the telemetry system. One way to view telemetry is as a peering eye, observing the doings of the toolkit. Data capture is built as a layer *on top* or *around* working systems that actually make the hamster wheels turn. As such, it is important that the data capture can easily be “injected” into various parts of our toolkit. That is to say, the maze algorithm or user interactions or researcher platform should not have to be made with data capture in mind. Instead, our telemetry system will utilize the subscriber-publisher model to allow various parts of the toolkit to “publish” the data they deem useful for capture, in which the telemetry system will “subscribe” to. Was a segment generated? Tell the telemetry manager. Did the user take a step? Tell the telemetry manager.

One important note to make about this model is that we will directly take and implement a concept discussed in the Tracking Data section: *continuous* vs. *event* data capture. The subscriber (the telemetry system) should be able to support publishers

(any part of the toolkit) to publish either / both types of data. Since data should fall in either one or the other category, this will ensure that all different types of data can be easily captured while using this model. Realistically, this will involve two different services provided for by the telemetry manager: a Continuous service and an Event service.

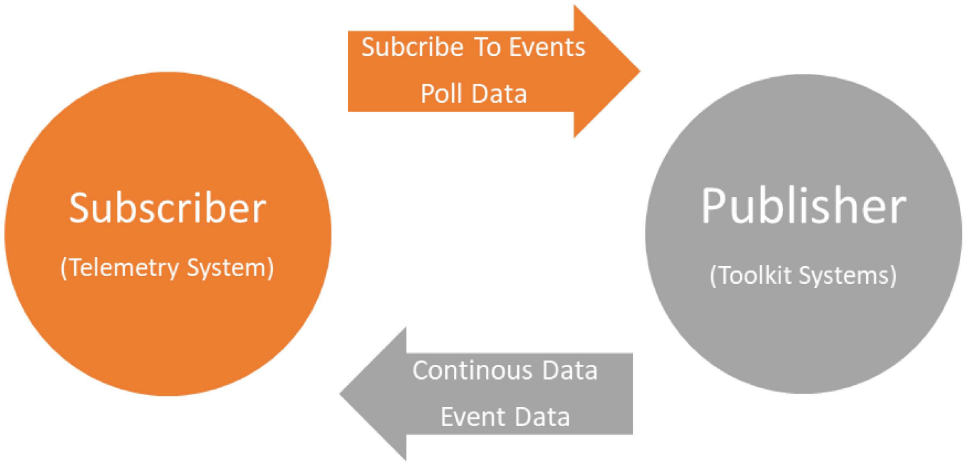


Fig 11. The relationship between the telemetry manager and the toolkit

17.4.1.1. Continuous Service

This service will take a callback function to invoke at some given polling rate. At each time interval, the callback function will be called. The callback function will then return the data for that timestep, which the telemetry manager will then record. The use of a callback function makes this polling agnostic to *what* is being polled; whether a score is being calculated, a transform is being read, or an average is being taken - the data is polled and recorded for that timestep.

17.4.1.2. Event Service

This service will “listen” to some given function to be invoked, in which it will capture the given data and record it. These “events” (which are just function calls) can be invoked at any time while the telemetry manager is listening. There is no expectation of when or how often these events will be triggered, and no order in which publishers

will publish their events. “Packages” of data will be sent to capture the data for the event, and the telemetry manager will record them.

17.4.2. Unstructured Data

The nature of how varied the data can be in this toolkit lends the system to need to not enforce structure on the data itself. There is not a thoughtful way to enforce “columns” or “required” parts of data that can not automatically be added by nature, such as a timestamp of when the data was taken or the order in which the data came in. Trying to mold some sort of shape for the data to be in will make it difficult to allow the toolkit to be truly widespread in its use, and more complicated to get up and running for various different studies. As such, we will not make any assumptions or enforcements when it comes to the structure of the data recorded in this toolkit. It is up to the researcher to utilize data cleanup to more thoroughly parse and structure the data in a way that is useful to them. This also helps the services mentioned above to be easier to use, containing less overhead.

17.4.3. Extensibility

In hand with the subscriber-publisher model, it is important that dependencies are not created in the telemetry system to aspects of the toolkit that are not vital to its core. That is to say, the telemetry system will make little assumptions about what data it will record, and what data it *can* record. Down the road, if a researcher wants to add some sort of new feature to the toolkit, they should easily be able to interject that functionality without worrying that it may interfere or otherwise not “fit” into the system in place. Each logical section of the toolkit should work as agnostically as possible, simply doing its part and passing along data for the telemetry system to take and record. This pillar is key in ensuring the toolkit is exactly that: a *tool*kit. We want to enable the researcher to do as they please, creating a more efficient work cycle.

17.4.4. Class Diagram

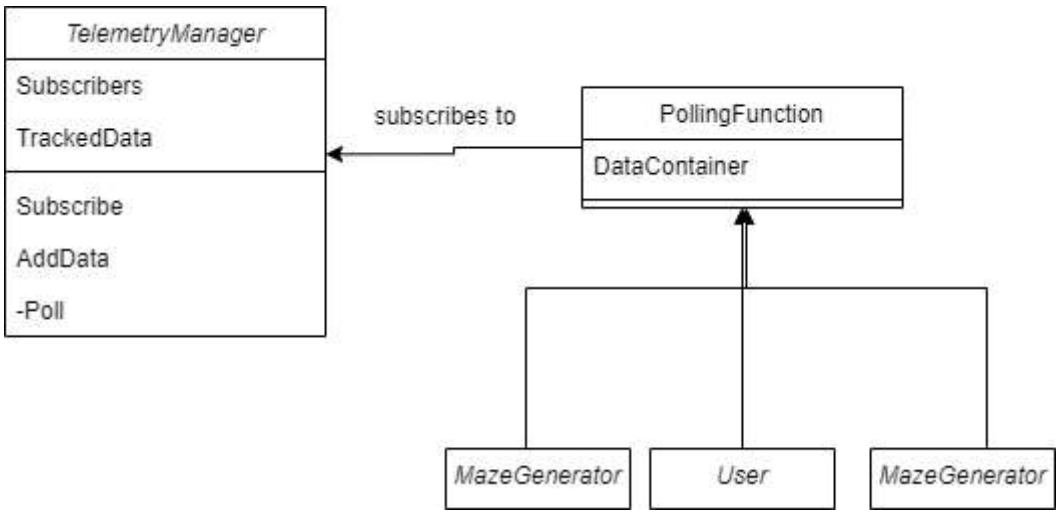


Fig 12. Class Diagram for the TelemetryManager

Made via LucidChart.com

17.5. Data Exporting

With all this data collected, it's important that we export it in a way that is both structured yet easy to use for researchers. This includes having the output be structured in a way that is easily navigable, with important info available at a glance. Furthermore, the data should be presented and structured in a fashion that most similarly lines up with how researchers normally run experiments and organize their own data.

17.5.1. Structure

A big part of our toolkit is organizing disk data in a way that is easy to use, with high fidelity to the structure in which researchers conduct their studies. Knowing this, we followed through with the format of our project and mirrored it into the structure of exported data from simulations.

First, each experiment is saved as a single, self-contained folder. All information including conditions and participant data relevant to that experiment are contained in the folder. This approach has two upsides: a) experiments are usually run in isolation for

research studies, and b) experiments can easily be shared / moved. On the first note, it's very common for an experiment to be made in a research study, which plans to tackle a single problem or question. As such, the use of this single folder encapsulates that question in isolation so that it can be analyzed and iterated on without being confused with other experiments. This also incentivises the use of structured conditions in the experiment, since wrongly spreading related conditions between experiments becomes very cumbersome. For the second upside, experiments being self contained means that researchers can easily share them with other researchers, or individually move them across workstations as they see fit.

To move on, each experiment then contains its conditions. A condition represents one configuration of a simulation. Thus, an experiment is made up of numerous configurations of a simulation. This allows researchers to compare conditions easily and use the exported data to come to conclusions on how the configuration of the conditions impacted the final results. The parameters for each condition are saved to a CSV, which can be viewed to ensure that the correct configuration is saved for that condition. In each experiment, we store a folder for each participant that is run through one of these conditions.

Participant folders are further structured by the individual runs they performed. A run consists of a condition present in the chosen experiment. Thus, for example, if one participant is run through the same experiment five times, they will have a folder that contains 5 run folders.

Finally, each run folder contains the relevant information for that specific run. The folder is marked with the experiment that was run, along with a timestamp. Then, the folder contains all the exported data from that run. This includes the data from the telemetry manager, along with an exported version of the maze generated in that run.

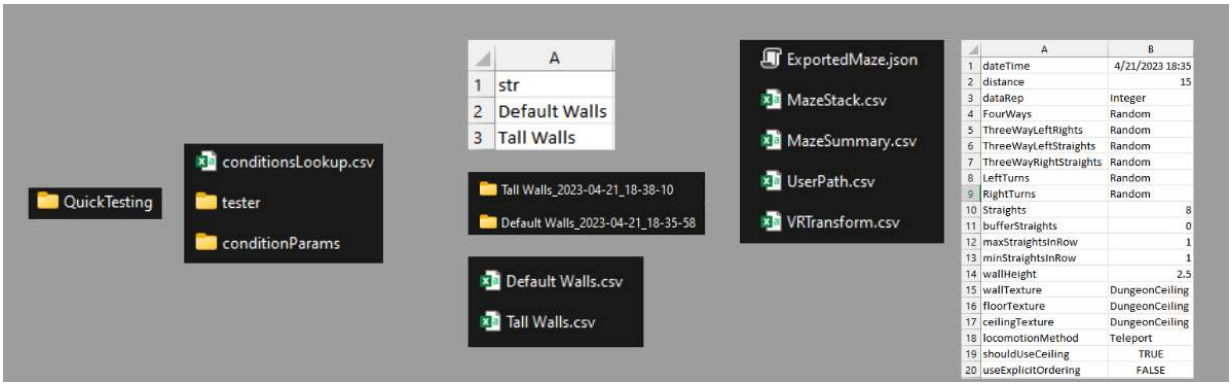


Fig 14. The structure of a sample experiment

17.5.2. Format

There are many common data formats that exist in modern computing. These include XML, JSON, TXT, and even binary. Each different format has its use case, drawback, and reason to be used. However, we chose to export our data from the telemetry manager to CSV.

One reason for choosing CSV is that it is already a very popular choice by researchers. The advanced capabilities of programs that can read CSV lend themselves well to organizing, graphing, and being computed over. Furthermore, CSV is a very simple data format, which allows researchers to create custom scripts that can easily read in this data as they need.

Another reason for choosing CSV is that it plays well with our use of reflection for taking the data classes and writing them to disk. Since objects already come as member-value pairs in C#, they can easily be written to CSV using reflection. Thus, we minimize the need for further processing to get the data objects into the desired output format used by the telemetry system.

17.5.3. Writing To Disk

Given our choice to allow captured data to be as agnostic as possible, a question arises: how do we save this custom data? Usually, data classes go through a “serialization” pipeline that converts the object in memory into something meaningful that can be written to disk. However, we did not want to make researchers develop this pipeline for each type of custom data that they choose to capture.

To combat this problem, we rely on the use of reflection in C# for writing our objects to and from disk. This allows researchers to make simple data classes that are filled out with whatever relevant info they want, and then we can read the properties of those data classes directly and write them to disk. Furthermore, researchers can still themselves use these data classes in-engine for various other uses. For instance, we provide a live researcher dashboard that has live statistics when a user is going through

a simulation. The same data that is stored in the telemetry manager for export after the maze is done can be used to populate and keep these live statistics up-to-date.

17.5.3.1. C# Reflection

Since we fully rely on C# reflection to allow researchers to use any agnostic data class for their telemetry, it's important to discuss what reflection is. “Reflection” in C# provides information about various pieces of code in a project. For our use case, we specifically utilized reflection to read the members of a data class. Reflection allows us to get a list of all members present in a class. Thus, we can take these members, get their values, and write them to disk. Furthermore, since we are writing data directly from the object to disk, we can go in reverse and *read* data from disk into an object - essentially loading it back in. Again, reflection is used to match up each piece of data on disk to the existing members on the type of class provided. In our toolkit, this is used for various files containing info for things like session info or data about researcher experiments. This expanded use of reflection out to project-specific implementations shows how powerful reflection is, and why it served as the best choice for serialization in our project.

18. Design Choices

18.1. Game Engine

In choosing what engine to use when developing this toolkit, there are 2 major choices: Unreal Engine or Unity. Both of these engines have their own caveats, advantages, and disadvantages. Both engines are fine choices and both have their validity. In the end, we chose Unity as our weapon of choice to build the toolkit. There are 3 main reasons we chose Unity over Unreal Engine:

1. VR Support
2. Documentation
3. Use Case

18.1.1. VR Support

Since Day 1, Unity has jumped on the VR train with enthusiasm. They built several in-house solutions and tools for development in VR, moving to the forefront of becoming a development tool for making VR content. Due to the relative infancy of VR,

this puts Unity at a significant advantage over Unreal Engine. In general, it is much easier to work VR

18.1.2. Documentation

Unity is known for its concise yet knowledge documentation on the engine. Almost every corner of the engine has some sort of example, explanation, or use case to give useful insight to developers. This is especially important to a team that is learning the framework along the development of this project, all in a relatively short time frame. Furthermore, there are many tutorials and forums online surrounding the creation of content in Unity engine. Since Unity is usually the engine to turn to when it comes to prototyping and “whiteboxing”, this in turn creates a lot more diversity in what developers use the engine for. We plan on utilizing this diversity to aid in our development of the toolkit.

18.1.3. Use Case

In the battle of Unity vs. Unreal Engine Unity is usually seen as a more “beginner” engine whereas Unreal Engine is more “industry standard”. There are many aspects of each engine that follow these labels. For instance, Unity utilizes C# for its scripts, which is a much higher-level code as compared to Unreal Engine’s use of C++. Furthermore, Unity has deep integrations into language-specific tools to reduce the conflict between writing C# code and writing Unity code. Unreal Engine, on the flip side, sports more “strict” architecture when it comes to development of assets and code in the engine. Everything is aimed towards making a refined, high-fidelity environment, which can get in the way of development. Since we are just making a toolkit that is functional rather than flashy, Unity seems to be the correct choice for our engine.

18.2. Virtual Reality API

The API we chose to use when developing this toolkit is SteamVR. This is a high-level API developed over OpenVR, providing lots of boilerplate functionality right out of the box. This includes virtual hands, movement techniques, a VR camera, and more. Furthermore, SteamVR supports almost all major VR headsets. This allows us to create the toolkit without narrowing the headsets that can be used with it. Again, our

strive is to make the toolkit as widespread in use as possible. This headset-agnostic API helps us to achieve this goal even further.

18.3. Version Control

There are many modern choices for version control. However, game engines create codebases that can be quite finicky to work with. Firstly, it's not uncommon for data to be in formats not built well to work with version control, such as binary data. Furthermore, graphical elements such as scenes represented in data are very difficult to resolve merge conflicts. Lastly, the version control needs to be able to support large files such as textures, models, and scenes.

With all these considerations in mind, we chose to use PlasticSCM as our version control. This tool is built specifically to work with Unity, giving a highly integrated developer experience. The Unity editor has a built-in interface to view and create changesets, manage branches, and more. Furthemore, the centralized model of PlasticSCM lends itself very well to the content we are creating for this toolkit. We want to minimize conflicts / branch issues that could come along with version control such as Git.

18.4. Why Unity

When setting out to create a VR application, the choice of engine is crucial in what the application will be able to do and where it will be able to be used. The main contenders for a VR application that we considered were Unity, Unreal Engine 4 (UE4), and Godot. There were a few considerations to make when choosing an engine, and those were:

- Ease of use for development.
    - How applicable is it to our use case?
    - How is the learning curve for those on the team who haven’t used it?
- Ease of usage for users.
    - How likely is the user to be familiar with the engine?
    - How easy is it for the user to use our tool?
- How well it supports VR.

With these factors in mind, we can take a look at the three main options, Unity, Unreal Engine 4, and Godot.

18.4.1. Learning Curve

For the learning curve, Unity is the most forgiving. Its core ideas surround game objects and scripts that are used to support complex functions. Because of the simplicity of Unity, it makes it relatively easy to pick up. It's scripting language, C#, is also relatively simple to pick up if one knows any other object oriented programming language. Unity supports VR applications quite well, so using it to create our tool would work adequately.

Contrasting this learning curve is Unreal Engine 4, which boasts a much more demanding learning curve. This is due to the number of features that Unreal Engine 4 has which make it difficult to build even simple games. Its language of choice is also C++, which is known for being a difficult language to use effectively. Unreal Engine 4 allows for VR support, so creating our tool with this engine would work fine.

Finally, Godot is a bit simpler than the other two. It offers less features than Unreal Engine 4 or Unity, but it is still capable of creating complex games. Its scripting language, GDScript, was created with readability and ease of learning in mind, so its overall learning curve is the most forgiving of the three. VR in Godot is supported, so using it to create our tool would work just fine.

18.4.2. Target Audience

The largest factor in our choice came down to the ubiquity that Unity has in the research community. We expect that most VR researchers are familiar with Unity and therefore would have a better experience using a tool that was built in Unity. Likewise, the more familiar the audience is with the engine used to create our tool, the more likely it is that they will be able to adjust things on their own, to better suit their purposes. Overall, we want to create a useful tool, and thus building it with an engine that is most likely to be understood by the majority of our audience makes sense in reaching our goals.

19. UI Design

The User Interface of an application is generally the first and last thing a user is met with. It is arguably the most important component of any software, and yet simultaneously the least recognized when implemented well. I believe this lack of awareness is actually what separates good user interface design from bad user interface design, as people will only start to really analyze a given user interface when things stop making sense to them or the user interface is interfering with the rest of the application. However, before we dig into what separates a good and bad user interface let's highlight at a high level some of the unique features we will want from our own user interface.

19.1. Breaking Down or User Interface Requirements

Let's begin by breaking down our more unique features and then follow it up with the more common features that you would associate with user interface design.

19.1.1. Separate User Interfaces -

Firstly, we want some semblance of two separate user interfaces that being, one for the participant, and one for the researcher. These should be thought of as separate for numerous reasons that we can highlight later, but the most obvious distinction between the two is that the participant will be viewing the maze from a three-dimensional perspective with a virtual reality headset, and the researcher will be seeing a two-dimensional perspective from a classic computer monitor.

19.1.2. Communication Through Separate User Interfaces -

Although the two user interfaces will be separate and distinct, they should have the capability to relay information to each other, either directly or through our virtual world representation of the maze. More specifically, the researcher should be able to trigger live-action events that are then reflected in the participant's view.

19.1.3. Main Menu

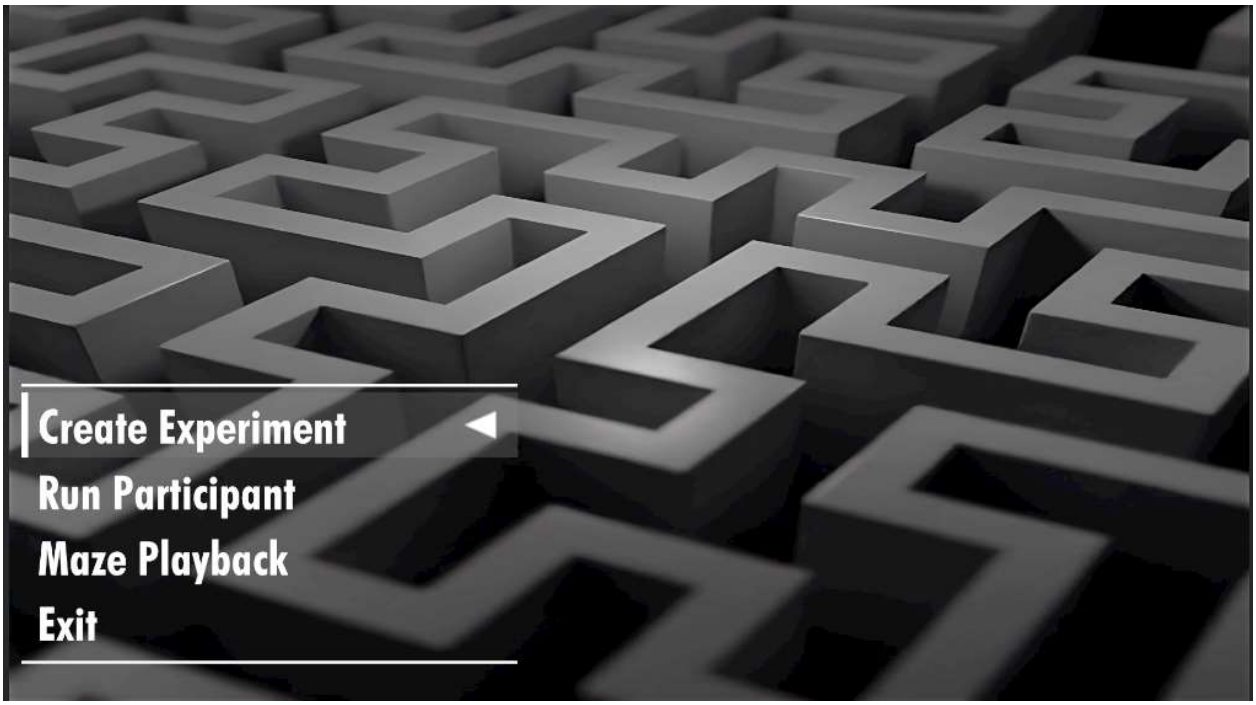


Image showing the final design of the main menu screen.

The goal of this main menu screen is to provide a simplistic interface upon entry to the first screen of the program. The selectable options on the bottom left of the screen enable the researcher to quickly move on to another UI screen depending on their needs. The buttons are also designed to display a small change upon hovering to give feedback to the user on which button is currently selected.

The current main options for aVRage paths is creating an experiment, running a participant through a built experiment and maze play. These options will be broken down below to give a workflow on how the application should be used.

19.1.4. Create Experiment

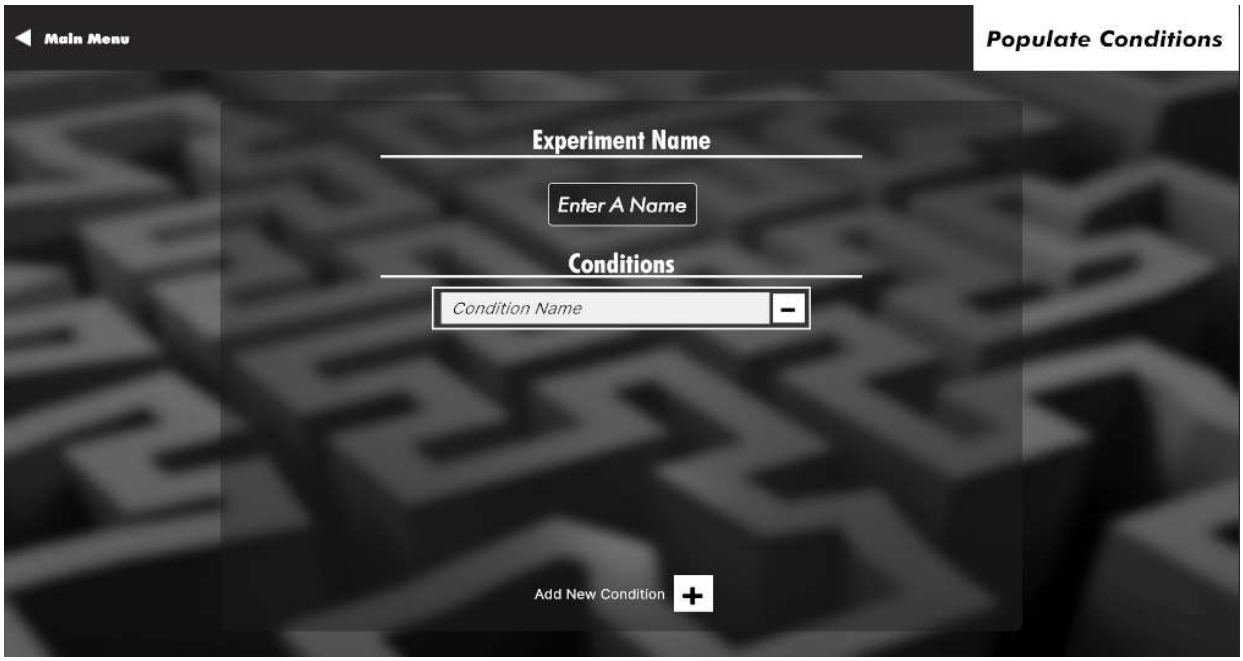
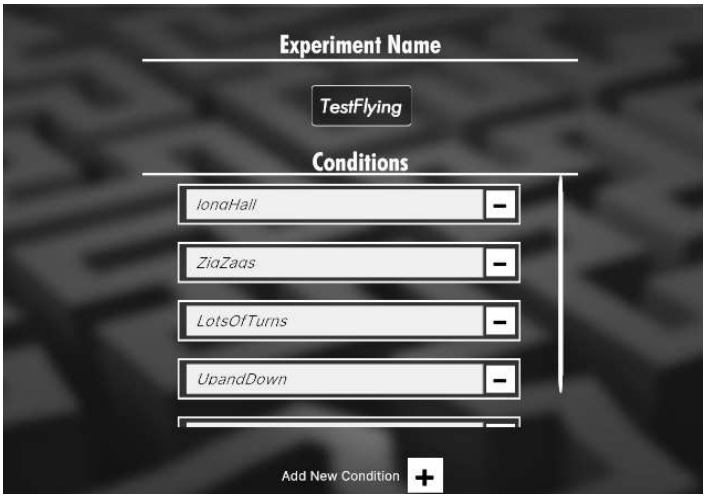


Image showing the final design of the create experiment screen.

The create experiment screen is where the researcher can name their experiment and provide any number of conditions for their experiment (any number greater than 0 as the aVRage paths team did not want the researcher creating empty experiments). Some key buttons exist on this screen, a return to main menu screen (in case the researcher misclicked or simply changed their mind), an Add New Condition button (to add a condition to the list for the experiment), and a populate conditions button (which finalizes the current experiment and takes you to the next screen where you will actually customize the given conditions).



19.1.5. Populate Conditions

Fig. A valid and completed Experiment showing the experiment is ready to save.

The following UI screens are part of the Populate Conditions screen that originates from the “create experiment” UI, the purpose of these screens is to allow the researcher to specify exactly how they want their maze to play out by changing parameters configurations to meet these goals. This area of the UI consists of three main screens: Path, Visual, User.

These three screens have their own distinct parameters to modify but also have common UI elements that persist throughout each screen to allow the researcher to know the current state of the experiment, confirming conditions, the tab navigation to switch to any tab, and returning to the main menu.

The first and most important element that persists through the three screens is the sidebar that shows the current experiment and its name, as well as the current conditions for those experiments. The UI for this side panel allows for the researcher to click through each condition, which if they have previously entered values then they will be stored so the researcher can come back and edit them before saving the entire experiment. Each condition is clickable and the UI also provides a white bar on a

condition to indicate which condition the researcher is currently editing. Each of these conditions come accompanied by a checkmark that starts off red, and upon clicking a confirm condition button will turn green to indicate that the condition is finalized as an easy reminder for the researcher. This side panel also features the previously talked about confirm conditions button that allows for the researcher to confirm conditions; the important thing about this button is entering invalid inputs turns this button red and disables it so the researcher knows that they entered invalid constraints on their maze and the aVRage paths team will not allow them to save such a condition.

Next element that persists is the navigation bars on the bottom and on the top. The navigation bar on the top allows for the researcher to know which tab they are currently on, and allows for the researcher to quickly switch between tabs to edit parameters in certain categories (such as textures being under visuals, etc.) This top bar also features a return to the main menu button in case the researcher ever needs to return. Lastly we have the bottom bar which just provides the researcher with

19.1.6. Path tab

Screen showing Path tab upon entry in populate conditions UI screen.

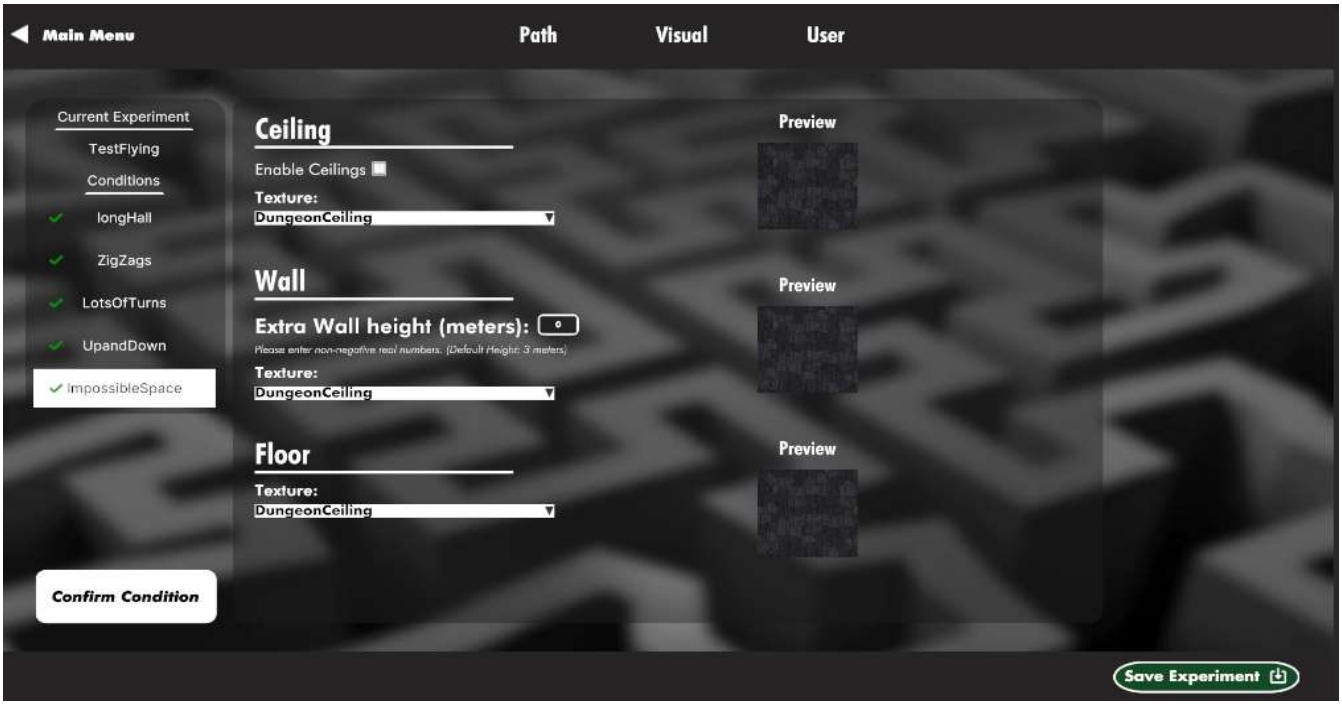
The path screen is the area that shows a box in the center containing segments and junctions, and the box on right containing desired distance and current used distance. The main goal behind this Tab is to contain all information regarding the path elements that generate the maze. As such, by looking at the figure provided above, the screen consists of Segment pieces, junction pieces, an entry box for desired distance input, and if they want the path to be explicit or not.

This screen features a number of small UI conveniences such as when the user checks the input box it will autofill the corresponding junction/segment with the default value for that entry box, or vice versa such as clicking random for that box will erase the input in the box. The input box also turns red to indicate that an invalid input was entered and will block the researcher from confirming the condition. Another important UX functionality is summing up the number of segments/junctions to display on the right to show the researcher how much of their allotted distance they have used up, and turns red if they go over.

As aVRage Paths has to have at least one segment between every junction the UI had to be modified to handle this limitation (explained later in this document as to why this limitation had to be implemented due to visibility issues). The segment box is always set to at least half of the distance as that is the lower bound on how many segments a maze can have, and the upper bound is the distance (if the researcher wants a straight hallway). As such, the number of segments is also influencing the minimum/maximum number of segments in a row, and the UI will update those values upon the segment number changing. The minimum number of segments in a row has a lower bound of 1, and an upper bound of (current segments / number of junctions) to prevent mazes where these constraints are possible. The maximum number of segments in a row has a lower bound of (minimum straights in a row) and an upper bound of (current distance - (current junctions * minimum straights in a row)) as this formula gives us the number of leftover straights the researcher has from the given inputs.

In the future, there should be a mini navigation bar added below the top navigation bar to allow for navigation between different sets of path pieces, such as 3D pieces, or unconventional pieces such as circular pieces, wavy pieces, or zigzaggy pieces.

19.1.7. Visual tab

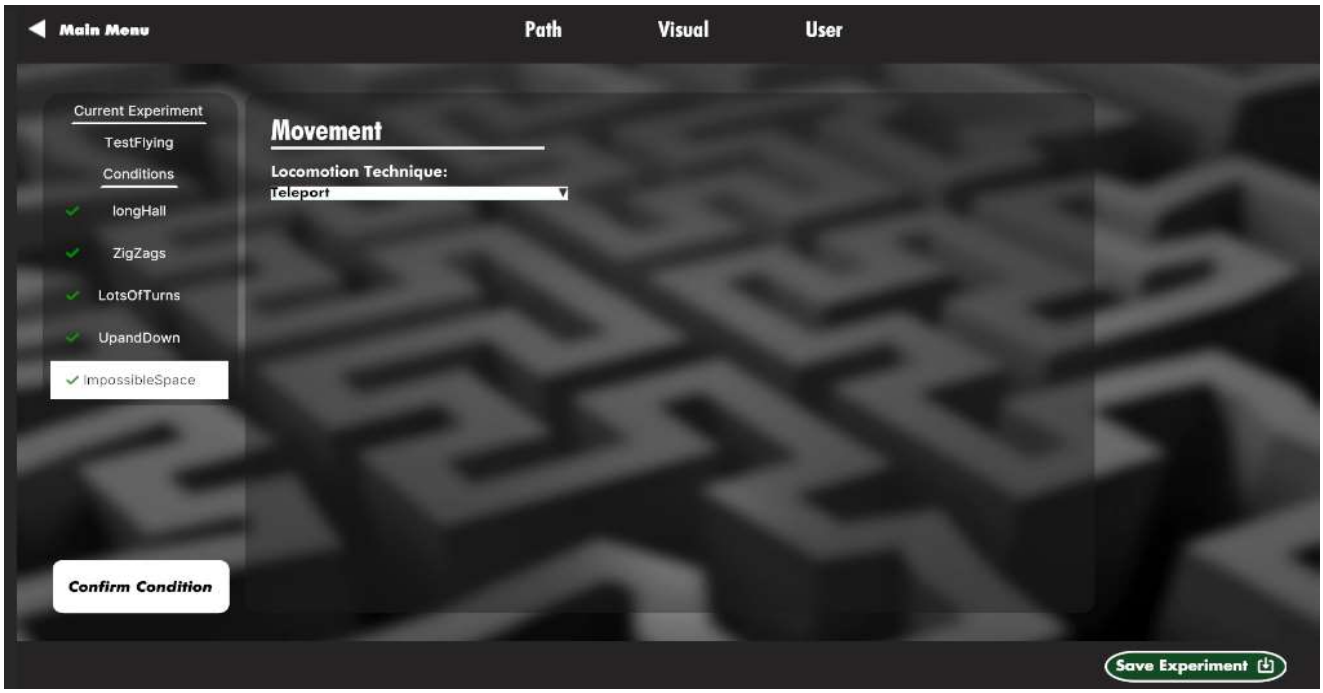


19.1.8. Screen showing the Visual Tab.

The visual tab consists of elements that the participant will be seeing in regards to the maze. In this tab the researcher will be allowed to customize Ceiling, Wall, Floor textures. The researcher will be allowed to drag and drop their own custom textures into the project folders and upon loading in the aVRage paths program the UI will automatically pull and give an option to select as well as a preview of those textures. This allows for researchers to easily bring in their own textures for their own experiments without having to make changes to the UI itself.

In this Tab we also allow for two things currently that adjust the maze pieces prefab. The first is the option to enable ceilings which will attach a ceiling to every maze prefab, including the start and end area prefabs. The next one is adding extra wall height to the pieces in case the researcher wants taller walls in cases where the participant could be experimented on things such as tall vehicles or flying, etc.

19.1.9. User settings



Screen showing the User settings tab.

This screen displays all the customizable settings regarding the user. The project currently only changes locomotion methods for the user and as such the UI only has this as a changeable parameter. The locomotion dropdown pulls all options from the locomotion object which researchers can drag and drop in the Unity Builder, and the UI will update to display that new locomotion technique and pass that on.

19.1.10. Maze Playback Screen

- Initial import screen that allows a researcher to import a previously simulated maze runthrough
- Upon successful import, a confirmation button that changes scenes to load the maze
- The maze shows just how it would if it was being generated, highlighting pieces as impossible and having their opacity controlled by the current piece that is stepped on
- A main menu button in the top left so that researchers can navigate out of the maze playback

- An “autoplay” box that traverses the player between pieces at a constant rate
- A slider that the researcher can use to scrub through pieces
- Forward and back arrows on the slider to move piece-by-piece

19.1.11. Confirm Modal

- Modal used to confirm a selection made by the researcher
- Used when going back to main menu from creating an experiment, and when aborting an experiment
- Confirm button “yes” takes functionality of the button that is supposed to trigger the modal
- “No” button closes the modal and nothing happens
- Nothing other than the modal should be intractable while it is visible

19.2. Screen Design Visual Drafts -

Now with all of the requirements flushed out, we can start making visual drafts of how we want our more complicated components to look. Doing this draft will allow us to perform easy iterations on designs as we can see if things are going to get too cluttered or otherwise confusing before we put all the time into implementing these screens. Furthermore, we can then submit our visual drafts to peers in order to get some sort of usability testing further adding a layer to our test framework to ensure good design.

The real meat of this screen design comes in the form of how we organize our maze parameters in a way that makes sense. As we have previously stated, we expect our input parameters to come in levels; in which defining each level raises a new level of parameters, and these levels can be left un-filled to default to randomization or our mazes default parameters. One major aspect of this design that needs to be considered is the structure. We’re talking about having access to a lot of different variables, and this could be incredibly overwhelming for someone just starting to use our app. So the goal is to mask the number of variables until they need them using a drop down menu of drop down menus. To demonstrate this, we will provide a very rough sample of what our menu might look like when the user decides to not configure any parameters and compare this to what the menu might look like when they go all the way down to the most explicit level of defining parameters.
In figure 14 we can see what our maze parameters might look like when they’re fully filled. This is far more overwhelming than seeing the information presented in figure 15 Note that each level we drop down to contains another round of option parameters. To

be more specific, we can't define a distribution of the junctions we want until we provide how many junction total we want. Furthermore, you can't specify the distribution of directions of the two

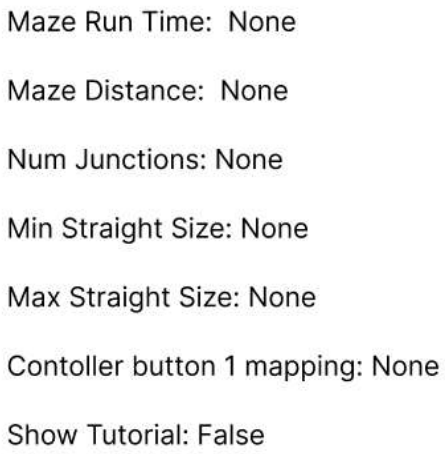


Fig 13. Image showing parameters left as default.

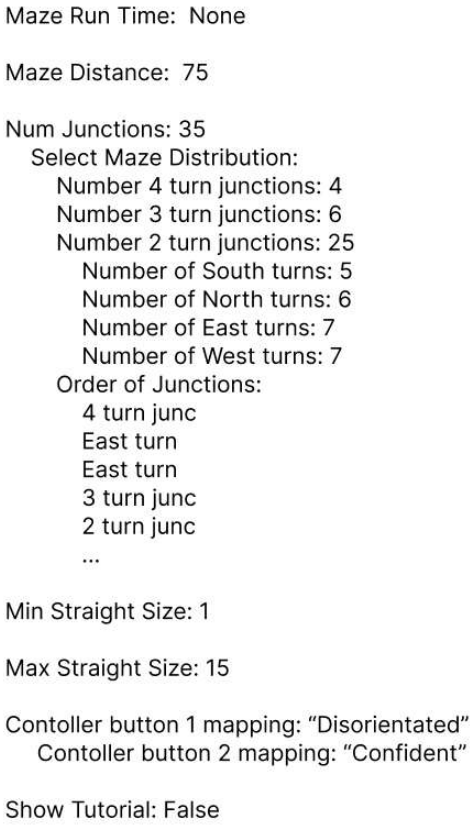


Fig 14. Image showing parameters fully filled

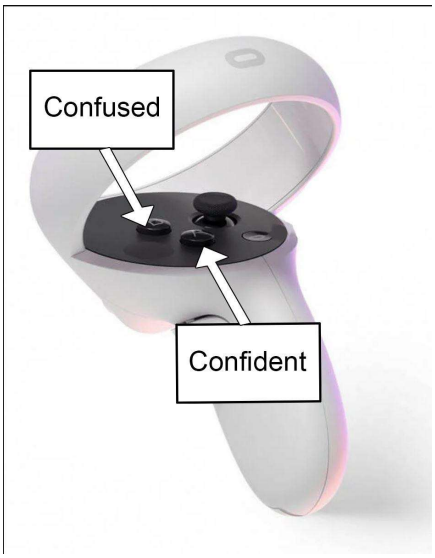


Fig 15. Image showing handset labeled with researcher specified mapping.

junctions turn until you specify how many two junction turns you want to begin with. And finally, you can't specify the order the junction will generate in until you specify the distribution of the junctions themselves. Finally, you can see with specifying the controller button mapping that the next level won't appear until the first is defined. This parameter hiding makes for a more intuitive design thats less likely to overwhelm someone new to our app.

Now with our most complicated menu flushed out, really the only component of the User Interface that is still hard to visualize is participants' controlled labeled with the researcher-specified mapping. Figure 15 shows what the labeled control mapping might resemble in our final application. This is important design as controls are easily forgotten when they don't actual correspond to any tangible in game actions. By providing the mapping for the user to see in game, it will help to take strain from the researcher having to interrupt the trial to remind the user of controls.

19.2.1. Implementing the Hierarchy for the User Interface as a Whole

As we have stated previously, at our core, our project is a testing framework for virtual reality locomotion. This means that the user interface should be as easily iterable as possible. This is a difficult challenge as there's so much that goes into coding a user interface, however, we must just look at this through the lens of someone who knows nothing about the application and try our best to cater to this demographic. One of the most fundamental design choices for a user interface to be easily iterable is the user interface components themselves should be completely separated from the game logic that controls them.

If this design principle is not upheld and graphical components become tightly coupled with logical components, then it becomes a nightmare just to make even the smallest of changes. However, when this principle is upheld, the user interface essentially just becomes graphical building blocks that you can use and reuse anywhere in your code. Say, for instance, you wanted to change when something like a hint pops up for a user. If the code for the detection is within the user interface component itself then this is no longer a simple task. However, if the component had a function such as displayHint that triggers it in the game world, then you could simply delete the existing call and move it to where you desire, making for a seamless iteration.

This notion of reducing coupling feeds directly into our next point, which is unifying the locations the user interface end-points are called from. Taking our previous example of changing the timing of when a hint is displayed, a user would much rather have a go-to place to see all user interface event handling as opposed to having to track down every instance of a function being called throughout an entire project that could be hundreds of files. This can be done fairly easily through the addition of a UI Event Listener Script (Yin, 2020, p. 222). Following this methodology, each scene that has a User Interface should have its own User Interface Eventhandler, taking this even further, this event handler should either have a separate set of functions for research and participant views or should just use two separate event handlers altogether.

I believe that the use of one event handler per scene probably makes the most sense in development. Separating the functions to be researcher user interface functions, participant user interface functions, and a shared set of functions will give us the highest possibility for reuse (as we would have a shared set of functions for items that overlap). Furthermore, this setup would minimize the number of files a developer would need to look at in order to make changes, overall improving the understandability of the code as a whole while speeding up the development process.

Now with an idea of what we want to do, lets make a class diagram and then discuss the inner workings of our diagram in more detail.

19.2.1.1. The User Interface class diagram -

This class diagram will represent the framework that we can apply to every scene, but it requires further explanation to understand how this truly creates a modular workspace.

Let's first begin by explaining that in Unity's game engine, everything that is in the game world has to be a game object, and every game object inherits from Unity's MonoBehaviour class. Furthermore, Unity already provides us with classes for Text, Drop down menus, File explorer windows, panels, canvases, and Buttons; we simply just need to define behaviors for these objects.

This level of defining behavior for these pre-made objects is where our modularity shines. But before we can see this, we must understand which elements of this class diagram persist between scenes (the various menus/stages of our application), and what information changes. There will be only one copy of the UI manager across our entire application, this UI Manager will be able to survive between scene transitions and reloads, this manager will then get references to the researcher and participant canvases for each scene it's a part of. This allows us to make the setup for every scene the same, but still gives us crazy amounts of customizability because each screen will have a canvas object unique to it. And although the organization of these canvas objects will be slightly different across scenes the overall premise of the canvas will be the same. The canvas will simply just be fed a list of panels and that will construct what we are seeing. These panels give us the most reusable freedom across scenes as we can simply re-use the panels and simply add them anywhere on the canvas.

Furthermore, since we are going down to such a small level of panels, it leads to fast and easy changes from someone that only has a general understanding of the application as a whole. This is because instead of trying to add a random component to the canvas and being forced to resize everything or move things around because of awkward scaling across different screen resolutions; the user would simply be able to add whatever they want to one of our pre-made panels and then just inject it into the User Interface, thus letting us handle all of the resizing shenanigans.

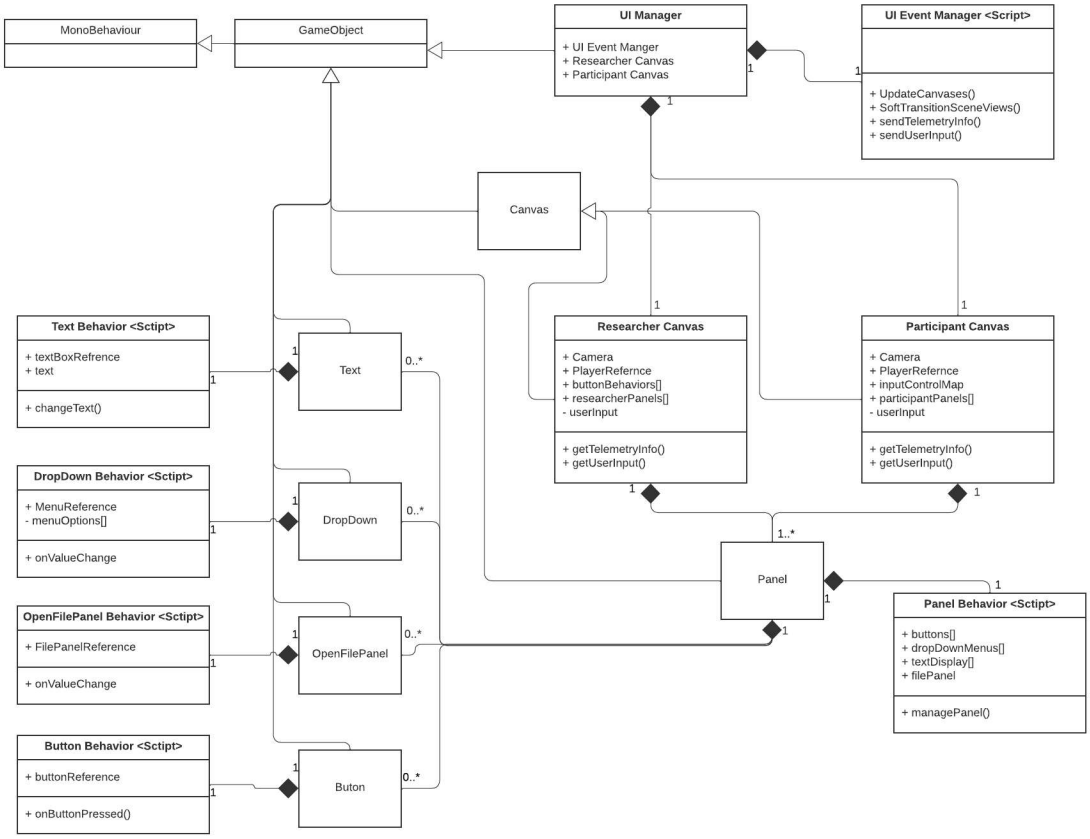


Fig 16. UML User Interface Class Diagram

Made via LucidChart.com

20. Camera Movement

An emerging issue with VR headsets is motion sickness for those using the headset. Motion sickness can occur in a vary of different ways, producing symptoms such as:

Table 1. Signs and Symptoms of Motion Sickness

Severity	*Signs*	*Symptoms*
Mild	Belching	Stomach awareness
	Yawning	Malaise
	Facial and perioral pallor	Headache
	Heartburn	Irritability
	Hypersalivation	Drowsiness
	Urinary frequency	Fatigue
Moderate	Cold diaphoresis	Nausea
	Flushing	Nonvertiginous dizziness
	Increased body warmth	Apathy
	Hyperventilation	Depression
	Vomiting	Disinterest in social activities
		Disinclination for work
		Decreased cognitive performance
		Exaggerated sense of motion
		Increased postural sway
Severe	Inability to walk	Social isolation
	Incapacitation	
	Loss of postural stability	
	Persistent retching	

NOTE: *Signs and symptoms are listed in decreasing order of prevalence.*

Information from references 1 and 2.

Fig 17. Types of motion sickness

As seen above, motion sickness can affect many different people in many different ways. The goal of the team is to prevent or at the very least mitigate the cause of these symptoms as much as possible. To accomplish such a task, the team must first understand what causes motion sickness to occur.

Motion sickness is not completely understood but current research indicates that most instances of motion sickness occur when there is a disconnect between visual, vestibular, and other senses between one another. In simpler terms, when an individual

experiences changes to one's senses while others seem to remain the same there is a disconnect that causes motion sickness. Due to the nature of this project experiment, this is something that will occur during the research and as such it is the job of the programmers to try and provide tools for the researcher to research or prevent such events from occurring.

Designing different methods of movement is one way the team will aim to provide researchers with this VR tool. The main issue with VR is how participants experience movement through their eyes but their vestibular sense indicates that there is currently no movement occurring in the body; The disconnect between the eyes and inner ear balance is the exact scenario that causes motion sickness to occur in participants. The solution to alleviate such an issue is to parameterize player motion by providing options such as walking at a slow speed, teleporting, and more.

Through a variety of options for locomotion, the researcher will be able to customize the experience to fit the current research topic. The most basic implementation of locomotion for the tool will be walking, which the default speed will be 1.4 m/s (a speed research has found is most similar to human walking speed allowing for mitigation of motion sickness). The speed of walking will also be customizable to allow for further research into different speeds and how participants are affected, as well as allowing for options in which the researcher wishes to perform the experiment on an omnidirectional treadmill. Another addition that the team plans to include is running, this would allow participants the power to switch from a walk to a sprint at a push of a button.

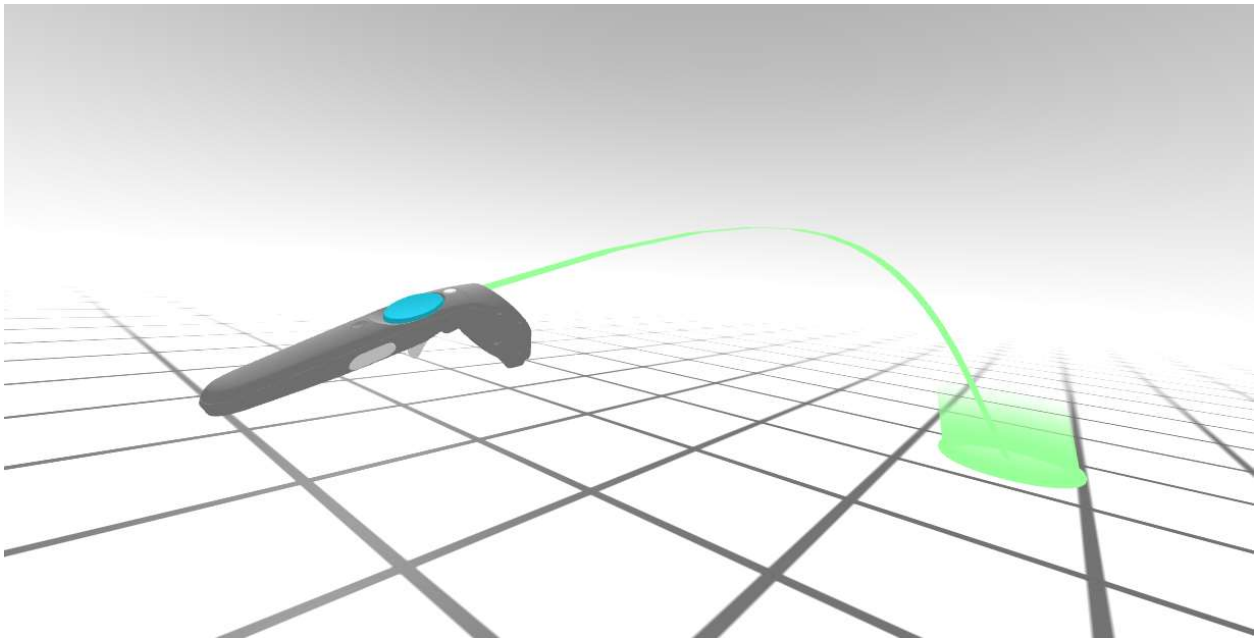


Fig 18. Teleportation movement style

Another form of locomotion that has been seen to be effective at reducing motion sickness is warping/teleportation. The way this will work is the player is provided with a pointer that will display on the ground in the direction they are aiming and clicking a button will teleport the player to the location. The purpose of this approach is to enhance the disconnect between the virtual experience and reality such that it will not affect your senses, i.e. no “walking” movement is being performed while the player’s body is at a stand still. This approach is not only interesting from a motion sickness perspective but also could explore if this form of locomotion helps reduce virtual reality fatigue from long sessions.

The last form of manual locomotion would be a “pinch-n-drag” movement. The way this works is the participant will use the trigger on the remote which will grab the current location of the map, then with movement from the controller will drag the player “across the ground”. Not a widely used form of movement as it can get tiring very fast across distances but could be an interesting option for future research. A handy option for this type of movement could be for zooming in and out of things.

Player movement around the level is not the only thing that concerns the team when it comes to motion sickness, but the camera movement as well. Camera movement is very important as there is research showing that incorrect implementation can cause a disconnect and create an unpleasant experience for the user. There are two things that need to be addressed when it comes to designing the player camera in the engine: camera movement in the world, and user controls for camera movement.

The first issue is solving camera movement in the level/world. There are two prevalent types of camera movements in industry, 3DoF and 6DoF. DoF stands for degrees of freedom which refers to how many directional axes are being used in tracking. 3DoF refers to tracking the pitch, yaw, and roll of the headset (mainly tracks where the user is looking such as up, down, left, right, head tilts), whereas 6DoF includes the same but also tracks the headset’s position (x, y, z coordinates of a user’s non-virtual space) using displacement based on the headset’s original position.

There are advantages and disadvantages to both approaches. The advantages of 3DoF is that it enables users with small areas to be able to use VR, programmers to create simple quick apps for basic tasks (such as 3D video viewing), or allows for users without expensive headsets to be able to experience VR . The disadvantages of the

3DoF is that many of the programs for it feel very “flat” or “2D-ish” which means any movement that occurs in space is not tracked and can quickly cause motion sickness. The following example shows how when a user leans forward what would happen in 3DoF vs 6DoF:

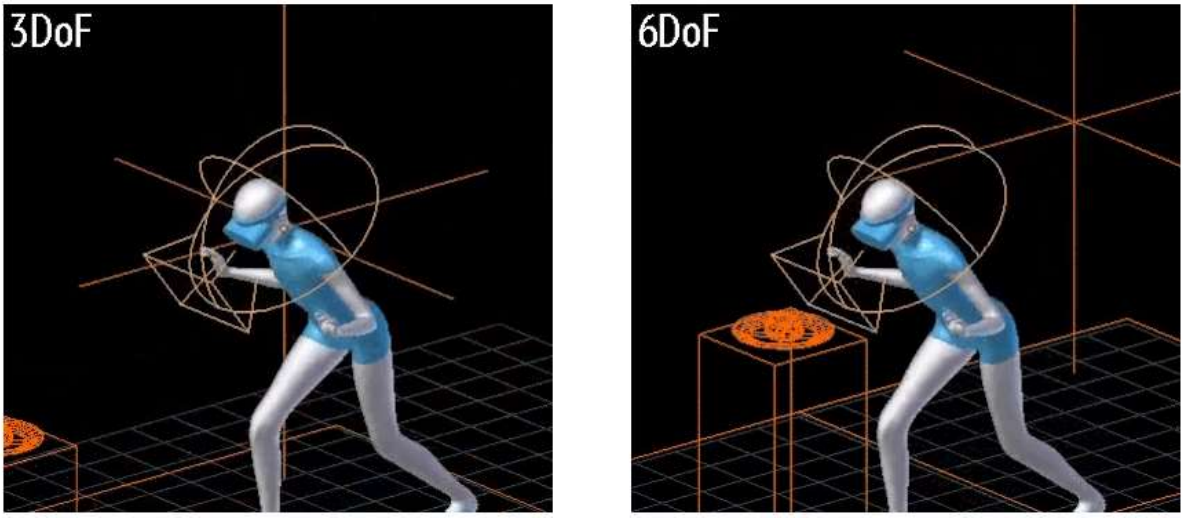


Fig 19. Leaning with 3DoF (left) vs Leaning with 6DoF (right)

Due to the issue of motion sickness, the application will be designed with 6DoF and will require researchers who wish to use this tool to use the appropriate headset. The reasoning behind this decision is due to the fact that much of the research that goes on in the VR field already occurs in a planned space with lots of options of movement for the participant and the latest higher end headset. As such, these factors allow the team to not have to spend time developing a 3DoF version as well as 6DoF version, allowing the team to focus on other prospects.

The next major issue is camera turning, or how the player looks around the world. The reason this is an issue is two main reasons: certain VR headsets have long cables and turning frequently can cause motion sickness.

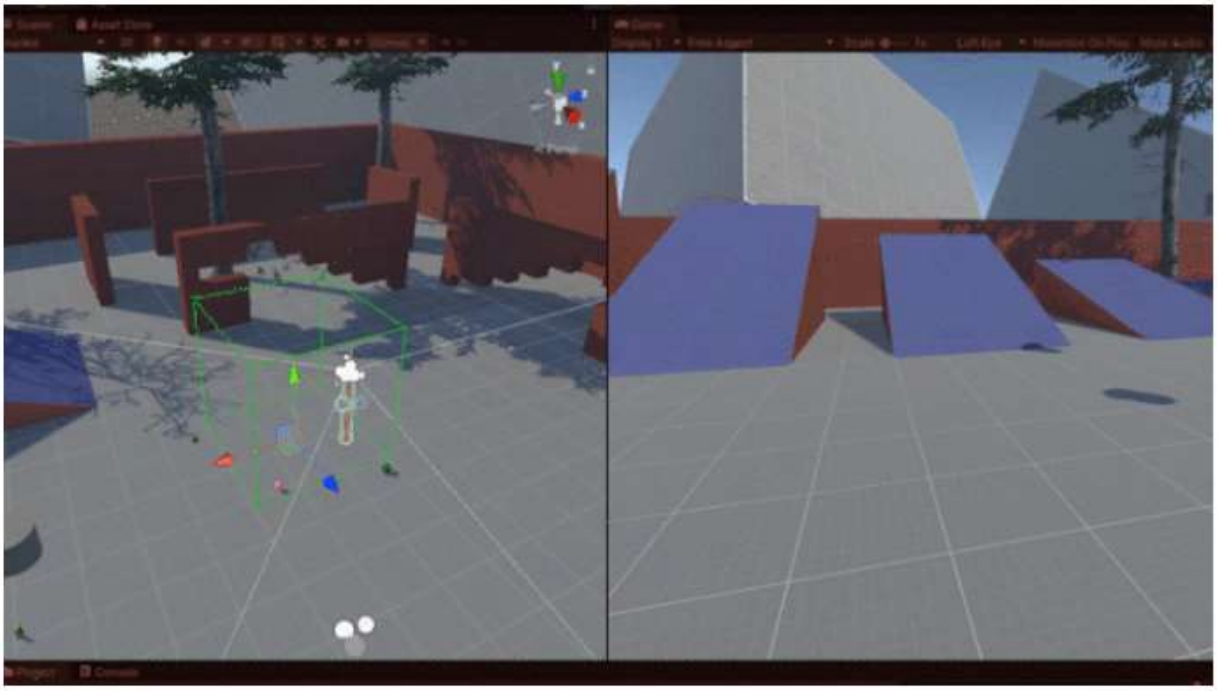
The first reason has to do mainly with older VR headset models as they tended to have long cables attached to the headset. Requiring users to turn around in the real world to turn around in game can quickly cause them to get tangled in wires and accidentally even trip and fall on such wires. Although new headsets require no cables they can be expensive and as such not all research laboratories can afford such an expense. The goal here is to create a solution that alleviates this headache of requiring turning or rotations in reality to avoid this cable tangling.

The next reason is requiring turning can quickly cause motion sickness in participants. This will become very evident in this specific VR program as there are

going to be a lot of generated twists and turns in the maze that a participant must navigate through. Requiring the user to turn their heads manually repeatedly over a short period of time will quickly create a disconnect in their senses leading to motion sickness.

There are many ideas behind how to solve these issues, such as snap turning vs smooth head turning. Implementing various turn mechanics and allowing the researcher/participant to pick would be helpful.

The idea behind snap turning is that rather than smooth head turning like one does in reality, clicking a button will “snap” the camera to a different angle as a method of turning. This solves the above two issues by allowing the user simply to click a button to turn the camera. This creates a major disconnect between the senses but in a good way as the player is not really experiencing movement but rather is experiencing something akin to a video or slideshow transition.



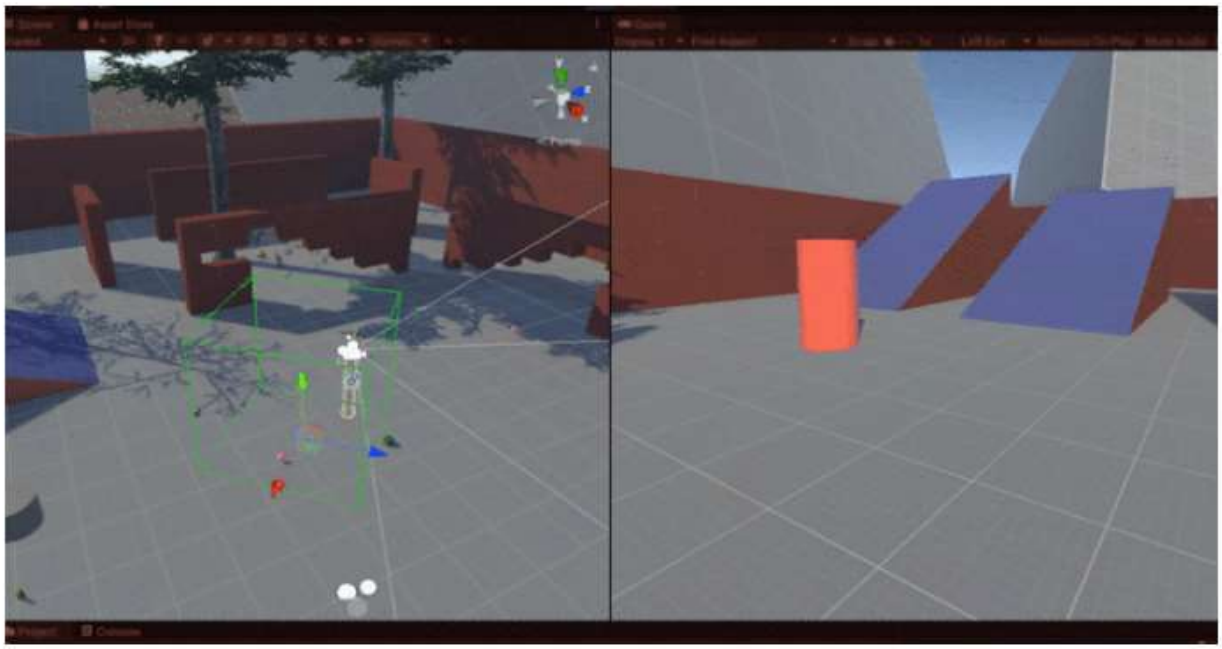


Fig 19. Original (top), Fig 20. snap turn 30 degrees (bottom)

As such by providing an alternative to researchers to use in the method of snap turning this also brings up more customization options for the tool. An example of this is allowing for modification of snap turn angles to explore for research. Currently there is no set industry standard for snap turns and allowing a slider for various angles would be an interesting tool to investigate what an optimal angle could be.

21. Participant Controls

The current market for VR controllers is wide with a variety of different options. Some of these options are very advanced from being able to track finger movements to some really basic controllers that are essentially just regular console controllers. This brings the question of what the tool's control set should be designed around. Should the tool allow for researchers to take advantage of finger tracking or is something like a basic controller enough?

Fig 21. Simple “wand” controller (left) vs Finger Tracking controller (right)

The current approach is to design the tool set around the basic control scheme that does not enable finger tracking and just a basic hand movement. There are two reasons behind this decision, time and practicality. When it comes to the practicality of implementing a system that allows for the tracking of individual fingers, the team does not see where this would be useful.

The current goal of this project is to create a tool more focused on path generation and exploring the results from having participants walk those generated paths. Allowing for finger tracking does not really enhance that experience in any way, and as such does not greatly contribute to the goal of the research or experiment. Currently the team also has yet to discover a place where individual finger tracking would be useful in the experience, as the participant would not be grabbing or interacting with any objects. If objects were introduced then a simple “hand” would still provide an adequate solution to grabbing or pushing such objects.

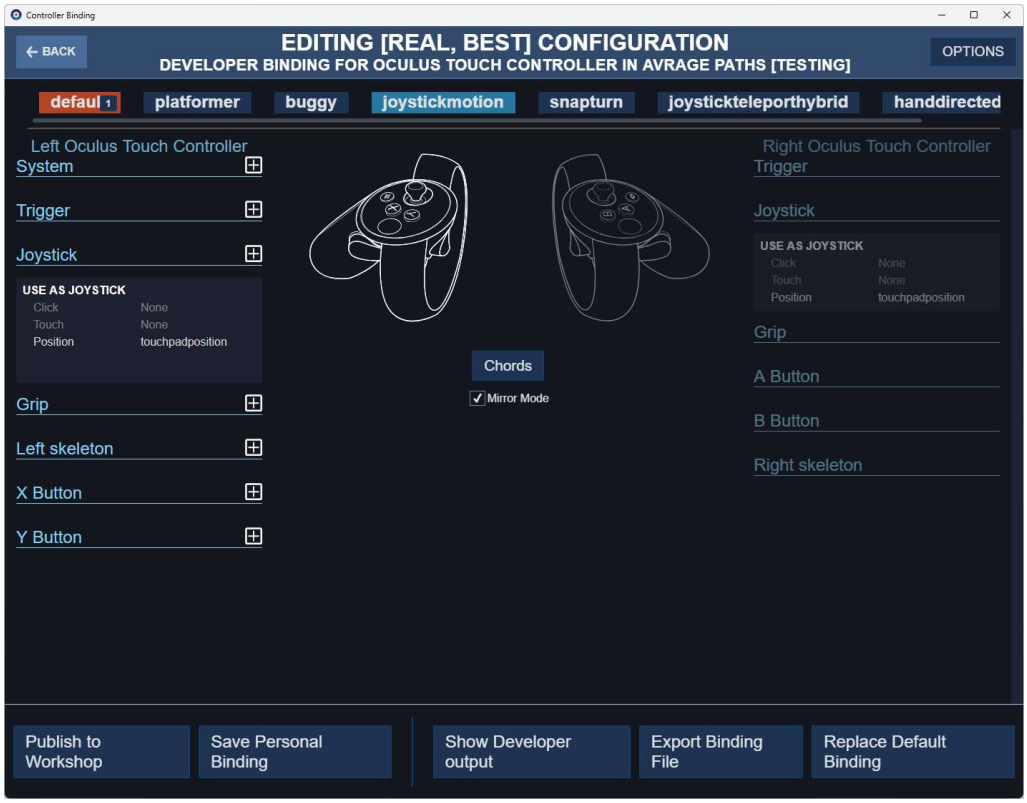
22. SteamVR

22.1. Action Sets and Bindings

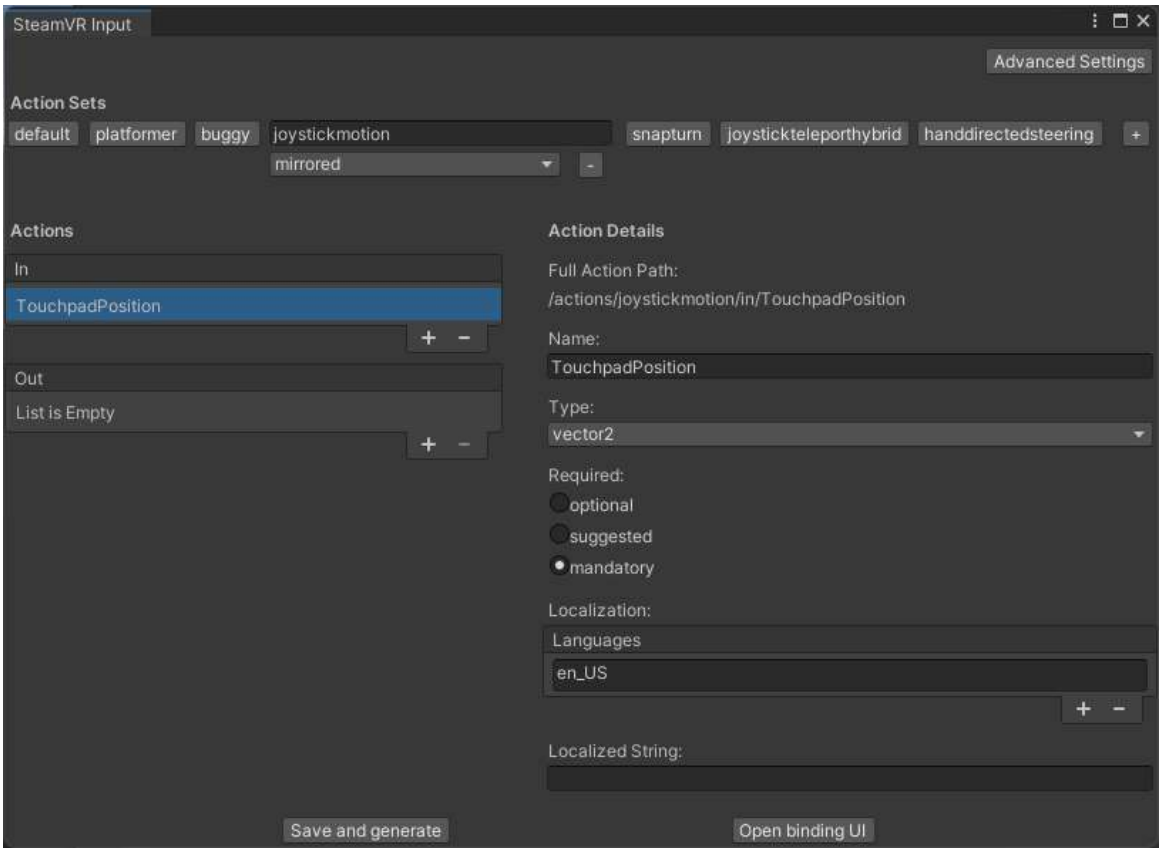
SteamVR is a useful VR API that we utilize for many different functions within the tool. Namely, SteamVR provides a method of binding controller inputs to actions that can be utilized in script. This is the backbone of our locomotion systems, as it is what allows controller input to be used to control player movement.

A SteamVR *action* is simply some type of input. It could be a boolean representing if a button is pressed, a vector describing joystick direction, a float describing how much a digital trigger is being pressed, and so on.

To use actions, they must be within an active *action set*. An action set is simply a configuration that binds controller specific inputs, like button, joystick, and triggers, to predefined actions. For an example, we defined an action set specific to our head directed motion locomotion system. It binds an action, specifically a SteamVR_Vector2 named touchpadposition and is assigned to both joysticks on the Quest 2 controllers.



The SteamVR Binding UI, showing the action set used in our head-directed steering locomotion method.



The SteamVR Input tab in Unity, where one defines the actions for a given action set.

SteamVR's action system has a few notable quirks that are worth pointing out for any future developers.

Multiple action sets may be active at the same time. Depending on the priority given to the action set when it is activated, an action set will either overwrite or be overwritten by other active action sets. If two action sets are equal priority, then both actions defined by those action sets will occur when the binding is used. This is important to understand if you plan to use SteamVR's default action set, which comes with many useful bindings and actions, in addition to your own action sets.

A binding of "None" is different from no binding at all. In the figure depicting the SteamVR Binding UI above, note how the "click" action of the joystick is set to "None" whereas actions for the other input types are simply empty. This has consequences: if this action set is activated with a higher priority than any others, the "click" action will be unbound, regardless if lower priority action sets define some action for it. This is a quite unfortunate limitation of the SteamVR action set system, as you may not necessarily

want that “click” binding to be overwritten. The only way to get around this is to define action sets of equal priorities or to simply bind those actions in the higher priority set.

Action sets are controller-dependent. You will have to create bindings through the binding UI for every different controller type you plan to use.

Further information regarding SteamVR can be found here.

22.2. VR Player Prefab

SteamVR comes with a base Player prefab, which automatically creates the VR camera and tracks the headset, handsets, and provides support for teleportation. While this prefab is great, it is missing a few things we need to facilitate our own locomotion systems. We will go over the additions and explain our reasoning for each one and describe their purpose.

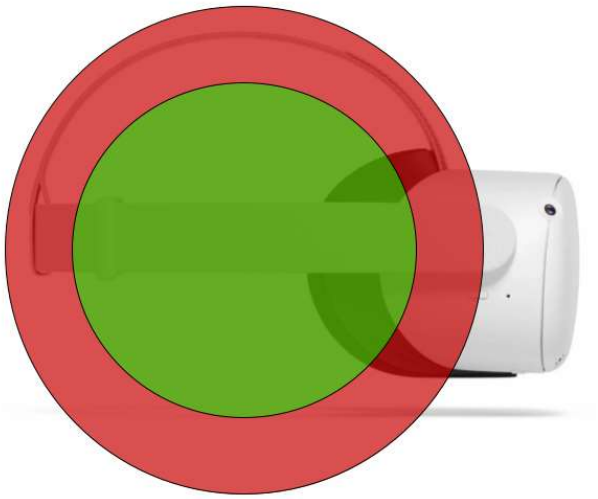
22.2.0.1. CharacterController

We added a CharacterController to the prefab. This is because the SteamVR Player only supports teleport locomotion by default, so it has no need for collision or movement. We opted to use a CharacterController over a RigidBody because it easily resolves collisions and handles the movement physics for us. This CharacterController is what our locomotion systems use to move the player through the maze.

22.2.0.2. Collision Resolution

We added two collision spheres to the prefab, rooted under the head transform. These spheres are utilized in our collision resolution manager, which is responsible for disallowing a player to peek into walls. The collision resolution manager has two parts: one that attempts to push the player out of the wall, and another that fades the player’s view to black.

When the outer sphere collides with a wall, the player is pushed from the wall. When the inner sphere collides, their view fades to black. We opted to fade their view to black as the illusion of impossible spaces was important to uphold, and this minimizes the chance that a curious player is unable to break the illusion if they manage to get deep into a wall.



A figure depicting the two spheres used by the collision resolution manager.

22.2.0.3. Control System Manager

To facilitate switching locomotion methods, we created a system to activate and deactivate control systems.

To define the different options one has for locomotion methods, we define an enum called *MovementTypes* that lists each option. On the VR Player prefab, the control system manager has a list of GameObjects, each with their own script attached that implements the locomotion system. This list of GameObjects, called ControlSystems, is indexed using the MovementTypes enum.

The control system manager handles activating the appropriate GameObject within ControlSystems and deactivating all others. Note that because our method simply enables or disables the GameObject which has the locomotion script attached to it, it is necessary to do any setup and cleanup in the MonoBehavior methods OnEnable or OnDisable, rather than in Start. This prevents any unwanted behavior when using multiple locomotion methods on a single usage of our tool.

Below is an example of a script implementing (an albeit simple) locomotion method. To integrate this into the control system manager, one would add the value “ExampleMotion” to the MovementTypes enum and add the GameObject containing this script into the appropriate index of ControlSystems.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Valve.VR;
using Valve.VR.InteractionSystem;

public class ExampleMotion : MonoBehaviour
{
    // The CharacterController you wish to move
    public CharacterController player;

    // The ActionSet containing the actions you want to use
    public SteamVR_ActionSet exampleActionSet;
    // An example action that is within exampleActionSet
    public SteamVR_Action_Vector2 exampleInput;

    void OnEnable()
    {
        // Setup
        SteamVR_Actions._default.Activate(priority: 0, disableAllOtherActionSets: true);
        exampleActionSet.Activate(priority: 1);
    }
    void OnDisable()
    {
        // Cleanup
        SteamVR_Actions._default.Activate(priority: 0, disableAllOtherActionSets: true);
    }

    public void Update()
    {
        // Example of taking input and using it to move the CharacterController
        Vector3 moveDirection = exampleInput.axis;
        player.Move(moveDirection);
    }
}
```

A figure depicting an example on how to set up a locomotion system to work with our control system manager.

The MovementTypes enum is used to populate the dropdown options on the User tab of the UI. Thus, if you wish to use the UI for selecting a locomotion method, you must use our system or extend it to use yours on your own.

22.2.0.4. Debug Mode

As you may not want to connect a headset every time you use our tool, we implemented a method of traversing the maze with keyboard and mouse. We call this Debug Mode, and it can be activated using the Debug Manager component on the VR Player prefab.

Activating Debug Mode will allow you to use the mouse and keyboard to traverse the maze. This controls the same prefab that is used when actually using VR, so it is exactly the same as running through the maze with VR.

Debug Mode uses a separate camera than the VR Camera, and it renders to Display 4.

23. Algorithm

During our development and brainstorming process, we found that the best way to come up with solutions was to individually think about the problems at hand, then collect our solutions and cherry-pick the best parts of each. We think this approach worked well because of the variety of different perspectives it brought. Thus allowing us to come up with the best possible solutions we had.

With this in mind, those of us on the algorithm portion of the project decided to research and come up with our own ideas individually, to get a wide variety of perspectives on the problem. The problem of dynamic maze generation with impossible spaces is somewhat complicated, with many different systems, details, considerations and requirements, so it was crucial that we had the best solution we could come up with.

So, the following section is a recollection of our individual research. Each is broken down into individual subsections that the individual decided was necessary for a better understanding of the problem at whole.

23.1. Introduction

When coming up with some sort of algorithm, we have to take into account many different aspects. We can start by trying to approach the problem in a human manner, and building on it from there. However, before we get too far ahead we should outline what exactly we are trying to solve.

We are trying to find a way to generate a maze. Simple enough, right? Well, firstly, let’s discuss the intricacies of that before we get into the specifics of what we want to develop specifically for our needs. With maze generation, we want to be able to make some sort of path, from beginning to end that has a few of the following:

- Dead ends

- Straights
- Corner pieces
- Choices

With these, comes many issues. Let us deal with each of our options as a separate entity, hopefully finding some common ground to combine them all.

23.2. Different Pieces

For the dead ends, they are a closed and shut case. We just make some sort of box, but instead of being fully enclosed, at one end, we have a portion that leads into the rest of the maze. Currently, there is no solution for the rest of the maze, but assuming we can arrive at this dead end, we can simply implement it as some sort of corner. Before moving on, let us make sure we consider all of our edge cases, if any:

- Should we have any dead ends a different size of shape than others?
    - For the sake of our project, it is likely that keeping it simple will be more advantageous due to research, but we can delve into that more once we figure out some more parameters.
- Could there be dead ends involving 2 or more paths?
    - By the definition of a dead end, it would converge to one point. If we have 2 paths intersecting, it would devolve into a corner piece. This is something we can tackle later.

To move forward, we will now consider the straights, which are the connectors between other options within our maze. Is there something profound about them? They are so simple, only one way to go! Although they may come off as one dimensional, there is plenty to discuss:

- How long should a straight be?
    - This is contingent on some parameters provided to us for the research, but it would be advantageous to make them of variable length. Having a fixed size could pose some issues for us in the future.
- Should it have paths branching from the sides?
    - If we have a path that is going straight, but we can choose to go in another direction, this will devolve into another case we will discuss later, known as the choices.

Now we can corner in on our corner pieces. Between two straights, we need some way to connect them. If we have a dead end or a choice piece, those turns will be handled by the connections to those, but individual turns will operate differently. With these, we can look at our different options and try to make some conclusions:

- Should we just have one type of turn? U-turns too?
    - Within our maze, we want to allow for different types of turns. The most basic being an L shape which would connect to two different straights.
    - The only other option for a turn would be a U shape. This can be proven since adding an edge to L creates U, and adding any more edges to U leads to a choice or circle.
    - Now that we have two options, we have to figure out which one to use. When should one corner piece be used over another? That would be reliant on some predefined parameters that would indicate which to use at which moment.

Lastly, we can decide to look at our choice piece. This is the most complex of all the pieces, but still fairly simple in idea. Based on what we already described, we are limited to 4 directions: up, down, left, and right. Let us see if we can hone in on any edge cases or issues with our ideas here.

- How do we know which direction to use?
    - The direction to use can vary based on the paths surrounding, but also heavily based on some parameters provided to the algorithm.
- Why are there not more directions than up, down, left, and right?
    - When we previously defined our corner pieces, they are exactly that, corners. If we combine to corners, we can get something that looks like ⊥, and if we add another corner, we get +. Adding any more corners would be redundant, thus leaving us with the four directions.

We have covered each of our building blocks for our maze, and now we can see some of the common patterns here. Within each unit that has a choice in regards to the maze's generation, we will never know for sure which piece to use. This is dependent on what parameters are provided. The other aspect we notice is that each piece builds on the previous, since they all eventually become the choice piece, except for the dead end.

Can we make any conclusions about any of this? We have two large areas to explore, which are parameters, and how we intend to piece together our maze. Parameters will be discussed during the examination of the different methods of piecing

together the maze, and a more in depth discussion can be found in other parts of this document.

23.3. Creating the Objects for the Maze to Generate

23.3.1. Forming a distribution

Before the Maze is able to start thinking about generation at all, we first need to understand what kind of pieces we want to go into it. We begin by first taking over all the inputs that the researcher has explicitly entered values for (e.g. copy over 4 straights, 2 fourways, etc.). After we have copied all of the pieces over and entered them into a data structure, we simply take the sum of all the pieces; while calculating this sum: we check whether or not this piece was marked as random. If it wasn't marked random, we add the value to the sum and move on. If the junction was marked as random, than we first check if the value is a straight. If the value is a straight and marked as random that this forms a special case. We first add to the distribution the minimum amount of straights that are required for the current number of pieces added, which is simply equal to x * min where x is the total number of junctions they've added (no including the straights) and min is the minimum number of straights allowed to spawn in a row, as specified in the U.I. by the researchers inputs.

After the minimum number of straights has been calculated and added to our distribution of pieces, we simply just make the check if we're allowed to include any additional straights in the distribution outside of the minimum number that was added to the distribution. We do this check in the following way: first take the current value of straights (this will currently be the minimum number of straights allowed in the maze) add 1 to that value. We than take the distance and subtract the new value of the straights we just made from it; this new distance is essentially equivalent to the number of junctions (as everything either has to be a junction, or has to be a straight). We then simply check if this new value of straights is less than or equal to our new number of junctions (the value we just calculated using distance) * the max number of straights that are allowed to be spawned in a row (again specified by the researcher through the U.I.). Because we have structured things in such as way that each junction is followed by a number of straights, we know that we can not have more straights than the case where every junction is followed by the maximum number of straights; and this is why this calculation works. Finally, now having the minimum number of straights already in the distribution, if we are allowed to have more straights, than we simply just add the straight junction type to our list of pieces that was marked as random. If we can not

have more straights than we simply just continue on to the other remaining pieces, without adding it to our list of random values. Now that we have straights covered, the more general case for the remaining pieces if they were marked as random is to simply add a value of 0 to our sum of the distribution, and then add that specific junction type to the same list of random junction types that we added the straights to in the former case. Once we have iterated through all of the pieces, we simply just subtract the sum from the total distance that was specified by the user. If we have 0 remaining distance, perfect! That just means that they used all of the pieces and left everything else marked random for convenience.

However, for the other case in which we still have remaining distance, this is where the list of random junc types that we just made will come in handy. We start by finding the maximum index of our list of random pieces and the minimum (this should always be zero). From there we enter a loop while we still have remaining distance. Within this loop we simply just roll a random number; this random number generation follows an even distribution with about 90% entropy by using the technique seen here https://math.stackexchange.com/questions/2580933/simplest-way-to-produce-an-even-distribution-of-random-values. By following an even distribution with entropy, we can be better assured that we'll avoid the bad cases of rolling just a really large number for one of the pieces if they researcher marked all the pieces as random (e.g. all pieces marked as random and a distance of 10; if we weren't following an even distribution than you could technically get 5 fourways and 5 straights; the least amount of variation possible in a maze with default parameters for straights). To further play into this even distribution scheme, we only ever choose a single piece at a time, meaning we are actually rolling values for the index themselves and just incrememting the value for that piece in our distribution whenever we land on it; this again, is to reduce the bad case formely described.

Once again, straights form an edge case to this random population. If the index rolled happens to belong to a straight junction type than before we add it to the distribution we again have to verify in the same way as before if we can have more straights without violating the minimum straights in a row and the maximum straights in a row parameters set in place by the researcher. If this will violate those parameters than we DO NOT increment the number of straights that are within the distribution and just continue to roll for other allowable pieces. This overall process continues until our remaining distance is equal to 0.

23.3.2. Bucketing Our Straights

Now that we have explicit discrete values for every possible piece type that will wind up in the maze we need to bucket up our straights. So firstly, what do I mean by bucketing the straight values? While as you know from so diligently reading this document, we abide by the following rule: every junction must be followed by at least the minimum number of straights in a row, at and most the maximum number of straights in a row. When first thinking about how you would want to place straights in the maze, its easy to think about using a probabilistic model for placing straights (i.e. after every junction just roll a random number between the min and the max, and that will be th enumber of straights that we spawn). This line of logic is quite intuitive to think about and even better, super fast to implement. However, it comes with quite a problem when you need to abide by an explicitly defined distance, and explicitly defined number of straights that we need to include in the maze. Take for example we have bad luck and happen to roll the maximum number of straights every time following our probabilistic model. This would mean that by the time we reached the end of our maze we would have two options, either break the distance of the maze and subsequently the distribution of the pieces by continuing to put down straights, or we could break the promise of the bounds that the researcher entered and start adding less than the minimum number of straights allowed because we simply don't have any straights left to place. Both of these cases are inexcusable if one of the major purposes of our application is to have a standardized distance across runs. This lead to the idea of bucketing, by breaking our straights in n groups between our min and max value where n is the number of junctions, we can avoid a repeated probabilistic model and ensure consistent behavior across runs. So, how does this work?

Firstly, its absolutely essential that this part of the algorithm is fed valid inputs, if something invalid is fed it simply won't work (i.e. 8 straights 2 junctions, distance 10, min 1, max 3, is not a possible scenario). Each junction would need to be followed by 4 straights in this example, which the maximum value provided by the researcher would not allow. So assuming valid inputs, we start by making note of a remaining sum value, this is simply initially equal to the number of straights that we have to work with. From there, we iterate through the total number of junctions (again excluding the straights), because at the end of the day, we want there to be equal buckets as there are number of straights. At every step of the iteration through this loop we perform the following: assign a min value. The min value is simply the maximum of either the minimum straights in a row, or the remaining sum (the number of straights we have to work with) - maximum number of straights allowed in a row * (the number of junctions - the number junction we're on - 1). Similarly, we get the maximum possible value by taking the minimum (the smaller of the two) of the maximum number of straights allowed in a row

or the remaining sum - the minimum number of straights allowed in a row * (the number of junctions - the current junction we are on - 1). Notice how these values will always strictly be between the minimum and the maximum values. We simply then just a roll a number between the minimum and the maximum and that determines a single bucket. Notice how are buckets always fall between the minimum and maximum values as requested, however, when the minimum doesn't equal the maximum, we are left with more freedom for choosing bucket values between.

23.3.1. Object Generation

Now with a full distribution of pieces, and a list containing the all of the straight values bucketed into corresponding slots matching the number of the junctions, we are ready to kick off our actual object generation. Its worth noting that this is done in two ways depending on the type of generation requested by the researcher (i.e. typical maze generation or explicit maze generation).

23.3.1.1. Typical Generation's Objects

We can begin by looking at the typical maze generation, which is the more difficult of the two cases. When using typical generation, the maze calls on the use of two objects, a stack of pieces to guide the visual system, and a dictionary of junctions used to ensure path lengths and the distribution is upheld. The section discussing the algorithm at large should go into detail, but keeping things brief here: the stack provides us more variation at the beginning of the maze, as it is based off of what the user can see, not where they are. This however, means that the stack is often left to small to provide enough pieces for the entire visual system. For example, take the case where its a maze with all fourways. At the very start of the maze, the ending should technically be visible because its a straight shot. However, if the ending is visable, that means that all of our junctions in the stack need to get spawned for that one path on the first fourway. This subsequently leaves no remaining junctions to attach to the other two paths on the fourway, and in such as case, our remaining pieces object takes over for the stack.

Now with an idea of why we have these objects and their purpose, lets get into some implementation details. Due to all the previous work we conducted forming the distribution and bucketing our straights, the actual generation of these objects is trivial. We simply just iterate through our distribution, choose a junction, assign it an id (in numerical order starting at 1), and give it one of the buckets from our list of buckets; we then just add these objects to both data structures. And thats it for generation, when it

comes to maintenance and actually use from within the algorithm. MazeGenerator will always first try to use the stack, if the stack is empty, the algorithm will check if it should actually be empty or if there was just multiple paths connected to our current junction that exhausted the stack. The algorithm does this by maintaining a list of the pieces it has spawned relevant to a single socket on a junction, it then simply looks at the length of this path and the length of the remaining pieces structure and if remaining pieces is larger than the path at hand, than we know that the stack has actually been cut short. In such an instance, the algorithm will then just get a piece from remaining pieces that is NOT contained on the path and spawn that piece instead. We heavily rely on the id system implemented with the junctions to ensure no duplicates are fed to and from the stack and remaining pieces as the junction itself is not sure which data object it came from at the end of the day. Furthermore, its worth noting that our concept of remaining pieces works so well because the algorithm never actually removes a given piece from the object when asking for it, rather, a different component of the algorithm (outside of the generation itself) removes the piece only after the user has physically stepped on the piece in the game space; alternatively, the algorithm will add a piece back to the remaining pieces structure in the event that a user back tracks. With these conditions placed on the object, we avoid the pitfalls caused by the vision system and don't run the risk of running out of pieces do to bad luck with generating an extremely long hallway.

23.3.1.2. Explicit Generation's Objects

Explicit generation provides a much easier case as in explicit generation every path will be the exact same; meaning that at a given junction, that junction can have a maximum of two types of junctions connected to it, the junction type that you entered from, and the junction type you will be leaving from. Because of this distinction, the use of the stack to provide variance as we did in the former method no longer makes sense. So this mode of generation simply just forms an id system, in which the algorithm can simply just provide the id of the piece its currently on, and receive the id of the piece that should come after it. So this behaves largegly similar to the creation of the remaining pieces object seen in typical generation. We simply just iterate through our distribution, choose a junction, assign it an id (in numerical order starting at 1), and give it one of the buckets from our list of buckets; then we simply just add all of that information to a dictionary with the id being the key. This process is as simple as it sounds, so it might not be discussed in great detail later on in the algorithm section.

23.4. Building the Maze

23.4.1. Introduction

When building the maze, we have to look at the conclusions drawn above, and find some common ground. Let us lay out everything we know once more to get a better idea of what we have derived.

- The pieces mostly combine to create one another.
- They are the same size and shape, except the straight.
    - The straight should vary in length.
- The dead end is the only obscure piece which does not conform to the others.

If each of the shapes combine to create one another, then we can create some sort of universal shape that can be changed to fit the different forms. This can be complex, but worth discussing. Let us assume we have a straight, which is the most basic of the forms. If we attach a second straight to the side of the first straight, then we have created a corner piece. Before we move forward, we have to come up with some way to both find where they should be connecting and create that opening. Otherwise, we would have an enclosed passage connecting to another enclosed passage, which would be useless. To solve this issue, we have two options:

1. Modifying the vertices of the object with Unity's ProBuilder.
2. Creating a plethora of basic objects with openings, in addition to the enclosed objects.

The first option can be quite complex when it comes to implementation, and the second option can become a rabbit-hole of creating endless prefabs which fit our purpose, or some new subsystem which can procedurally create holes within walls.

Both of these solutions do not seem like they are helping to solve the problem much, with one being too complicated, and the other, impractical. If we move to our second point, we can bring in more data, which may pose new issues that help us converge to a solution.

If all the pieces should remain the same size and shape, we can try to find a system that utilizes that property to our advantage. Here, we can likely utilize some fixed step system, where we can predict all of the sizes of our pieces. Doing so provides a way for us to maintain consistency within all of our pieces. That is necessary here since we need our pieces to line up correctly, otherwise we would have some misaligned edges which would break the illusion of a seamless maze.

23.4.2. Connecting the Pieces

If we have solidified that we are using a grid system, we now have some more data we can work with. There are two major aspects which will help us construct the maze:

- Step size
- Connection points

The step size is what allows us to maintain consistency throughout our maze, and the connection point is where exactly we want to make the transition between one segment and the next. We discussed our options of how we would like to connect the mazes before, but as a reminder, here they are again:

1. Modifying the vertices of the object with Unity's ProBuilder.
2. Creating a plethora of basic objects with openings, in addition to the enclosed objects.

With both of these options, it may be difficult to decide which would be a smarter choice, or perhaps it would be best to combine the two. With our new data of the utilization of a grid system, we could likely find some solution to best fit our needs. We can solve this dilemma by tackling each option, one at a time.

23.4.3. Placing Walls

The other choice would be to create a plethora of static objects that can be put together, similar to building blocks. Let us determine each type of possible object we could have to create our maze based on what was discussed prior. We have four choices:

- Dead ends
- Straights
- Corner pieces
- Choices

If this is all we have, we can try to find some common pattern when it comes to actually creating them.

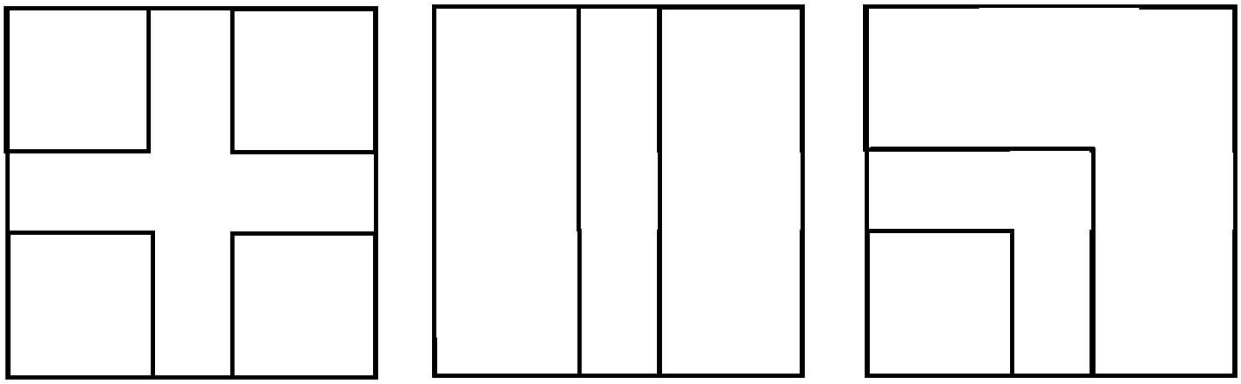


Fig 23: A collection of our different options for our maze. From left to right: Choice, Straight, Corners

Here, we can see our different options for our different segments. The squares they are enclosed in would represent a single point in our grid. If we examine these shapes closely, we can see that the first segment, the choice segment, contains all of the other segments inside of it. Perhaps we can find a way to use that to our advantage.

If we are able to simply put walls to block off certain areas, this would allow us to provide any of the other pieces, just by using the one, 4-way choice mesh. For example, if we wanted to create a straight, as seen in the figure above, we can place a wall on the left and right paths, limiting the user to only go forward. Similarly, we can place walls on the top and right paths to create the corner piece seen in the figure. However, is determining where to place the walls trivial? Assuming we know information about the current piece, such as the current walls blocked, we can construct the other pieces. For the sake of discussion, the possible locations of the walls will be discussed by using north (N), south (S), east (E) and west (W):

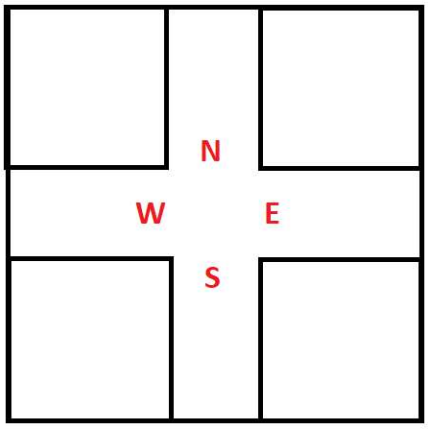


Fig 24: Shows the directions that will be used to refer to wall placement locations.

- If we are currently in a straight piece

    - Choice junction can be made by simply removing or adding whichever walls we would like to make the choices. This is independent of which walls were open prior, but we must keep the connecting wall open.
    - Straights can be made by copying the wall locations from the current straight.
    - Corners can be made by keeping one of the current walls, and making a new wall in the last unused con.
    - Dead ends can be made by adding a wall in all locations except the originating wall.
- If we are currently in a corner piece
    - Choice junctions can be made in the same manner as straights approach the problem.
    - Straights can be made by keeping the north wall and adding a south wall.
    - Corners can be made by placing walls anywhere other than the opening wall.
    - Dead ends can be made in the same manner as straights approach the problem.
- If we are currently in a choice piece
    - Choice junctions can be made in the same manner as straights approach the problem.
    - Straights can be made by adding a wall on any two parallel paths.
    - Corners can be made by placing 2 adjacent walls.
    - Dead ends can be made in the same manner as straights approach the problem.

With this outline, it becomes quite easy to build shapes using the one mesh, and thus, a valid solution. The ability to use the N, S, E, and W markers would allow for trivial access to each of the walls for comparison, since they would all be within the same coordinate system.

Minor issues with this would be ensuring that the walls created are seamless, both with textures and vertices. Both of these problems, however, need to be solved once, since they can be applied to all of the other directions, and for any one piece.

One major issue with this concept is that it does not allow for paths with different sizes than the original 4-way choice. For example, if it was desired to have a straight

that was ¼ the distance of a normal straight, that would not be possible due to the placement of walls dictating the size of our segments. Shrinking the size of the pieces would cause distortion and be quite problematic, so it is best to avoid that idea. It can only be concluded that different sizes would not be possible with this method. Based on factors and conclusions in other aspects, this may not be too big a worry, but it is also best to keep searching for other ideas.

23.4.4. Graph Analysis

Our last possible solution to most of our issues would be to look into how exactly we connect our pieces. With current means of solving our problems, we utilize some sort of grid. This allows for a quick check in a spot within our grid to determine if a piece already exists there. However, this does cause problems, as we have seen in the other methods above, when we try to utilize pieces with lengths different than one unit within our grid.

To get around this issue, we can make an analysis of graph theory, and how exactly graphs are connected in computing. When making graphs, the two typical representation in a coded software are as follows:

- Adjacency Matrix
- Adjacency List

Both of these are a viable means of connecting our graph, but do not necessarily solve any problems. These help when the software needs to see what is connected to what, but our issue is more fundamentally about checking if a piece of our graph is existent, or visible, at a specific area. To solve this problem in a graph, different graph traversal methods can be used, given a start point and an end point. The common algorithms used for graph traversal are depth first search (DFS) and breadth first search (BFS).

DFS is primarily used for going down the extents of one specific path, before returning to a point of divergence. BFS is used when it is desired to explore each path, layer by layer. Both have their justifications for usage, but the one to choose depends on the application. For the maze we intend to create, it would be smarter to use BFS, primarily due to the ease of exiting early.

Considering how the maze is constructed, it would be beneficial to conduct a traversal in a similar manner. Since at each junction, it is necessary to load in at least some parts of the maze up until the new path is taken, BFS would allow for the program to explore from the center of a junction, going outwards. Compared to DFS on the

average case, BFS would likely stop execution faster and arrive at what we are looking for.

After concluding we can utilize BFS to search our paths, we need to return to the question of how we represent our graphs in code. There needs to be a means of connecting one junction or straight to the next. As discussed prior, we have our two options: adjacency list and adjacency matrix.

Adjacency matrices are more often used for a graph with many edges, since it is dependent on the number of nodes. Adjacency lists are better for graphs with few edges, since it expands based on the edges. When it comes to adding new edges, both the list and matrix can do so fairly quickly; more specifically O(1). When it comes to searching to see if an edge exists, both are also very quick with O(1) lookups (this is dependent on adjacency lists being built on hashmaps). When it comes to adding new vertices, adjacency matrices can be quite slow, since the matrix has to expand to support the new vertex, specifically $O(|V|^2)$. For adjacency lists, the new vertex can be simply added on in O(1) time.

With the analysis of all these pieces, we can try to make some conclusion of how to best represent our maze in the code. We can see that the adjacency lists overall seem to be more efficient. The main downside to adjacency lists is that the space complexity can become quite large for a graph with many edges. Adjacency matrices do not have that issue, but would become problematic since a new vertex is added very frequently. Whenever the user explores a new path and a new junction is created, that would consequently result in a new vertex being added to the matrix or list.

On the grander scale, the adjacency lists seem to perform better, as the expansion of the matrix on each vertex addition would be disastrous to performance. Comparatively, the issues with the adjacency lists can be overlooked. However, it is important to not only accept the theory derived, but to possibly find means to apply it to the specifics of our project.

Let us return to our primary question: does something exist or is visible in this location? We have concluded BFS, as well as some derivation of an adjacency list is required to answer that. The list format works in the following format:

1. Start at a given vertex
2. Go to the next connected vertex
    a. If there are multiple paths, traverse all with BFS
3. Consider the current vertex as the start vertex
4. Return to step (1).

5. Stop once the destination is found.

BFS is useful since once the location is found, the algorithm will conclude. However, it is important to note that there must also be a depth limit for how deep the search would occur, otherwise the whole maze would be explored.

An additional optimization we can make is the representation of the adjacency list. For our maze, at most, there will be 4 paths to take: up, down, left, and right. If that is the case, we can have an array of size 4 for each vertex to store the connections. This would save on execution time when attaching a new edge since there is no need for additional allocations. This would allow for a quick traversal over each of the arrays for the given depth.

To find the worst case runtime for our BFS with the current structure, we can perform an analysis of how the algorithm would execute and converge on a solution. For each vertex, there are at most 4 different paths to be taken. This would occur, in the worst case, depending on the depth of our search. Our depth is how many layers the BFS should execute for. If the depth is the same as the current level, the algorithm should conclude. Thus, the worst case runtime would simply be ($depth$ x $paths$) → ($depth$ x 4).

This is all quite good, but an additional change can be made to how the adjacency list is represented. Currently, the list is an outside resource and refers to GameObjects within the scene. If we know for sure that each node will be referring to the nodes it is directly connected to, it could be beneficial to have the list embedded with the maze, rather than being a separate entity. This allows for each junction to refer to the individual pieces it is connected to, and thus, the BFS can occur quite seamlessly with that. This primarily makes the concept of coding the BFS more simple and organized, since each junction is responsible for its neighbors.

To conduct this method, it involves making the connection whenever the new piece is initially spawned. Upon creation, the junction should do two things:

1. Connect the new piece’s entrance to the previous piece’s exit
2. Connect the previous piece’s exit to the new piece’s entrance

This simulates a doubly linked list, but instead of simply being an entrance and exit, there are 4 possible routes that can be taken. The modification would simply be an array of 4, or 4 separate variables, instead of the 2 used in a doubly linked list. This caters itself well to the concepts of computing, as this data structure is fairly robust and widely used, meaning there is much information to guide us during the implementation.

Now we have found a great way to connect all the pieces together, but can we solve the main problem at hand? How do we know if a piece exists at a specific location? There is a solution that is not too complicated, but it does rely slightly on a fixed step size. The solution is as follows:

1. Keep a count of each of the directions needed to reach the destination
2. Initialize all counts to 0, except the direction intended to spawn the new unit
3. Conduct BFS, and whenever a route is taken, update the count to subtract one from the opposite of what was taken.
4. The BFS ends when:
    a. The counts all are 0; there is indeed something at that location.
    b. The counts never reach 0 and the depth is reached; there is nothing at the location

Here is an example of how this works visually. The user is currently at the blue start and intends to create a new path. The red indicates the new path, and the black indicates the paths already created.

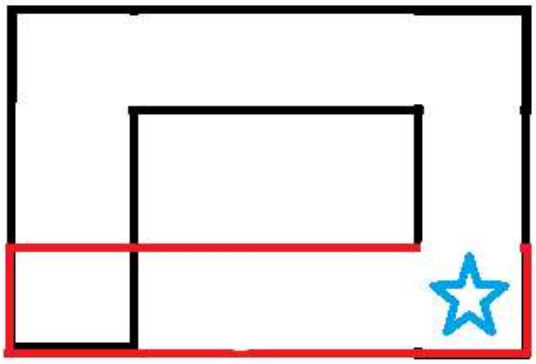


Fig 27. Creating a new path

The new path is to the left of the current location, so the blue star needs to be one to the left. So we initialize our counts as follows.

Vertical	Horizontal
0	-1

We can use -1 to indicate both *left* and *down.* Positive 1 can indicate both *up* and *right*. Then we conduct a BFS in all directions, so the star moves up one. That is the only path from the location.

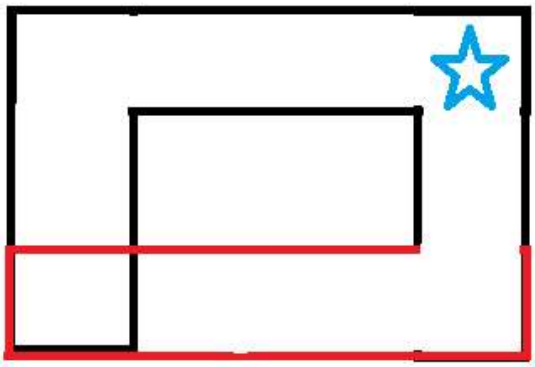


Fig 28. Moving the player

We update our table to decrement our vertical count, since after going up, we need to eventually move down 1 to counter the up movement that just occurred.

Vertical	Horizontal
-1	-1

The star only has one path to go. It should not move back to the right, since that would result in an infinite loop, so it goes to the left.

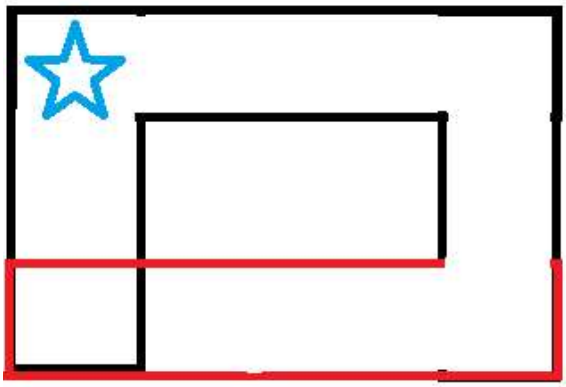


Fig 29. Further movements

Since the star went to the left, we have just counteracted the initial left movement done earlier. This means we can increment our horizontal count by 1.

Vertical	Horizontal
-1	0

Following the same logic, the last move would be to move downward, which is shown below.

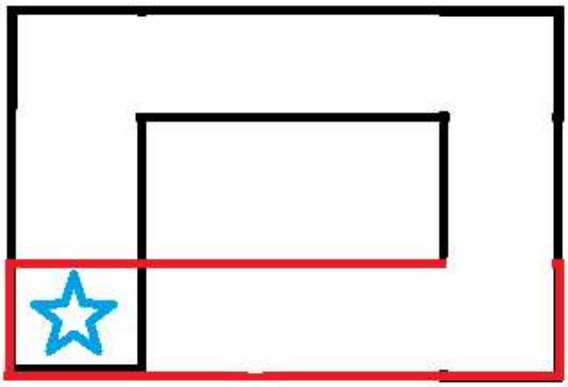


Fig 30. Detecting an overlap

Vertical	Horizontal
0	0

Our counts show that both the vertical and horizontal are 0, meaning we arrived at the spot, meaning there is an overlap. This is also shown visually in the drawing above. Again, this is reliant on there being some fixed step size, however, there are additional problems with this approach.

If our paths do not intersect at the ends, but somewhere in the middle, the way for checking if there is an intersection does not work. Take the following example:

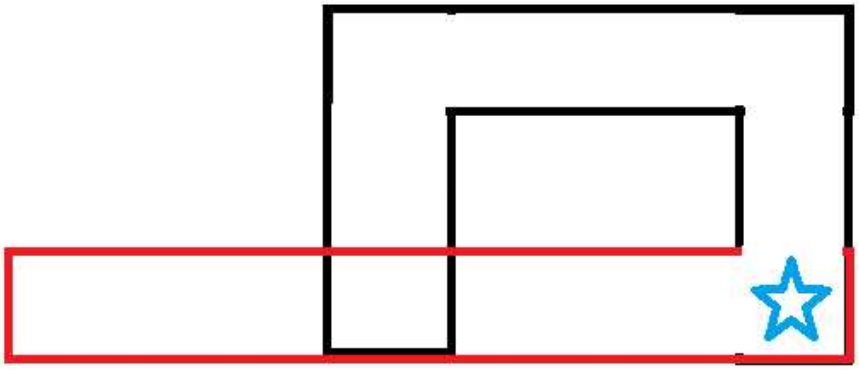


Fig 31. An edge case for detection

Following the algorithm, the left counter should be larger than 1, since the length is definitely longer than the 1 unit seen for the other pieces. Even if numbers are not used, it is quite clear that the red piece is longer than the other black pieces, and that should be reflected in the algorithm. However, this leads to issues since the BFS will eventually arrive at the point of intersection, but the counts would not be 0. That is the basis for concluding that an intersection exists, and it fails at such a trivial level. The solution would be to do some sort of intersection check between the bounds of the two units.

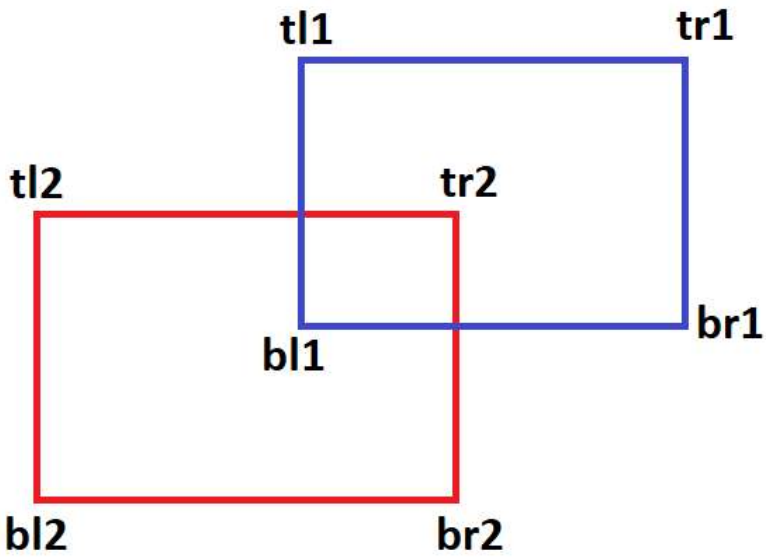


Fig 32. Bound intersection visual

The image above shows two rectangles and their respective vertices. The number corresponds to the specific rectangle, and the letters provide the position:

- Top Left (tl)
- Top Right (tr)

- Bottom Left (bl)
- Bottom Right (br)

Using these markers, there are simple mathematical statements that can be used to determine if these two overlap. Within software, this can also be done with a handful of if statements. Applying this to what was already discussed with BFS, we have a more robust solution.

However, since the maze will be made using Unity, it is important to look at the software to determine if any of the heavy lifting has already been done. When we look at the overlapping rectangles, we may derive that there is something similar between that and collisions within Unity. If we have two overlapping collisions in Unity, the program provides means for the developer to use that data and handle it accordingly. For the uses of the maze, the information would be used to determine if there are two junctions or segments colliding. Thus, instead of doing the calculations with the aforementioned rectangles, we can use the built-in collision system with Unity.

With this new tool at our disposal, there are many possibilities, and one could be a way to get rid of the BFS altogether. While it helped to arrive at this point, there may be a solution without using it at all, but the intelligence of Unity's collision system. If each piece has a collision attached to it, then a simple check of overlapping could solve our primary issue. The requirements for a collision being present would simply be that the piece of the maze is visible. If the piece is hidden, the collision should be inactive. This makes our entire problem of detecting if two pieces occupy the same area trivial, and applies to any one of our different approaches.

This solution is disjoint from the different methods, and thus can be applied directly to whichever is chosen. It is especially helpful in solving the grid dilemma, but can still be applied to grid based constructs. Now, the main concern is determining which method or implementation should be used.

23.4.5. Conclusions

After careful evaluation, we developed a way that gathers concepts from all of the available options. We opted for a grid-less behavior, since it allowed us to infinitely expand the maze in an arbitrary direction; pieces simply load from the previous connection that is empty from a piece, although their initial state is to look the same, which would be a four-way junction, they would be programmatically be updated to become whatever is the desired piece that should be loaded. Since these pre-made objects would also have pre-made meshes, the overall alignment in both axis of the

maze would look like it is on a grid, this will be expanded more on the *Prefab Generation* section of this design document.

23.5. Maze Straightness

23.5.1. Introduction

One interesting part of our dynamic maze generation are aspects of the maze that don't realistically change how the maze works, but rather the “personality” of the maze. More formally, characteristics of the maze can be put into two categories: conscious and unconscious. Conscious characteristics of the maze are those that bring upon some sort of distinct choice or option to the player. This includes having junctions with multiple choices to traverse, where the user has to make a *conscious* choice about what to do next. Unconscious characteristics, however, are those that don't have immediate effects on how the user interacts with the maze. This includes impossible spaces and, as in the heading, the “straightness” of the maze. In this section, we will specifically discuss what maze straightness is, what it can achieve, and how we will implement it into our maze generation algorithm.

23.5.2. Formal Explanation

Maze Straightness refers to sections of the maze that connect two junctions which have user choice (3-way and 4-way junctions), where the sections are not simply straight between the two junctions. In practice, this describes sections of the maze that *could* be straight, or could *not* be straight; the choice is rather arbitrary. Below is a figure depicting maze straightness. Note that this is just *one* possibility of how the maze could wind between the two common points.

A maze / section that is “straight” is one that, in general, makes straight connections between the conscious sections of the maze. In the same sense, a maze / section that is **not** straight is one that, in general, has the inclusion of unconscious segments between the conscious sections of the maze.

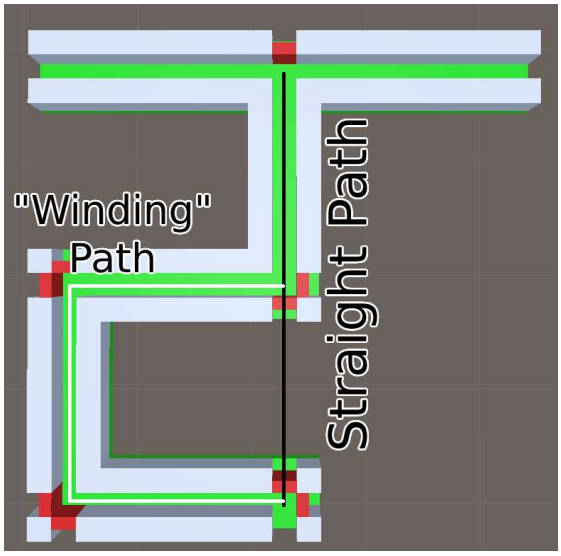


Fig 33. A generated path that “winds” on its path to a common point

23.5.3. What Maze Straightness Does

Maze straightness serves as a way to give the user some sort of variability in their journey throughout the maze. While maze straightness is, again, an unconscious characteristic of the maze, it can still affect the user’s ability to traverse the maze, identify impossible spaces, and more. Furthermore, and notably more importantly, maze straightness provides a medium to increase or decrease the length of the maze without changing the “functionality” of the maze, since it is unconscious. In this way, we can generate mazes with longer sections between junctions that have user choice, further allowing researchers to tune and influence the experience of the user as they traverse throughout the maze. In general, maze straightness and conscious maze characteristics are inversely correlated when plotted against maze distance.

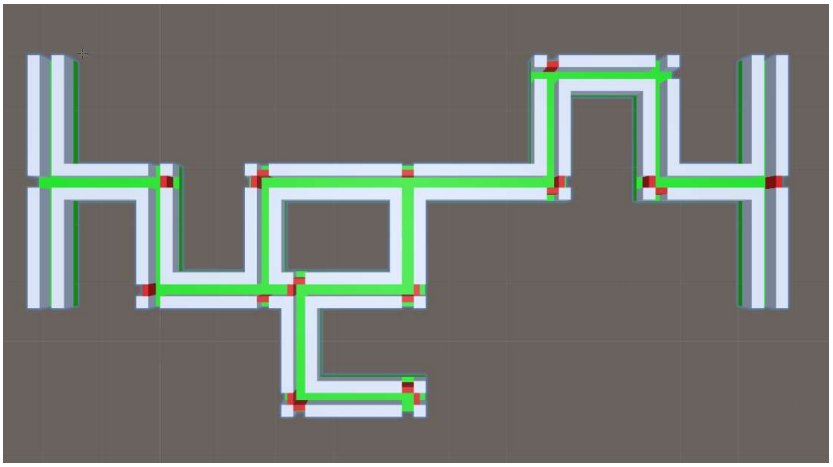


Fig 34. A generation that has numerous windings; a winding maze

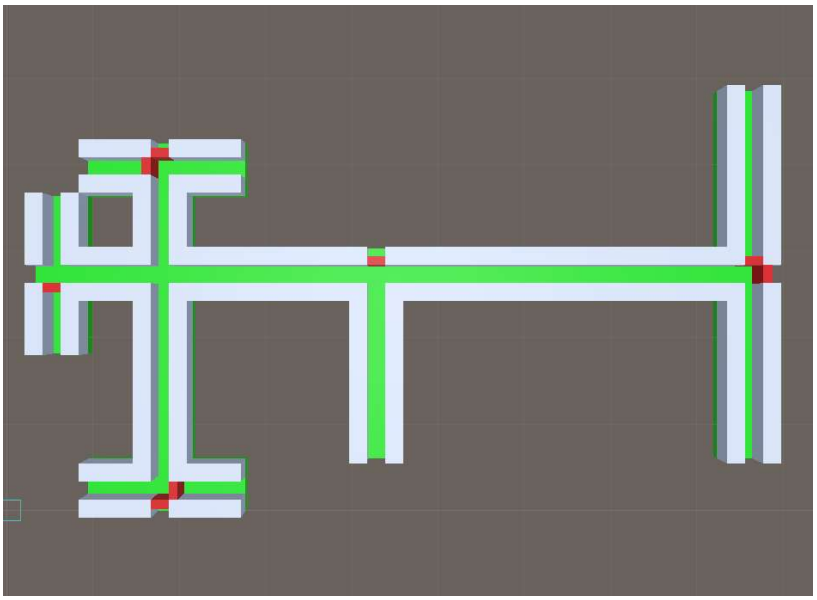


Fig 35. A generation with no windings; a straight maze

Another effect of maze straightness is the influence it has on the density of impossible spaces in the maze. Recall that an impossible space is a spot in the maze where two (or more) sections overlap in physical space. However, the overlapping pieces are disabled such that only the player only perceives one at a time. While *any*

two pieces of the maze can intersect (thus making an impossible space), noting specifically that maze windings can also cause impossible spaces has relevance. As mentioned earlier, winding adds *unconscious* characteristics to the maze. This can make it seem like winding doesn't have any true consequence to the maze itself. Sure - it can influence the experience of the player so they are not just going in one straight line. However, the fact that windingness can create impossible spaces - something far beyond just affecting the perception of the maze but rather the literal possibility of the space being real - brings light to how these unconscious decisions are not *boring*. That is to say, maze winding has “real” effects on the maze.

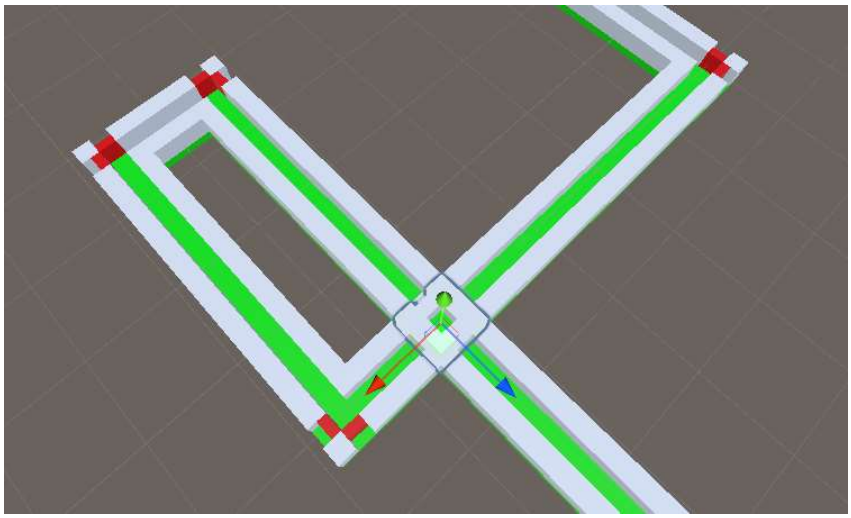


Fig 36. An impossible space generated solely through winding

23.5.4. Implementation

Implementing maze straightness (in a form that is controllable rather than strictly by random chance) is quite *straight*forward. At first, we viewed straightness as some quantifier that would be *applied* to a segment that conceptually already went from point A to point B. This thought process was closely tied to the fact that the straightness of the maze is, again, an unconscious characteristic and as such seems to be something that is applied outside of the main algorithm that generates the maze. However, we can cleverly use a specific configuration of a junction to facilitate straightness in the maze.

By closing 2 adjacent “sockets” on a junction, we create what is essentially a right turn. This junction can be rotated to be in any of the 4 cardinal orientations, making

the piece universal. In essence, this configuration of a junction is a 2-way junction: one way is where the user “enters” the junction, and the other is the only choice the user has to take to continue traversing the maze. In the figure below, one side (such as A) is where the user would enter, and the other is where the user would continue forward through the maze. In other words: the user has no *meaningful* choice to make. As such, we create an unconscious piece in the maze, introducing straightness (or rather, the lack thereof it).

Thus, to implement straightness in our algorithm, we simply need to influence the spawn rate of this specific junction. By including more of these junctions, the maze would generally be windier. By including less, the maze would be straighter. Furthermore, since this is just a specific configuration of a junction, and since junction configurations are already randomly chosen, this configuration could simply come up as a random junction without explicitly influencing the spawn rate of this configuration to facilitate straightness. That is to say, the maze will *naturally* have these unconscious characteristics due to the random configurations of the maze. Even so, being able to have an influence on *this specific* configuration could be useful to researchers so that they can investigate specifically how straightness affects the user’s experience navigating the maze.

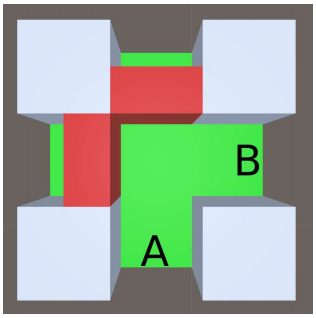


Fig 37. A junction configuration that facilitates winding in the maze

23.6. Code Breakdown

23.6.1. Grid Pieces

In generating our maze, we will use modular pieces that “fit” together to form the maze. (Note that “fit” does not necessarily mean they fall into open spaces that are in

some way cookie-cutter. As will be discussed below, “fit” only means that pieces can interface with each other and seem seamless in their connections. That definition of fit would imply a grid-based approach is used for arranging the maze) The most basic pieces (and the only pieces the toolkit will have off the bat) are junctions and segments. Junctions span from being in 1-way to 4-way configurations, and segments are simply straightaways that connect junctions. Junctions are configured by blocking off various openings on each of the 4 sides. Open sides of the junction have connections to segments or directly to another junction, seamlessly connecting the pieces to form one continuous flow of hallways. Since these pieces need to be connected, we formalize our methodology for these connections: sockets. Sockets are an in-house component that represent areas in which another socket can “connect”.

23.6.1.1. Sockets

Sockets are the places at which grid pieces can connect together like lego blocks, or like train couplers. For segments, *open* sockets are those in which there is not already a connection. A socket becomes closed when a connection is made on it. For junctions, open sockets are those that don't have a connection *or* don't have a wall. As a user, this fact is rather obvious since they can not even see that there is a wall present in the junction; the walls simply make the junction look like a whole different type of junction rather than the same 4-way junction that has parts blocked off. If a connection was made on a side of a junction that has a wall, there obviously would be no way to access the connection as the user. In the figure below, the yellow squares represent open sockets since there is no connection or wall. The red squares represent closed sockets, since there are walls present.

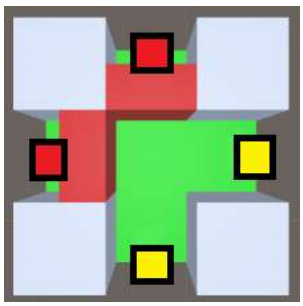


Fig 38. A figure of a junction with its sockets visualized

To use sockets, there is a “connector” and a “connectee”. The connector plays a passive role, simply providing the socket to be connected to. The connected, on the other hand, plays a more passive role. Given one of its also-open sockets, the piece is

moved and oriented to align with the connector. Finally, both sockets are now marked as being “occupied”, which makes them closed.

Using this modular system for how pieces interact allows researchers to create their own grid pieces that fit into existing ones. All they have to do is place transforms on the piece and give it the Socket component, defining the orientation and position of the connection. While sockets automatically mark themselves as occupied when they are used in a connection, custom business logic can be added to the custom piece to control other circumstances when the socket is occupied. One example of this is with the junction, where the inclusion of the wall makes the respective socket on the now-walled side to be closed.

Finally, it's worthwhile to note that having the *pieces* define how they connect to each other rather than having some global grid system where pieces are placed adjacently on the grid has some benefits. Firstly, it's possible to create pieces that don't necessarily follow a grid pattern. Currently, all pieces connect and take shapes that fit squarely into the four cardinal directions, but a (rather experimental) researcher could - for example - make a junction that is Y shaped. Clearly, this shape would break the “symmetry” held by square pieces, making it impossible to use a global grid system for placement. Thankfully, the use of sockets allows the researcher to simply put sockets on each end of the junction and the rest of the pieces can fit like normal. Another benefit of this socket system is that it highly reflects the way we will represent the grid in memory. Since we will be using a tree that links junctions between each other, it's analogous when we view each piece as also having links between one another *physically*.

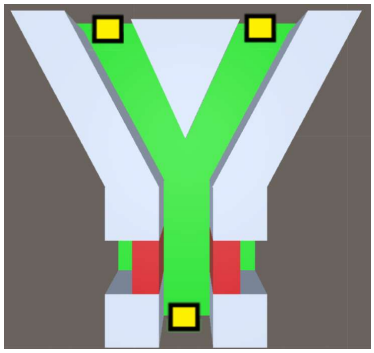


Fig 39. A theoretical Y junction with its sockets visualized, displaying socket utility

23.7. SocketHandler.cs

A single socket. Sockets connect to other sockets using ConnectTo() so long as both sockets are open. A socket is closed when it is marked as “occupied”, which (natively) happens when the socket is either connected to another socket or, in the case of junctions, when there is a wall blocking the socket. Custom cases that block a socket can be added by manually calling the Occupy() and Detach() methods. Sockets should usually be associated with a respective SocketController that is on the grid piece that the sockets belong to.

23.8. SocketController.cs

A controller that groups sockets together. A single grid piece should have one controller, which has a reference to all its respective sockets. Scripts that want to get the sockets of a piece should query the piece for this class.

23.9. JunctionHandler.cs

A 4-way grid piece, where each side has a socket and a respective wall. When a wall is active, the socket is marked as occupied. Junctions are randomly given a configuration of walls at spawn, but can be manually overridden with SetWallsManual(). Junctions should not have all four of their walls active at once, as this puts the junction in a useless state. Randomization of the junction accounts for this bad state.

23.10. MazeGenerator.cs

A manager class that generates the maze using grid pieces. Generations can be manually started and restarted. TakeStep() takes a single step in the current generation. GenerateMaze() calls TakeStep() until the maze is finished. A maze is deemed “finished” either when all sockets on all spawned pieces are closed, or when a maximum threshold of pieces is met. This threshold prevents infinite mazes. This class also provides helper functionality, such as adjusting all the pieces into the view of the camera.

This class will carry the bulk of the “parameters” that define how the maze is generated. This can include the windyness of the maze, the termination type of the maze (length vs time), the allowance of impossible spaces, etc. These parameters will probably be decoupled and boxed into an internal class so that developers do not have to directly work in this manager class. Furthermore, decoupling the parameters makes them easier to serialize to save to disk. Lastly, since the researcher will input their

parameter choices through UI, it is easier to directly change this “package” of parameters rather than access the maze generator directly.

23.11. TelemetryManager.cs

This manager handles telemetry throughout the simulation. Data is laid out according to an internal class, DataContainer. This container defines what a single “package” of data looks like. Data is submitted to the manager in two ways: polling or recording. Polling data is gathered by calling a list of callback functions that follow a delegate PollingFunction. The telemetry manager polls on a regular basis, each time calling these functions and storing their returned DataContainers in the internal data list. The other method of submitting data is through recording. Scripts call RecordData, passing along a DataContainer instance that the telemetry manager will store. All data is stored in one list, which can then be exported off as needed.

23.12. Impossible Space Detection:

One of the more novel parts of our project from a research perspective will be our algorithm's ability to detect impossible spaces. Before we can talk about detecting an impossible space, we first must discuss what an impossible space really is. Impossible spaces are landscapes and geometries that are not bound by euclidean geometry (i.e. a triangle’s angles can add up to more than 180 degrees, or a tunnel can be longer on the outside than on the inside).

Specifically, when referring to impossible spaces in the context of our maze generation algorithm, we are thinking along the lines that making three right turns will not necessarily make a square. To further elaborate, visualize the scenario where the user approaches a junction and can either make a left turn or a right turn; when they make either turn, the segment that spawns overlaps with the segment that would have spawned had they turned the other way. However, the fact that these two segments would’ve occupied the same physical space in the game world is entirely unbeknownst to the user.

Now with a solid understanding of what an impossible space is, how will we detect it? This question is largely determined on how we end up implementing the overall algorithm that handles the maze. It’s influenced by this for a few reasons. Firstly, how will we handle despawning elements of the maze? This is noteworthy information

as if we were to leave the segments in the world for a while, there's the chance that we can use the unity collision engine in some way to detect overlaps. Also, general structure of how we store the maze and read it could be utilized in some fashion as well. Due to all of this uncertainty, I will break down possible methods for detecting when the player has entered impossible spaces.

23.12.1. Using Unity's Collision System -

One of the most straightforward ways to tackle this problem would be to utilize unity's collision system to determine when two segments of the maze intersect. This could be done quite easily through the use of a script attached to the game-world pre-fabs that utilize the onCollide call-back. We could then simply have a singleton game object in the scene that subscribes to all of these prefabs to keep track of when segments overlap.

There are a few areas of uncertainty within how we structure the algorithm as well as Unity's collision system that come about with this approach; we can start our discussion on the side of Unity. Unity's collision system only has access to see when a collision is entered; this creates problems when the segments that we spawn are already colliding with another maze component, as they technically will never 'enter' the collision. Although we can not directly use the easy callback that Unity provides there are workarounds to this problem.

Namely, with objects having colliders, that also allow them to be detected by a physics overlap sphere, which is another function provided by Unity's collision system. The overlap sphere is not typically used for such a task as it takes in EVERY object that it comes in contact with. This is generally an inefficient process in games that have a ton of game objects in any one given location, however, in the case of our application, we really shouldn't have too many game objects detected by any one sphere. Rather our challenge lies in differentiating which segment we are actually overlapping with; this is necessary as a sufficiently large sphere will have the possibility of returning multiple segments attached to a junction.

We could handle this case by doing some quick math. To determine if a given segment from a list of segments will overlap with a given segment we are about to spawn. We can simply compare the necessary coordinates in space, more specifically just look at x and z, which are horizontal components, as we don't associate our maze with having more than one level vertically. Looking at the coordinates we can simply just check for overlap.

Now that we have addressed the shakiness brought about by Unity's collision system, we can now address the limitations that could be brought about by the design of the maze generation algorithm itself. One case that could come about with falsely reporting impossible spaces is due to inaccuracies with placing future segments. Mainly if the ends of these segments intercept by a few pixels then it could falsely be reported as an impossible space. We can solve this problem by simply implementing a thresh-hold value that's equal to the size of the walls. This would eliminate the cases of wall connecting being flagged as impossible spaces, but would still allow for the cases in which a segment fully protrudes through another segment.

An additional limitation that could come about with our algorithm's design would be to not store our segments as prefabs and rather create them from primitive objects on the fly. If we were to only use primitive shapes as opposed to pre-fab game objects then we would not be able to attach the necessary scripts to the components to have them analyze and report when they've encountered an impossible space.

Additionally, if we were to despawn segments prior to them becoming an impossible space then we would lose the ability to sphere overlap them, invalidating our prior procedure. However, if we were to despawn them when they become an impossible space, this would still be acceptable and yield better performance as we would have fewer objects on the map at any given time.

Now with an understanding of limitations and how we will detect impossible spaces, how will we actually document them in a way that makes sense? Now that we are no longer using Unity's built-in call-back for the reason previously discussed, this task becomes slightly more daunting. However, it can still be done through the use of just the singleton manager object we were discussing before. A segment upon being placed will simply determine if it has formed an impossible space and if it has will simply use a public function as a part of our prefab to update our data.

By exploiting Unity's prefab we can actually gather far more interesting information than just whether or not we've encountered an impossible space, rather, we can get the maze configuration the user traveled that led to said impossible space. We want to do this because, at our core, the project is a research framework; with this being stated, any information is good information if we are trying to make a robust framework. But back to the process, if each pre-fab simply stored a reference to the pre-fab that it leads to, then whenever we collide with a segment that constitutes an impossible space, we could simply start with the prefab that we collided with and then traverse our makeshift linked list of prefabs until we hit the segment we're standing on.

Following this methodology, we could additionally have each pre-fab store data unique to it such as time, and configuration. Then upon doing our traversal we could

also get information such as time spent between each impossible space, as well as the configuration that led to the impossible space.

23.12.2. Using Data Log of the Algorithm and Coordinate Data -

Overall this will be a ground-up approach that is largely based on how our algorithm stores data. Although we can not make an explicit claim on how this data ledger will look, we can make the assumption that some sort of ledger will have to exist in order to handle the case of a participant backtracking within the maze. Since we know there will be a methodology to reconstruct hidden pieces of the maze, we can infer that within this data ledger there is going to be some sort of coordinate information so that the algorithm can respawn the missing piece to provide the illusion that it was there the entire time.

Knowing that we have access to the coordinates of every piece that has been spawned will therefore give us the power to simulate the reconstruction of previous pieces in order to check if there's any overlap with our current piece. As we described in the former method, we could once again use some sort of threshold value that a piece would need to exceed in order to qualify as an impossible space. This fixes the problem of walls being counted as an impossible space falsely.

In the most basic principle, I have described, the efficiency of this algorithm would be absolutely appalling, running in O(N) time where N is the total number of segments that have spawned as it just compares the current piece to every other piece. This is nowhere close to the instant look-up time of the previous algorithm and could start to pose problems to the frame rate which is no laughing matter. Furthermore, since we would just be doing a single pass through every piece, we would lose all of the interesting bonus data (such as time spent before encountering impossible space, and impossible space configuration) as it would take a lot more than a single pass to determine the configuration of the space.

However, if we were to make further assumptions regarding the algorithm we can start to see possibilities where we have relatively comparable performance to our linked list of pre-fabs previously described. Say for instance our maze algorithm stores all of the coordinate data using a tree structure such that adjacent coordinates are connected. If this were the case then we could simply just start from the lead we are currently at and trace it all the way up to the root comparing coordinates along the way. This still wouldn’t be as fast as instant access of the former method but would be faster than the previously described algorithm while still allowing us to gather some of the more interesting bonus data such as impossible space configuration.

23.12.3. Comparing our Two Methods -

Both methods aforementioned have their strengths and their weaknesses so I will highlight both to make the decision easier once we have more information on how our algorithm will work.

Collision System

Strengths -

- Super fast algorithm.
- Allows for the collection of more interesting metrics.
- Intuitive design leads to minimal code bloat.
- Easy to implement.

Weaknesses -

- Relies on the maze algorithm using a prefab for all segments and joints.
- Could get slow if a researcher decides to add a lot of additional game-object.

Using data directly from the maze algorithm

Strengths -

- Allows total customization.
- Can still get some interesting bonus data if some additional relational data is saved.
- Imposes no restrictions on the maze algorithm.

Weaknesses -

- Requires the maze algorithm to be solidified before a clear solution can be found.
- Will most likely be slower than the Collision System.
- Would lead to a lot of bloats if we wanted the full telemetry capabilities of the other system.

At this point, it still doesn't make sense to commit to an algorithm for impossible space detection until the maze algorithm is in a more finalized state. However, I believe

the Collision System would make for the best option assuming the requirements to use it are met.

23.13. Mouse Maze Research

23.13.1. Introduction

When we are looking at our proposed maze, we will find many similarities between ours, and a maze given to a rat in a scientific setting. The difference here is that our “rat” is a human, and the mazes are generated with given parameters, rather than being fixed. Considering this, we can make a conclusion that these mazes, in theory, have overlapping concepts, and it is important to explore them to see how they impact our maze.

It is also important to note that this research is independent of the specific parameters we find to implement our algorithm. These aspects impact the maze on a more fundamental design level, rather than being the driving forces that alter the way our algorithm functions. This will be more clear as we delve into our research.

Looking into the document titled, “Some Determining Factors in Maze-Performance” by M. H. Elliott, we find that there are 3 different variables that impact the rats when they attempt the given maze:

- Knowledge
- Reward
- Drive

Each of these concepts have a significant impact on the outcome of rats on the maze, and are all independent from one another. If we delve into our topics further, this will become more apparent. *Knowledge* is the idea that the rat has some prior knowledge about how the maze functions before attempting it, in order to finish it quickly. Without this, the research shows that the rat will end up wandering aimlessly and will not successfully finish the maze in an efficient amount of time.

23.13.2. Knowledge

To examine how *knowledge* affects our research, we must first discuss the way our mazes are created, and some terminology to accompany it. A common term used

within this area of study is *procedurally generated mazes*, which are simply mazes that are created using an algorithm. The main avenue we are attempting to learn about is real time generated mazes which are created algorithmically, which lends itself to the concept of procedurally generated. If we create a pre-generated maze, it is fully decided before the user explores it using an algorithm. It too is considered procedurally generated, and thus, both pre-generated mazes and real time mazes are a subset of the procedurally generated mazes.

To discuss the concept of knowledge pertaining to the study with rats, we can look at our options and determine how it applies to our intentions. Having prior knowledge about the maze, as per the definition of *knowledge*, requires that the maze is already generated. This causes problems with the intentions of our project, since we create our mazes in real time. This is an issue, because if the maze is created in real time, we are not able to have the “rat” examine it before traversing it, since there will be nothing to examine. Pre-generated mazes solve this problem for us, since it is more representative of what the experiment with the rats delved into.

However, if we indeed utilize a pre-generated maze rather than what the researchers in the study used, what would be the real difference? This is where we can discuss the impact of our procedural generation and how it impacts this study, regardless of whether it is real time or pre-generated. If the maze is procedurally generated, this means that we can cater every turn and path to what the researcher intends to discover. This is a powerful tool that is indeed possible in the real world with rats, but would be much more time consuming to implement for the infinite possibilities of mazes. With a pre-generated maze with respect to this project’s description, we can avoid building times by creating a maze rather quickly based on some predefined values, which would be a great stride in the concept of *knowledge*.

If we provide a pre-generated maze to humans, there could be an impact on how efficiently they solve it, which could be something worth exploring. The only issue with this concept, is that it is entirely reliant on the creation of a pre-generated maze, whereas the stipulations for this project are limited to real time at the moment. While it would be a large task including the creation of pre-generated mazes to our project’s features, it could provide a great impact to this field of research and is worth considering.

23.13.3. Reward

Our next variable is the *reward*. As the name implies, this is simply the reward received for reaching the end of the maze. In the research, it was found that if the rat

knew what the reward was, their performance in completing the maze was greatly increased. The inverse is also true, where researchers found that removing the reward from the maze decreased the rats' performance. For example, if there was food at the end of the maze, the rat would be able to smell it which would not only help guide it in the right direction, but also incentivize it to work harder, since it knows what lies at the end of the maze.

With the intentions of how we choose to implement the maze in this project, this would pose a few issues. The most glaring issue is the medium we are choosing to have users traverse our mazes. Since they are digital, we cannot stimulate any of the senses besides sight and hearing. A solution to this issue would be providing some sort of scent emitting device, but that starts becoming a bit too complicated for the intentions of this project. Thus, we are limited to only sight and hearing, and our *reward* must be applicable to one or both of those categories.

It is also important to note that regardless of whether the maze is done within VR or a simple monitor with a keyboard and mouse, our limitations are the same. We encounter problems simply due to the digital medium.

We can note more of these limiting features and try to converge on some common ground between the two:

- Visual rewards
    - Must look enticing
    - Should be usable or be redeemable for something usable
    - Should provide the user with a sense of relief or gratitude
    - Should have some sort of visual indication of where it is located
    - Example: a very pretty or astonishing visual sight, only achievable in the digital space
- Auditory rewards
    - Must be nice to listen to
    - Should have auditory indication to help track the location
        - Spatial audio would be helpful to achieve this effect
    - Should be useable or redeemable for something usable
    - Example: a break from a bad sound heard throughout the maze, or a piece of their favorite music.

After listing these features down, it is unclear how well these two would converge towards one idea. It seems that the features of visual cues are disjoint from the features of auditory cues. The closest conclusion that could be derived would simply be to overlap the best of both, or choose which of the two the researcher would like to test.

The implementation of the *reward* in the digital space does not seem to be as impactful as a reward in the real world, so it is tempting to simply exclude the concept from our intentions with the creation of this project. If that is chosen, perhaps it would be helpful to look towards solutions within the real world. However, if rewards are provided in the real world alongside some verbal instruction, this would fall out of the hands of the individuals creating the project, and thus, we have to conclude our discussion on *reward*.

If we choose to invest time and resources developing some concept of a reward in the digital space, it must be worthwhile and impactful, which would likely be seen during the testing phase of this project at a later date. This poses an issue, since we would be using a great deal of resources on something that may not be too helpful.

In many other business models, virtual rewards exist in the case of virtual currency applicable to the application. Seen in many video games, players earn “experience points” based on their performance or time investment. A similar idea can be seen with video game currency. In our case, we do not have any use for the user to earn any means of “experience points” or “currency”. Creating an entire subsystem and ecosystem for the sake of the *reward* variable is out of the scope of this project. Thus, for the sake of time, it would be smart to exclude a reward, as the impact compared to the cost to implement it does not seem to be useful.

23.13.4. Drive

The last variable to cover within this research is *drive*. This pertains to the physical state of the rat within the maze. For example, a hungry or thirsty rat was found to be more inclined to finish the maze, compared to a satiated rat. This idea is not too applicable to our case, since it correlates to the physical state of the individual using the maze. As the team responsible for the virtual realm of this project, we have no impact on the biology of the individual testing. That would be entirely reliant on the researchers to investigate the current state of the person testing. As seen with the rats, this includes many situations, such as:

- The hunger levels of the rat
- How thirsty the ray is

- The sex-drive of the rat

Considering that this team is strictly involved with the creation of a digital interface, it is obvious that these states are nothing that we can control.

The only way to achieve any sort of effect that stems from the *drive* variable would be to incorporate a variable to extend the maze for a long period of time. This would create an issue for the person completing the maze, as they would get fatigued. As a researcher, what they provide to the individual testing is critical to this component of testing, as the researcher can choose to either help or hurt the person testing. For example, if the person exploring the maze gets tired, it could be an option to deprive them of nutrients until they complete the maze, or fail in their attempts. This, of course, would be done with the discretion of the tester, and the choice of the researcher as to whether they would like to implement such an idea.

Similarly to *reward*, it seems that we are approaching a dead end, where the concepts here are entirely applicable to the real world and do not transfer well to a digital environment. The intention of providing a maximum time is already intended, so that would not change much in regards to the implementation of our software, hence, we have reached another conclusion. *Drive* is another variable in which it would not be useful to consider, as the project does not lend itself to impacting it correctly or efficiently, thus leaving it in the hands of the researcher.

23.13.5. Reward-Value

As seen prior, these 3 concepts are all independent from one another. During our discussions, it was quite clear that *knowledge*, *reward*, and *drive* are separate concepts, but there is one more aspect that can alter the outcome of the maze that happens to be dependent on all of the ideas mentioned prior. This concept is known as “reward-value”. This pertains to how impactful our *reward* is, based on the *desire*. An example of this could be a hungry rat and a satiated rat. If the reward is known by the rat to be food, the hungry rat was found to complete the maze faster than the satiated rat. Similarly, if the rat is thirsty and the reward is known to be food, the rat’s performance in completing the maze will decrease.

While this information is important and helpful in the discussion of mazes pertaining to rats, when determining if they are helpful to the creation of this project, it would be helpful to look towards the dependencies of the *reward-value*. We know that *knowledge* is possible, but an unlikely feat due to our time restrictions. With both *reward* and *drive*, the concepts discussed are too harshly rooted in the physical realm, it would

be difficult and sometimes near impossible to portray effectively aspects of those variables. If it is known that *reward-drive* is a direct derivation of both *reward* and *drive*, and both of those are known to definitely be problematic, we can conclude that *reward-value* too would be problematic. Although the *knowledge* portion is plausible, it is pointless to even consider it, since the other two factors still remain unsolved and unlikely to be considered due to time.

It is also important to note that all the variables presented in this study are independent from the parameters that affect our algorithm. These concepts are more on the basis of how the research would be conducted. The purpose of analyzing rats in mazes is to determine if there is more that can be implemented into the project, allowing for more assistance when the researchers begin testing. Based on the conclusions found, we can come up with the following statements about the different variables:

- Difficult to implement
- Does not lend itself to the virtual environment
- Time consuming when there are other priorities during development

Attempting to stay in the virtual space limits our ability to create new possibilities regarding these variables, but entering the physical world is not feasible for the scope of this project. It is an unfortunate dilemma, but the extent of our possibilities would be with pre-generated mazes and the possibility of simple rewards at the end of the maze. Everything else would be the responsibility and under the jurisdiction of the researchers in their physical environment.

23.14. Background and Introduction

There are many algorithms for maze generation. A significant portion of them can be broken down to some variation of the recursive backtracking algorithm.

23.14.1. Recursive Backtracking Algorithm

In the recursive backtracking algorithm, the maze is made up of square cells in a grid with walls along every shared edge by default. The algorithm works by starting at a cell and randomly picking an unvisited adjacent cell, traversing to it, marking it as visited, and repeating until the current cell has no unvisited neighbors. At this point, the recursive function calls return up until one of the cells has an unvisited neighbor that it can visit. Each time a traversal is made from one cell to the next, the wall between them is removed.

If you repeat these steps until every cell has been visited, you're left with a maze. This is one of the simplest methods for generating mazes, and what you can control about the final maze is very limited – just the size of the final maze. Because the algorithm randomly chooses a cell to visit next, and it repeats a different random choice each step, the overall maze is random. With random mazes, it is most likely going to have unfairness and high variability in its traversal, so this algorithm by itself will not work for our purposes.

```
def genMaze(gridOfCells):
    genMazeRecursive(gridOfCells[0][0])

def genMazeRecursive(cell):
    cell.visited = True

    # Get a list of valid, unvisited neighbors
    neighbors = getUnvisitedNeighbors(cell);

    while (len(neighbors) > 0):
        # Randomly choose which cell to visit next
        nextCell = random.choice(neighbors)
        neighbors.remove(nextCell)

        removeWalls(cell, nextCell)
        genMazeRecursive(nextCell)
```

(A code snippet showing an example implementation of the recursive backtracking algorithm)

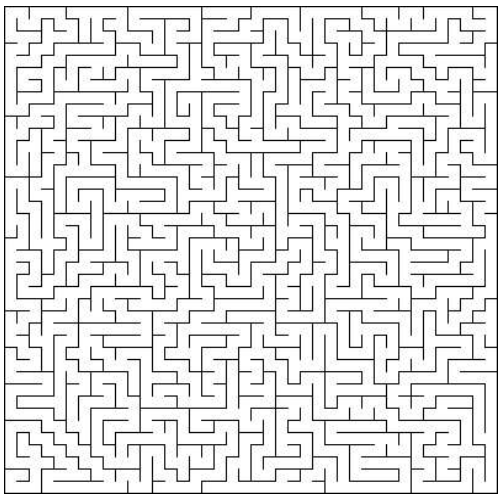


Fig 40. (A maze generated from the recursive backtracking approach with random cell selection.)

It is worth the time to review this algorithm and understand how it works before trying to create a more complex maze generation algorithm. There are, in fact, a few additions we could make to this algorithm to allow for a more fine-tunable maze generation. These ideas might be applicable to the final algorithm, so exploring them in more depth seems useful.

23.14.2. Variations on the Recursive Backtracking Algorithm

For example, one way one could tune this algorithm is by specifying how to pick the next cell to traverse to. Rather than randomly picking a cell to traverse to, one could introduce a variety of selection methods by choosing with some non-uniform distribution of their choice. This idea could allow for mazes with long vertical hallways, by increasing the probability of choosing an “up” or “down” cell in comparison to “left” or “right.”

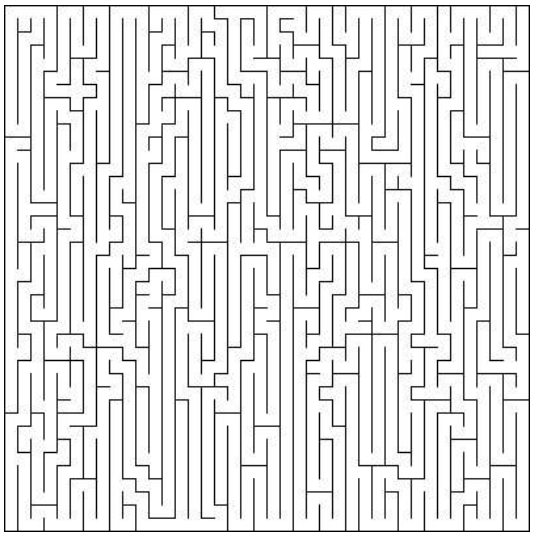


Fig 41. (A maze generated from the recursive backtracking approach with non-uniform distribution selection.)

Furthering this idea, one could keep track of the direction of the previous choice of direction, and vary the decision for the next cell based on some function of that value. With this, you could have long vertical *and* horizontal hallways, by favoring cells that are in the same direction traveled as the previous. One could even practically eliminate straight hallways by never choosing the same direction twice, unless it was absolutely necessary.

As you can see, additions onto the traditional recursive backtracking algorithm allow for a fair amount of control over what the generated maze will look like. However, at face value, it does not seem to have the capacity to eliminate the unfairness we are hoping to get rid of.

Our hopes were that impossible spaces would help reduce this unfairness by creating mazes where each path led to the end with similar lengths and number of turns. So, with that in mind, we searched for ways to generate mazes that allowed for overlapping segments.

23.14.3. Weave Mazes

This research led to us finding *Weave Mazes*. A Weave Maze consists of paths that are able to pass over and under other paths, which is similar to the idea of impossible spaces, which can have overlapping segments.

The rules of a weave maze are as follows:

- Passages may only move over or under each other when they are perpendicular to each other.
- A passage may not terminate when under or over another passage.
- Passages may not change direction when over or under another passage.

This set of rules generates mazes that contain paths that can pass over and under each other, which is analogous to what we want our impossible spaces to achieve.

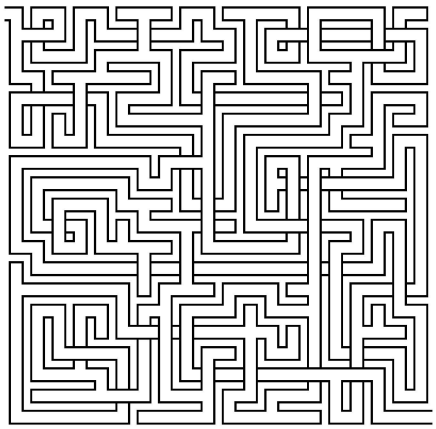


Fig 42. (A weave maze.)

However, this idea falls short because only perpendicular paths may cross over and under each other. It is completely in the rules of impossible spaces to have two parallel hallways map to the same location, which is not possible with weave mazes. So, while weave mazes seem like a good start, they don't actually have the effect we are looking for.

23.14.4. Pre Generated Mazes Fall Short

Interestingly, it seems that any pre-generated maze will always have variability because of the fact that the start, end, and walls of the maze are decided independently of the user's movements. Thus, with multiple paths from start to end, each with varying length and number of turns, the user will always be able to pick suboptimal and potentially misleading paths.

So, we are led to a solution involving a dynamic maze, which while more complex, will hopefully offer us the invariability of player paths and impossible spaces that we are hoping to achieve.

The main idea of the algorithm is centered on placing *pieces* of the maze in front of the path of the player and removing those pieces if the player decides to backtrack. Also, the pieces may spatially overlap with each other, but only one piece is traversable by the player at once.

In fact, if the player should have no idea at face value that the piece they are traversing over overlaps with another. The only way to detect if this is happening from the point of view of the player is by reasoning about where they are spatially based on the path they took to get where they are currently located.

This is exactly the *impossible space* effect that we want to achieve. What follows is an overview of the algorithm and a few various considerations we have to make before deciding on an exact implementation.

Firstly, a maze consists of a combination of *pieces*, which are gridlike structures, of which there are two types: *junctions* and *segments*. A junction is simply a point of connection between one, two, three, or four paths. If a junction connects only one path, it looks much like a dead-end. These junctions look like this:

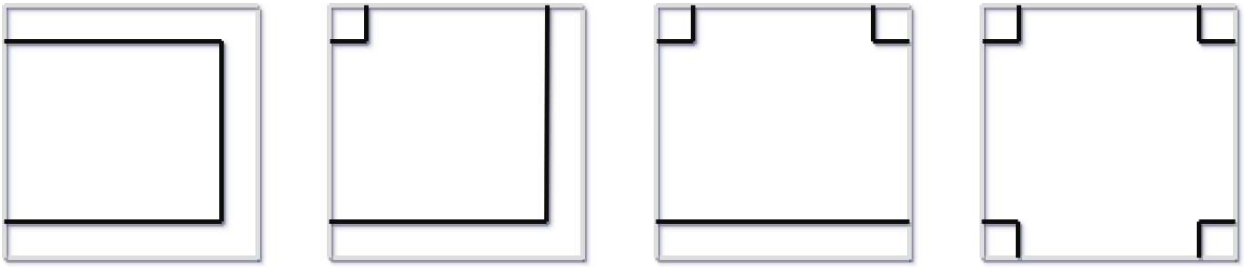


Fig 43. (A 1-junction, 2-junction, 3-junction, and 4-junction, from left to right.)

Note how the rotation of all but the 4-junction is important for determining which sides can connect to another piece.

A *segment* is simply a connection from one piece to another piece, in a straight, one-cell path. It would look like this:

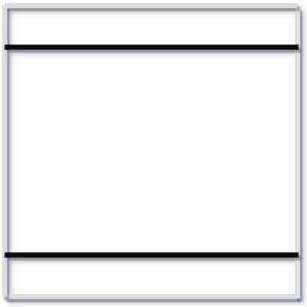


Fig 44. (A segment piece.)

Together with junctions and segments, you can create any grid-like maze, so it will be crucial to understand how the pieces fit together before thinking about the dynamic algorithm that we are trying to create.

23.15. Algorithm Overview

The algorithm has many goals, as previously mentioned, and to achieve these, many design choices need to be made. In order to achieve a fair maze, it is necessary to dynamically generate the maze as the player moves throughout it so that their choices can influence the structure of the maze. If their choices were taken into account in just the right way such that the maze ends near the *desired end*, then we could enforce the fairness we are looking for.

But before moving further, we need to define what a *desired end* is. This will be given by the user, and can be either a length of time or distance that the player must achieve before the maze ends. If, for example, the user wants the player to complete a maze that takes approximately two minutes, then the maze will generate in such a way that once two minutes have passed, it will look for a spot to place the end. Similarly with distance, once a player has traversed a certain number of segments and junctions, the maze will look for a place to end the maze.

So, the general idea of the maze is as follows: pieces (junctions or segments) are placed at the end of already generated pieces as the player traverses through the generated pieces.

If, by random chance, the player decides to backtrack up the path they took, then the space that was once occupied by that path will become free space for the maze to generate over. Note that this does not mean if the player were to come down this path again that it would be generated any differently, as the generation only

happens the first time and every subsequent time the player accesses this area it is simply loaded from memory.

23.15.1. The Tree

In order to keep track of each path and what to render versus what to store in memory, it is crucial to have a data structure to store this information. The most clear solution involves using a tree structure. In this tree, each node would represent a junction, and each edge between two nodes would represent either adjacency in the maze grid or the segments connecting them.

Thus, as the player traverses down a path, the tree would expand to accommodate the newly generated paths, and likewise we keep track of which node in the tree the player is located at. To determine what to render, we could simply look at the path from the root of the tree to the current node, which would represent the path the player took from the start of the maze to where they currently are.

In a traditional maze, the presence of this tree structure would not be necessary. However, because paths within our maze may overlap, it is important to tell which path a certain cell within the grid should be a part of. In traditional mazes, a grid cell could be a part of exclusively one path, but because we are allowing for paths to overlap, then any cell may be occupied by *multiple paths at once*. Thus, we must be able to determine which path to render, which logically leads to the requirement of our tree structure.

23.15.2. The End

Another important design decision to make is how to determine where to place the end goal of the maze. We've come up with two different methods, a static method and a dynamic method.

In the static method, the end is defined before the player begins traversing the maze. This implies that the maze must *guide* the player towards the end in such a way that is first: always fair, and second: it does so in the required amount of time or distance necessary for the desired ending to be met. This method requires some mechanism of guiding the player, which we will discuss now.

The guiding mechanism would work like so: when placing a junction piece, the orientation of open branches should be directed towards the exit. Likewise, junctions

with many branches should be picked less so that the algorithm tends to generate fewer paths with a stronger direction towards the end, rather than many sprawling paths.

The static method is difficult to implement in a way that guarantees an end close to the desired end. This difficulty comes from the inherent randomness of where the player is, the state of the maze, and the challenge of how to determine when to start guiding the player.

In the dynamic method, the end is *generated* as one of the pieces during standard maze generation. The end is only generated once the conditions for a desired end is met, whether it be distance or time based. This method is much simpler, conceptually and practically, than the static method because there is no coercion that must be done with the algorithm to get it to *always* lead the player to the end when the desired end is met.

Despite the difference in the challenge of both of these methods to work, if both methods work flawlessly, then the intended effect is the *exact same*. In the static method, from the point of view of the player they cannot tell the maze is guiding them towards the end, and it feels as if they find the end naturally. Likewise in the dynamic method, the end of the maze naturally appears in front of them after they search for a given amount of time or distance. In both methods, they appear to do the same thing from the point of view of the player.

Thus, the main concerns on which method to choose comes from other factors. One major factor is whether or not we want to give the user the choice on where to place the end of the maze or not. With some consideration, we felt that because the maze is dynamically generated, there is no need to place the exact location of the end, as to the user it will be mostly meaningless because of the lack of context of other parts of the maze. If the user wanted to specify it at a certain distance from the start, they should set the desired end, rather than the location of the end of the maze.

Another factor is the ability to guarantee the desired end is met in all cases. With the static method, the algorithm must always be able to guide the player to the end. Though this is theoretically possible, when it is actually implemented there could be imperfections and edge cases causing the guarantee to break down. However, with the dynamic method, we can always guarantee the desired end is met to the best ability of the generation algorithm.

The runtime of this algorithm would be $O(n)$ where n is the number of pieces generated. C Because picking a piece to generate and placing it is done in constant time, then it follows that the runtime is linear with the number of pieces generated.

The space complexity of this algorithm depends on the number of paths generated. If we use the tree structure to maintain a record of each path, then at worst the space complexity would be $O(4^n)$, which is the case that each cell is chosen to be a 4-junction. At best, the space complexity would be $O(n)$, which occurs when the maze is generated to be a single path of segments, where *n* is the number of segments in the path.

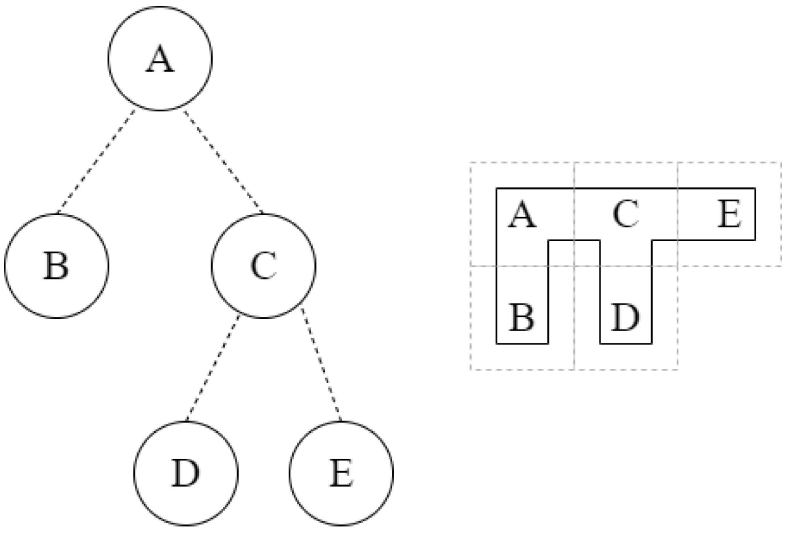


Fig 45. (A diagram showing the relation between the tree structure and maze structure)

23.16. Parameters

Users may want to control certain aspects of the maze. We want to accommodate this by adding a number of parameters that influence the maze generation algorithm. Some examples of what a user might want could be a maze consisting of very long hallways with few turns or maybe a maze consisting of very short hallways with many turns. They may also want to specify a starting sequence of junctions and segments so that the maze always starts in a predictable way. Overall, there are many aspects of the maze which the user might want to control, so we introduce a number of parameters to achieve this.

There are three types of parameters, *starting* parameters, *generation* parameters and *skew* parameters, each with a different purpose.

The purpose of starting parameters is to allow the user to define a sequence of junctions and segments that the maze will *always* begin the generation algorithm with. For example, if the user wants the maze to begin with five straight segments followed by a 4-junction, then the user would specify that in the UI, and the maze would use those parameters to override any random generation with those.

Similarly, generation parameters define a specified number of *x*-junctions and segments that must be met within the maze, in any order. For example, if a user wanted to have at least 50 segments, they would be able to set the number of segments within the generation parameters to 50 and any maze generated with those parameters would have at least 50 segments. This differs from sequence parameters as it does not specify the order in which pieces appear, but only specifies the minimum number of pieces of that type to generate.

Generation parameters are useful to the user when they want a maze that always exhibits a certain type of behavior, such as many segments or many junctions of a specific number.

Conversely, skew parameters determine how frequently a type of piece is picked in contrast to other types. There are many skew parameters to discuss.

23.16.1. Skew Parameters

Firstly is *straightness*, which ranges from 0 to 1, and determines how frequently to pick a segment piece over a junction piece. If *straightness* is set to 1, then the relative frequency of pieces to junctions will be very high, with the resulting maze consisting of far more straight segment pieces compared to junction pieces.

Note that a *straightness* value of 1 would not mean there are no junctions, as this would make a maze-like structure impossible to generate, but the frequency would be highly in favor of segment pieces. The same applies for a *straightness* value of 0, which would generate a maze consisting of very many junctions and few segments.

Secondly is *length*, which also ranges from 0 to 1. This parameter determines how likely it should be to pick a segment piece to be placed adjacent to another segment piece. With a high *length* value, the resulting maze would consist of long hallways interspersed with normal junction sections. This differs from a high straightness value as it only affects the frequency of picking another segment *after already picking* a segment beforehand. The probability of picking a segment after

picking a junction is unchanged, which results in a slightly different maze than one with simply a high *straightness* value.

Thirdly is *impossibleness*, which ranges from 0 to 1. With a high *impossibleness* value, the algorithm favors guiding the maze into areas that already had a path before the player left the area. This parameter leads to more overlap in paths, which may be useful if the user wants the player to experience impossible spaces frequently.

Lastly is *junction_frequencies*, which is a vector in $\mathbb{R}^4$ that describes the probability of picking a 1, 2, 3, or 4-junction. The first entry would correspond to the frequency of 1-junctions, and so on. This vector should be run through the softmax function to ensure each entry is a valid probability, which would be done before using it so the user does not need to specify a vector with exact probabilities. With a *junction_frequencies* vector with a high probability of picking a 1-junction, the number of dead-ends in the maze would be high compared to the number of 2, 3, and 4-junctions.

This parameter is useful if the user wants a maze that has a high number of specific junctions in relation to others, which could be useful for defining the relative difficulty of a maze. A maze with more 3 or 4-junctions could be seen as more difficult than one with less, for example.

```
def maze_algorithm(start_sequence, generation_parameters, skew_parameters):
    ...

    # Pick next piece based on start sequence
    if (start_sequence != [] and sequence_idx != len(start_sequence))
        next_piece = start_sequence[sequence_idx]
        sequence_idx += 1
    ...

    # Pick next piece based on generation parameters
    for piece_type in piece_types:
        if num_generated[piece_type] <= generation_parameters[piece_type]:
            choices.append(piece_type)
```

```
    # Don’t pick a new piece if we already picked one from the start
sequence
    if next_piece != null: next_piece = random.choice(choices)


    ...


    # Pick next piece based on skew parameters
    if next_piece != null: next_piece = random_piece(skew_parameters)


    ...
```

(A pseudocode snippet depicting how each parameter influences piece choice)

23.16.2. Limitations

Before moving on, it would be beneficial to go over the limitations of the parameters and some edge cases.

Firstly, the starting parameters may be faulty if a user specifies a maze that is longer than that of the desired condition. In this case, the placement of the end of the maze should not overwrite the starting parameters. This is because it is assumed that the user's most strict requirement is the starting parameters, as it is unlikely the user does not intend to use the entirety of the starting sequence, given they had to set it manually.

The same could be noted with the generation parameters. If the user specifies a number of pieces that exceeds the conditions of the desired ending, then it will be assumed that this is intentional and the ending will not be placed until the generation parameters have been met *and* the end conditions have been met.

As mentioned with the *straightness* parameter, the extrema of each parameter does not necessarily imply that *all* influenced pieces will be what that extreme implies. For example, a *straightness* value of 1 does not imply all pieces are segments, and a *length* value of 1 does not imply there will only be segments following segments.

To achieve this, the range from 0 to 1 is *mapped* to a new value between 0 and 1 that represents the actual probability of picking what the parameter describes.

There are multiple ways to map to this real probability. Firstly, you could simply clamp values over or under a threshold to that threshold. For example, anything over 0.85 could be set to 0.85, and anything under 0.15 could be set to 0.15. This would mean 15% of choices determined by the parameter, when the parameter exceeds either end of the threshold, would still be chosen.

This method is simple and easy to implement, but a problem arises when the user attempts to set a parameter at some value over this threshold and notices there is no difference in generation. For this reason, it's worth investigating for a method that results in a more intuitive result for the user.

A more user-friendly technique would be to have some maximum value for the real probabilities. Then, you scale the parameter value by the maximum value and offset by half of the amount lost by the scaling. This way, the parameter values from 0 to 1 are uniformly distributed from the minimum and maximum value of the real probability.

For example, let $range = 0.8$, indicating that the parameter value can have a maximum of 0.9 and a minimum of 0.1. Then, the parameter value *p* is mapped to the probability value *x*, where $x = p \times range + (1 - range) / 2$. Thus, if $p = 0.5$, then $x = 0.5$, and if $p = 1$, then $x = 0.9$, and so on. This method of mapping the parameter value to a probability preserves the uniformity of the parameter to probability conversion, so it is more intuitive to the user.

23.17. Algorithm Class Diagram

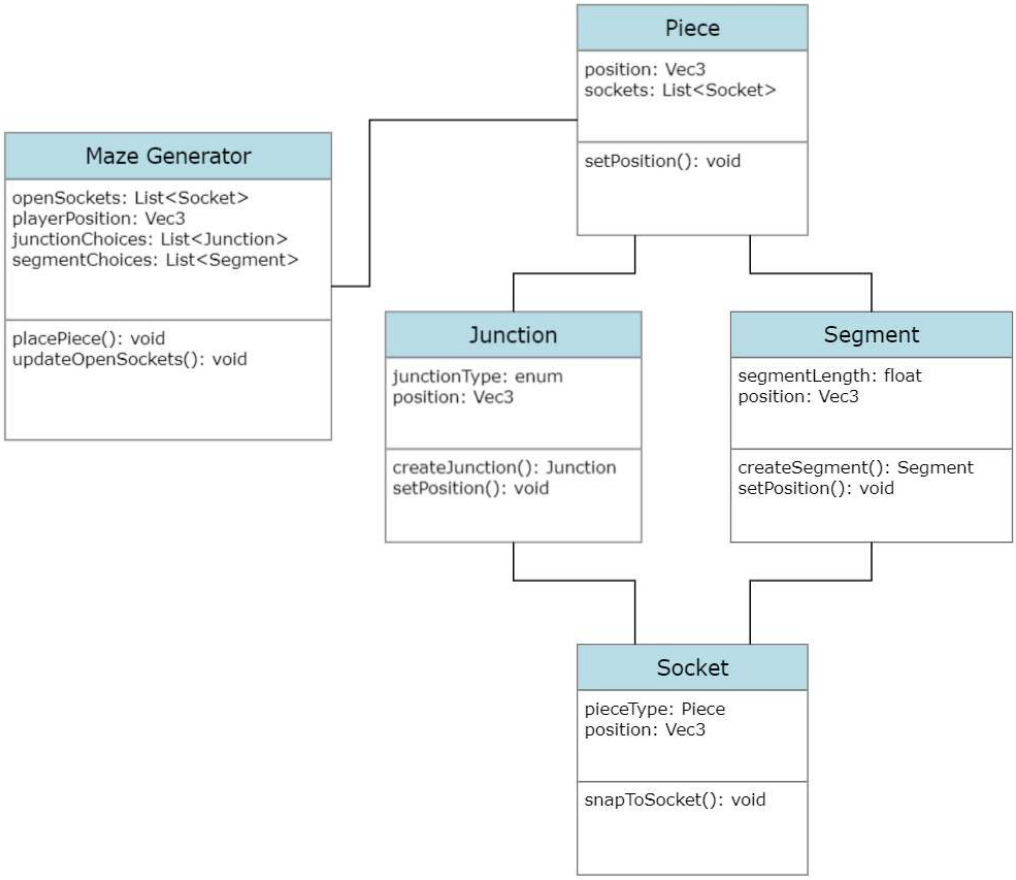


Fig 48. Algorithm UML Class Diagram

23.18. Implemented Algorithm Solution

23.18.1. Introduction

While we had many ideas going into this project, we differed quite a bit from the initial hypotheses. These changes were made to better fit with the intended goal of the project, which was to make a maze that can be arbitrarily sized and is not forced to align to a grid. This was always the intended, larger goal, but at the time of our initial planning, it was more challenging to conjure solutions to this issue. After working with the Unity editor, we were able to make those changes come to life.

23.18.2. The Different Components

When discussing the algorithm, it ultimately ended up as a couple of different components working together, typically in the following order:

1. Setting up a single piece
2. Stepping on the piece
3. Visibility calculation
4. Generation
5. Impossible space detection

Each of these concepts became quite a formidable task, so it will be discussed in great lengths to ensure a suitable representation of the work that was put into it.

23.18.2.1. Setting Up a Single Piece

A single piece has a lot of information that it needs to keep track of, but for now, we will only discuss the aspects necessary for the maze's generation. To begin, a piece must contain a trigger. This is what allows both the user and other pieces to collide with one another. Without this, there is no way for things to interact with one another, and thus, the maze is pointless, so a trigger is necessary. Doing this is simple and involves the base Unity collision systems that were already set in place. The only important thing is setting up the collision matrix so that different layers interact with each other correctly, which can be changed in the Unity project settings.

Additionally, each piece must have an ID associated with when it was spawned. This is useful for later for the visibility system and ensuring the final path taken by the user is correct.

Each piece also has access to a GameObject that represents the impossible space it is a part of. So within the maze, there are many pieces and each piece is connected to a shared impossible space. Within each impossible space, it has references to the pieces that are inside of it, as well as how many pieces are currently visible.

Additionally, each piece has a list of all of the other pieces it is connected to. So for example, in the currently discussed pieces, there are 4 directions. So if we let "up" be considered as "0", then in an array with 4 elements, the 0th element would be the up direction. This is extended to all of the other directions and must be unanimous across pieces. We hoped to abstract this to make it unnecessary to rely on directions, but it is quite a challenge considering the other directional reliant systems we have in place.

These are the fundamental aspects of a piece that cannot be changed much, and are critical to the other systems discussed within this section. There are a plethora of other components to a Piece, but these are the major ones. For example, with the aforementioned triggers, it is vital to determining what piece the user should be stepping on.

23.18.2.2. Stepping on the Piece

When thinking of the question, “Which piece did the user step on?”, there are a few factors to consider:

- What piece should the user logically be on?
- Are they interacting with two pieces?
- Did they teleport, which would skip pieces?
- How should skipped pieces be handled?

These are all aspects needed to consider to properly calculate where the user should end up, and how they ultimately interact with each individual piece. Let us first discuss the simple case of a user going from one piece to the next adjacent one.

This would primarily occur when using a linear locomotion method, for example, walking. Doing this poses no issues, as the user would exit one trigger and enter another. Since it is linear, it will not be skipping anything along the way and thus, a straightforward situation. However, there could be a situation that represents the following:

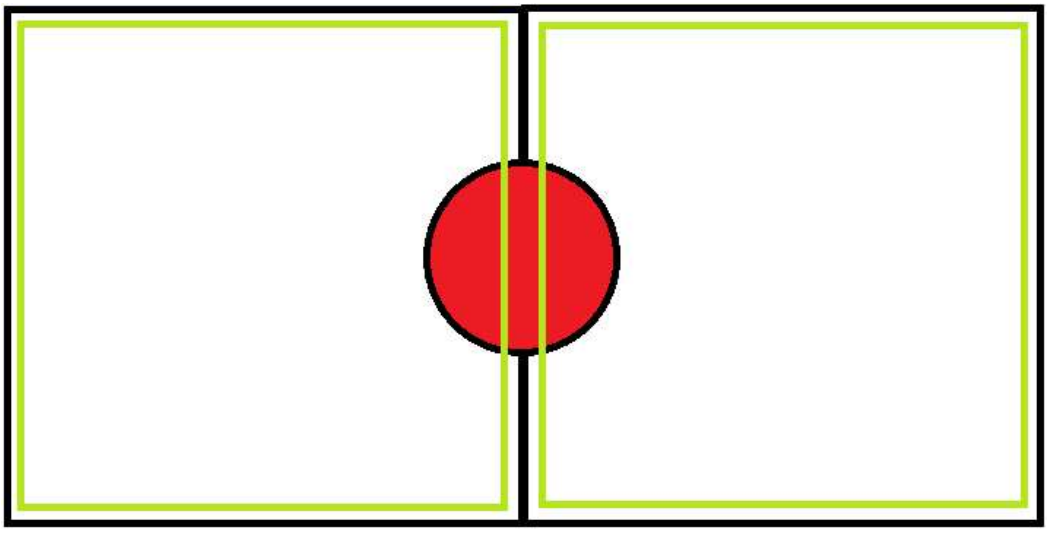


Fig 49: A player (marked in red) colliding with two pieces simultaneously.

Here, we see the player, which is represented by the red circle, intersecting the two green squares, which represent the triggers of each piece, at the same time. Here, we can run into the issue of not knowing where the user should logically be in. While not the most elegant solution, we opted to simply make the size of each trigger small enough so that it is impossible for a user's collision to interact with two simultaneously. With our testing, it proved to be the most straightforward solution and did not seem to cause any issues.

The one problem someone could pose would be, "What happens if you increase the size of the user?". With this edge case, it is a nuisance, but we are limited by either the size of the user or the size of the triggers. Regardless, future researchers can opt for whichever they see fit, as long as the triggers are at the center of each piece, and when a user goes from one piece to the next, it is impossible to avoid the trigger. If these conditions are met, we do not see this to pose an issue.

The other possibility is if the user utilizes teleporting locomotion. When doing this, they may skip pieces, which means we need to also consider those when on the journey to the piece teleported to. Doing this means we need to find the next piece and get all the pieces on the path from the current piece to the next piece.

However, the primary issue with teleportation is finding that next piece. If we simply have a Unity GameObject of where the user arrived, what can we do with that information to find the path from the current piece to that piece? If a user teleports from one piece to another, that next piece must be connected to that current piece, so we can simply run a graph traversal to locate the path the user must take to get to that next piece. While this is a good solution, it is slower than need be. Here, the maze has four fixed directions, so if we calculate the direction from the current piece to the piece teleported to, we only need to search the one direction!

The main question now is, "How do we find the direction?", which is nothing more than some geometry! If we consider the unit circle, we have a few directions associated with degrees. For example:

- "right" is 0 degrees
- "up" is 90 degrees
- "left" is 180 degrees
- "down" is 270 degrees

If we think about the problem in a 2D, top down space, we can achieve exactly this! The vertical aspect is not relevant here since all pieces occupy the same plane. So the steps to calculate the proper direction is as follows:

1. Subtract the end position vector from the start position vector.
2. Calculate the direction with arctangent of the x-axis and z-axis.
3. Convert the direction to the space of (0, 2π)
    a. Direction = (Direction + π) / (2π) * 4 % 4
    b. Here, 4 is the number of directions, and for simplicity this is 4, but this system allows for this to be expanded to any number of directions.
    c. For example, this would be able to calculate the correct direction for a piece with 8, 16, 32, 256, etc. possible directions.
    d. The number of sockets must align with the unit circle, thus the number of sockets is equivalent to 2^n, where $n \geq 2$ and $n \in \mathbb{Z}$.
4. Round the value received to an integer (it is either a whole number or near in the middle of two whole numbers)
5. If it was in the middle of two numbers, check which of the sockets is valid and round the number accordingly.

This process correctly provides the intended socket, given a start and an end vector. This logic can also be applied arbitrarily to other systems one might find useful within the scope of this project. This also can be expanded vertically by considering metrics such as the dot product with the y-axis, but for the sake of what was accomplished in this project, it was limited to simply the 2D xz-plane.

Since we have the direction, we can now traverse along that path until we arrive at the intended piece and store all the pieces along the way. So now we are ready to pass that information to the visibility system.

23.18.2.3. Visibility Calculation

After we have all of the pieces we stepped on, we give all of those pieces to the visibility system, which does a depth first search algorithm on each of the provided pieces. The conditions for stopping are based on if the current piece we are looking at, in the provided direction, has a wall. This means we cannot go any further and we can stop our search. Along the way, we also add on the pieces to the immediate left and right of each piece:

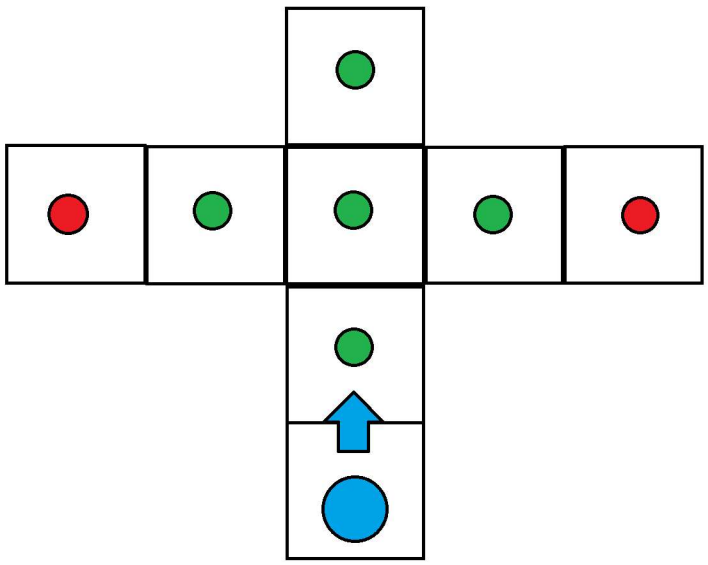


Fig 50: A user (marked in blue) going forward, with green dots representing the pieces inside the visibility, and red dots representing pieces outside the visibility

Here, we see the pieces that would ultimately end up being visible to the user. The adjacent pieces along the way are added for the situation of a user peeking around a corner without stepping into the next piece. Doing this ensures that the user never sees any blank spaces within the maze and is necessary for proper immersion, which is vital in a virtual reality application.

On the subject of blank spaces, in the case that the traversal does not hit a wall, but nothing at all, as in an empty space, we want to generate a piece instead, and after it is generated, continue the traversal.

This must be repeated for each of the pieces found in the previous step, since any skipped pieces also need to have their visibility calculated. To help with this, we store the visibility after this has been called for later use. We can be sure the visibility for a given piece never changes since all empty sockets are filled, and the pieces will never be removed, thus the visibility will remain the same and can be stored.

Once the visibility has been called for all the necessary pieces, we want to correctly set only the pieces visible to the player as visible. Doing this involves simple set arithmetic, where we take the current pieces visible and the new pieces visible. With these we correctly turn off and on the pieces that are no longer visible and the pieces that are newly visible, respectively.

In addition to this, we have the issue of the stack, discussed more in-depth elsewhere in this document. When a user does not take a path when given the choice to do so, we have to put that path back onto the stack in the correct order. We can do this

by getting all the pieces that have just gone out of the visibility of the user, and putting the back onto the stack, backwards, in the order they were generated. Similarly, if we encountering pieces that have been seen before, we want to remove those items from the stack when they reenter the user's visibility

23.18.2.4. Generation

If we have a blank space, as discussed before, we need to fill that spot with a new piece. While the actual creating of piece won't be discussed in length, the general idea can be elaborated on:

1. Spawn a new GameObject for our piece, using a modifiable prefab.
2. Take the next junction to be called from the stack.
3. Modify the prefab to conform to the junction type provided.
4. Add the necessary amount of straight segments along each direction

After all this is completed, we have properly spawned pieces and can continue the traversal.

23.18.2.5. Impossible Space Detection

After all our pieces are generated and the visibility is calculated for each piece, we have to deal with a few things:

1. When a piece is spawned, we must check if it collided with another piece.
    a. If it did, create an impossible space and add each of the colliding pieces to it.
    b. If the impossible space exists in one piece, add the new one to the existing space.
    c. If both pieces have an impossible space, merge the spaces.
2. Change the visuals of pieces (for the researcher) to reflect what the user sees and to visualize impossible spaces. **It is important to clarify, these changes are only made to the researcher's view, not the user.**
    a. If the piece is in an impossible space, add a color tint to it.
    b. If the piece is currently visible to the user, make it opaque.
        i. All other pieces not visible in the same impossible space are made invisible

c. If no pieces in the impossible space are visible, make all of them semi-transparent.

All of these changes allow for a clear visualization of what is going behind the scenes, and is quite useful information from a researcher's perspective.

24. Runtime creation

A pivotal component of our algorithm is generation of the path occuring at runtime of the program. The participant's choices at junctions dictates how components of the maze will be generated which can only occur at the runtime of a program. As such, the traditional way of pregeneration of "levels" like other common path or maze programs can not be used which introduces a layer of difficulty in programming the aVRage paths path generation. This means that a system in the Unity engine is required to allow for generation of objects to occur at runtime.

Upon further research, Unity's Prefab system allows us to solve the issue of generation at runtime. The goal of the prefab system is to allow developers to create predefined objects (setting things such as shape,size, physics, scripts, etc.) and save them into the scene. These prefabs can then be instantiated or cloned into the scene at runtime, which is a very common method of developing procedural levels in industry.

The path pieces will be prefab objects that can be cloned and can be reoriented to shape the maze. An important thing to avoid is generating future prefabs as children of the current prefab. The reasoning behind this is that each piece should be its own standalone object, and having a hierarchy can lead to some buggy edge cases. For example, if my current segment spawns a junction, this junction should not be a child object of the current segment because deactivating the current segment will also deactivate the junction.

25. Junctions

An important thing to consider is what to generate at each junction. As stated before, the pieces generated at the junction will be influenced by certain parameters being passed. All The most important cornerstone of the algorithm is specifically what pieces the function generates.

The first important thing is to approach what pieces are generated at each junction. The current plan is to have a prefab corner piece that allows for turning. This

corner piece can be rotated around to indicate a direction. The other prefab piece we will use is just a straight hallway. These two pieces can be mixed and matched to allow generation of any possible path.

A possible plan is to make the end of every segment a junction. This would mean we have two, three, four - way junctions where a two way junction is essentially just extending the current segment.

There are two approaches to handling junction generation regarding prefabs and segments:

1. All exits from the junction can be anything (another junction, or a segment), essentially all exits must recall the algorithm to generate the next piece
2. Exits have to choose from a set of pre-selected prefabs (a straight, a corner turn, etc)

The first approach is a hands off approach where anything can spawn at the exits, for example, two four-way junctions can spawn right next to each other. This approach aims for anything which just “rolls the dice” and spawns anything from any of the junctions to just a simple segment.

The second approach is a more directed approach where the path generation can be forced into a direction. The idea with this approach is at each exit there will be a generation from pre-defined pieces to be picked from, i.e. a corner that forces a path to turn or a straight segment that allows for continuing straight. The reasoning being this is relying on junctions to spawn as a method of continuously turning could lead to too much randomness.

To give an idea on why the first is too random, take a look at how many prefab pieces there are to choose from. There are three different junctions, 2-way, 3-way, 4-way and there is a straight segment. The 2-way junction has three different options (the entrance is already connected to the current segment so that leaves only one side to pick from three walls), and the 3-way junction has three different options. This brings us to 8 pieces total that can spawn at an exit, with a majority of them being 2-way and 3-way. This means that selecting a straight is highly unlikely and can cause us to generate too many junctions rather than straights.

Although, randomness could be considered fine if the researcher decides to pursue such an experiment and as such maybe a mix of two is the correct approach for the final algorithm. The idea for mixing the two could allow for some customizability for the tool set. For example, having a couple junctions selected from a predefined set chosen by the researcher and the rest of the junctions are truly random. This would give

more control over to the researcher as they themselves decide how they want their maze to develop. For example, incorporating approach two would allow the research the option to never generate a left turn for the participant, to test and check if the participant ever picks up on it during the experiment.

26. Ending The Maze

Finishing the maze triggers numerous different systems, and is the last step in the cycle of running participants through the simulation. The maze can be ended in two ways: the user finishing the maze, or the researcher aborting early. Both ways ensure that relevant data is still exported and that the simulation is safely ended for continued use of the toolkit.

26.1. User-Ended Maze

User-Ended mazes are the typical case, where the user enters the end piece. Upon entering the piece, custom functionality closes the wall behind them, preventing them from backtracking back into the maze. Particle systems that look like confetti and party horn sounds play, conveying to the user that they have successfully finished the maze. Cleanup functions are called to export data from telemetry and to save the maze to the respective folder. After a couple seconds, the scene changes back to the main menu.

It should be noted that end pieces are a special type of piece in the maze; whereas most pieces of the maze are generated using a junction as a template, the end piece does not use this template. In this way, end pieces can be configured with custom functionality - such as the functionality described above. In the current state, it isn't too obvious to participants that they can see an end area along their path. Whether this is good or bad is subjective to the researcher and experiment at play. However, a researcher that, for instance, wants to have the end area be more pronounced could easily do so by adjusting the end area prefab to their liking.

Furthermore, end pieces spawn at the end of every path that has reached the desired limit set by the researcher. This is one surefire way that the participant can learn that the maze is impossible. In other words, it's very possible that a participant never becomes aware of the fact that the maze is being generated on the fly as they navigate

through it, and thus would know no better that the maze is just a normal maze they are familiar with. However, seeing numerous paths lead to an end breaks this false reality in a very clear-cut fashion. As such, it may be desirable for special cases to be made in the generation that prevents this from happening. The specifics of this change are up to the researcher. One possible approach is to introduce a set of turns and hallways before an end piece is generated, making it more likely that a participant who goes down one of these hallways will be led into just a junction with one choice - which would be the end area. This prevents the user from possibly seeing numerous end areas attached to one junction, and gives less motivation for backtracking to another part of the maze that would then regenerate an end area.

26.2. Aborted Maze

Aborted mazes are those that result from the researcher aborting a simulation while it is still in progress. This is done from the live researcher dashboard. This might be necessary sometimes because of a user choosing to not continue traversing the simulation, because they become ill, because something goes wrong in the maze, or any other similar scenario where the participant wont be finishing the simulation. By aborting the maze, the same function that is called when a maze is ended by the user is called. This ensures that the telemetry manager outputs whatever data it collected up to that point and that no data is lost. A confirm modal is used to make sure researchers don't accidentally abort a maze that they don't want to abort.

27. Generation Amount

An issue that needs to be addressed is how far away from the player should path generation occur? The participant should never be able to see generation occurring in front of them, as such it is very important to provide an illusion of the path already being generated everywhere they look. There are multiple solutions to this problem but none of these come without their own pros and cons.

- Generate next junctions and segments based on participants current location.

The way this approach works would be that segments and junctions contain invisible “checkpoints” that when walked over will call the algorithm to generate the next couple of pieces for the path. This approach is very easy to implement code-wise but quickly runs into many in short segments and paths.

The issue with this approach is that it will work on very long segments but generation on smaller segments might generate a large enough path length to cover all

of the participant's viewpoints (aka they will just see an open horizon/drop off rather than a hallway stretching to the horizon). The way to fix this would be to adjust the number of next "lookahead" segments to generate based on the segment lengths that are generated. This just becomes a band-aid solution that will cause headaches down the line as it adds a layer of complexity and edge cases that will have to be carefully combed through.

- Generate based on the participant's point of view.

A more common solution in industry is to generate things upon coming into a player's view area. The main issue with this approach is it will be a more time intensive system to implement as it seems there is no built in field of view library for playable character objects. This means the system will have to be built from scratch. The idea here is generation is only performed when needed, which brings a list of pros and cons to this approach.

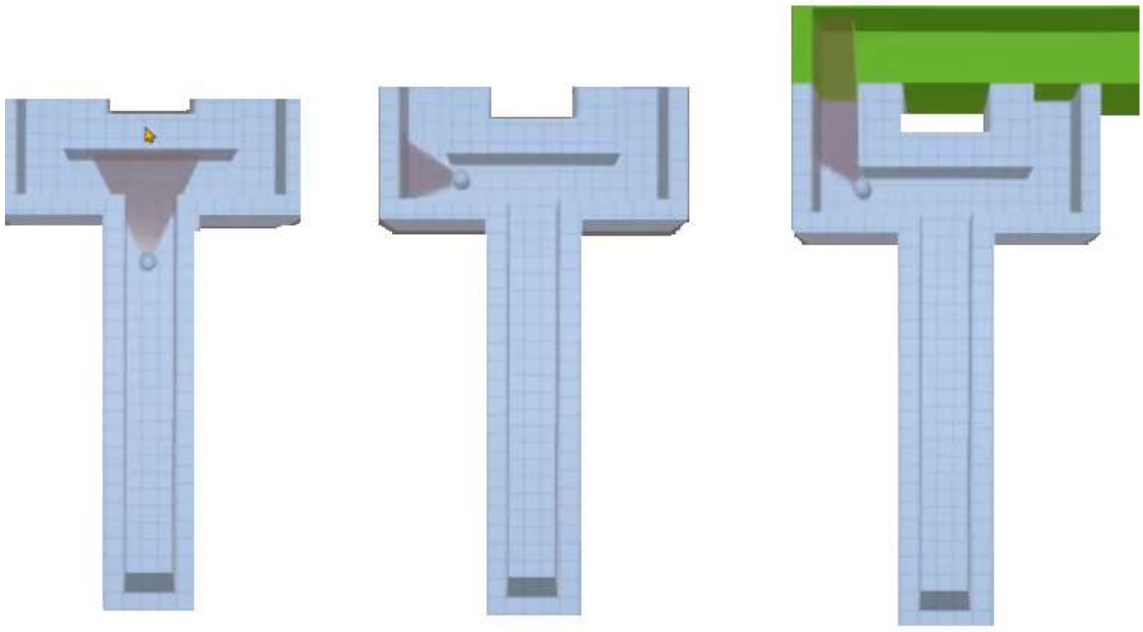


Fig 52. Image depicting field of view generation for the aVRage paths algorithm. Green represents a newly generated path.

First, an algorithm will need to be developed to create a that will generate a "cone" beginning from the participant's camera and extended to a certain length outward in front of them. As depicted in the picture above, the purpose of this cone is to detect any space that has yet to be generated. However this is where the first issue with this approach arises, how does one dictate what is empty space vs what is "air" in our segments.

To solve this issue, the space inside segments could be marked as “filled” and as such the algorithm will be able to tell about the inside of segments from empty space. Another edge case is when the camera is pointed upwards, the field of view cone will now move in the z-direction and since our object does not have a roof it will detect empty space.

There are two approaches to solving this issue. First approach is to generate an invisible wall at the top of each segment, this would allow for a sky box view and solves our issue. The second approach would be to ignore the z-axis in the coordinate system. The chosen approach would be the latter as it reduces space and computational complexity

An alternative solution is to implement a fog to limit the participant’s point of view. This allows for helping overcome some shortcomings in some of the above approaches, however this brings its own issues. The main issue is that now the tool set must now have a fog system, which means implementing such a system creates more workload for the aVRage paths team.

Fig 53. An example of fog depth blocking

Since the fog system is required then it also removes a layer of customizability for the aVRage paths tool, something that the team does not want to do. If a researcher wishes to conduct an experiment that does not include fog, then they will be unable to conduct such an experiment. As such, current players to include a fog system is mainly a customization option going forward and not a band-aid solution to path generation.

A key point to understand is the same approach that will be implemented for generating paths must be used for the despawning of paths. Since the only difference between generating a new path and reusing an old is loading from memory or creating a path it would make sense to reuse the same loading technique for both, such as if the “checkpoint” system is being used then it would work as a two way system where passing it one way would call a new path generation algorithm and passing it the other way would load the previously stored segment from memory.

28. Despawning/storing previous paths

An important aspect of the implementation for the impossible space is that we despawn and store previous segments. A proposed implementation is to activate and deactivate the object using the Unity library. Activating and deactivating the previously discussed prefabs works perfectly for when a participant decides to backtrack. Upon finishing/exiting a segment, the previous prefab will be set to an inactive state effectively disabling the object and the current segment/junction prefab the participant will be visible. This approach is amazing as it not only saves CPU cycles allowing our code to run faster, but also frees up the space in case another segment were to overlap and form an impossible space at that location.

The implementation of this would require references to previous segments/junctions that generated the current segment/junction, as well as references to current segments/junctions that the current object has spawned. This essentially creates a “tree” sort of approach where one piece holds references towards future segments and past segments, and backtracking is as simple as calling on the parent reference to be reactivated. When an empty space is approached the algorithm will first check to see if the has already been a previously generated path in that reference slot, if there is then simply re-active the object, if not then the generation algorithm is run.

The reason generated segments/junctions hold references to segments/junctions it has generated itself is because if the participant were to backtrack and then decide to continue the original path, the path is already generated and all that should be done is

to enable the previously generated path. The reasoning for this is the participant has already seen that part of the maze, and failing to re-render an already seen part of the maze will cause confusion not only for the participant but also for the researcher essentially killing the experiment on hand.

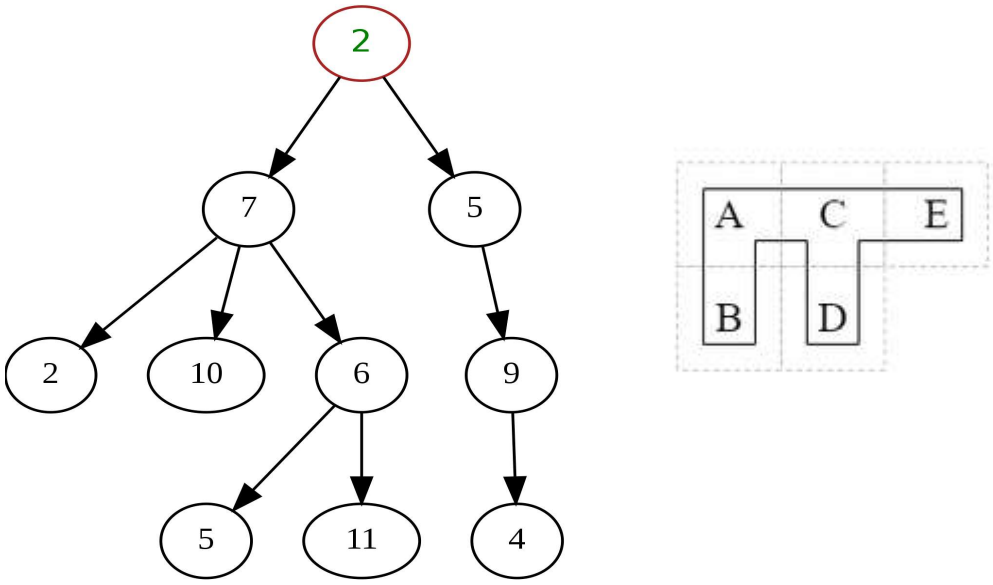


Fig 54. Reference Hierarchy (left) vs Actual path (right)

There is one major issue with this approach: Memory. Although the object is not being processed and rendered, it still takes up space in memory waiting to move from its non-interactable state to a rendered interactable state. The main culprit of this issue is storing direct references to objects as they are ready and waiting to be loaded in at any time. This is Unity's default behavior when disabling and enabling objects.

One approach to solve this issue is to use addressables. The use of addressables allows memory to be saved, however it comes at the cost of processing speed. A known issue with switching addressables is that now disabled objects are not really on "stand-by" mode and must be loaded in from memory. This latency can cause choppiness in that objects are not loaded in time or all at once. This could be a potential issue as if the field of view approach is used to decide when to reload a piece then if the prefabs are not loaded fast enough then the participant would see the prefab being rendered in front of them, which is something that needs to be avoided.

As such, if memory usage for the maze is lower enough then the use of addressables could be avoided and prefabs sitting in memory should be more than

alright. The tool set is pretty simplistic in terms of the amount of objects in a given scenario compared to other projects that are developed in Unity. If memory becomes an issue, the team has an alternative approach to fall back on.

29. Prefab Generation

One of the pillars for this toolset to work is the physical generation of pre-fabricated assets that can be used to show the visual representation of the generated environment. This section is heavily tied with the algorithm, but due to the time at which this document was due, approaches for implementing this essential feature had to be thought of, before a concrete algorithm was designed, which later one of these approaches was decided to work in tangent with the aforementioned algorithm. There were different paths to approach this feature, all of which had their benefits and caveats.

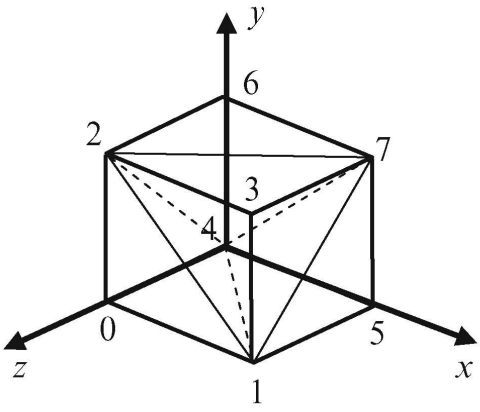
Two things that have to be mentioned are that while there were different approaches to the asset generation, and different algorithms that could complement one another, what both of these features have in common is that they will be generating at a distance where the user could not tell that the maze is being created, this goes back to giving the user the illusion of choice, so that they think that they are in a real static maze. The second thing is that the output of the algorithm will always be a coordinate in 3D space, and this section would assess how to use that coordinate into a visual element.

29.1. Plane-Based Mesh Generation

The first design idea that we had for our maze generation was to stray away from a grid-based generation algorithm, and look into a plane-based generation algorithm. The benefits of this approach is that it would give maze segments non-uniform sizes, which then gives more variability into the generation, it could also open a possibility for implementing less low-polygonal mazes that could have curves, or turns that would not only be 90-degrees.

While this approach might seem like the most visually appealing, it is the one that takes the most time to implement, with a lot of sub-problems that have to be solved for it as well. There are three basic things that make up a mesh in terms of data: Vertices, Indices, and Texture Coordinates; vertices, as the name suggests, are edge coordinate

points that make up a single triangle in the complex polygon that is the entire maze; indices work by indexing the previously mentioned vertices to tell the renderer where to draw the line and interpolation from one vertex color to another, this is what would actually end up making the triangle:



Vertex List		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

Triangle List		
i	j	k
0	1	2
1	3	2
2	3	7
2	7	6
1	7	3
1	5	7
6	7	4
7	5	4
0	4	1
1	4	5
2	6	4
0	2	4

Fig. 55. “Mesh Processing (Advanced Methods in Computer Graphics) Part 1.” what-when-how, https://what-when-how.com/advanced-methods-in-computer-graphics/mesh-processing-advanced-methods-in-computer-graphics-part-1/

Since this toolset is meant to have the ability to change the visuals of the virtual environment, texture coordinates have to be passed as well, these coordinates are used in a mesh-based coordinate system that goes from (0,0) to (1,1) that is used to map a texture image into the mesh. While Unity engine has two different APIs for constructing meshes—Built-in Mesh API, and the Probuilder API—there are a few concerns with the loading itself; first, if we are loading vertex points from the algorithm, we would have to be able to arrange them in way that they make a floor with two walls on each side, that is to say that the two walls are part of the same mesh, while also creating indices for each new triangle in the vertex; second, if the user were to “backtrace,” which triggers an impossible space generation, we cannot delete the entirety of a previously traversed segment in one go, since the whole generated maze is a single mesh, we would have to keep two parallel resizable arrays that keep track of the order of creation and assignment of vertices and indices to the mesh, and delete all of them up to the current segment that the user is in, which then incurs the task of also having to re-adjust indices.

One way to ameliorate this issue is to create a grid with very large dimensions, but with small distances between each point—for example: a 1,000,000 by 1,000,000

grid, with the distance between points being 1 centimeter apart—the algorithm would then only generate points in fixed grid positions. Although they are placed discretely instead of continuously, this makes it easier to program a way for the algorithm from where to delete old vertices, where to start plotting a new segment, and based on the length of the segment that is to be draw, it converts that length into a set amount of vertices in one axis required from the grid, as an effect due to the very small nature of distance between points, we can display segments of varying lengths, that *look* continuous, which make them harder to tell if they look similar to others. The potential pitfall here is that making such a large grid, specially if we allow the researchers that will use this toolset be able to customize the dimensions be bigger than the example mentioned, it could slow down performance, which is very essential in Virtual Reality applications, where even the smallest amount of rendering delay can ruin the experience for the user, which could also lead to low performance results when conducting tests.

29.2. Prefab Grid-Based Generation

While some of our group members were in the midst of figuring out elegant solutions for the first proposed option in visualizing our algorithmically made maze. We also thought of a discrete and simpler alternative to mesh generation, which is what also inspired the improved version of our first method. We would implement the use of a grid to generate prefabs of unit sizes.

Unlike the use of a grid to create pseudo-continuous segments, the distance between grid units would be much more spaced out, such as being 1 meter apart from each other, however, the dimensions of the grid would still be vastly large. This version of the mathematical construct allows for the easier placement of prefabs, since the algorithm could be developed to only generate in step sizes and load them at the center of dedicated points, rather than creating multiple points that correspond to the different vertices of a single mesh. To come up with a way to implement this method, we considered three different approaches to this already alternative solution to visualizing the virtual maze, although, one thing that is consistent with for all of the following approaches is that when a straight segment is created by the algorithm, whatever value it sets a straight segment’s length to be, the way the prefabs would be loaded would not be in a way that it would stretch the segment to the assigned number, this could cause texturing issues, and possibly custom model issues, warping and making them look unnatural, instead, repeated straight segments would be placed until the chain that is created from them reaches the length that was sent; this removes the issue of warping,

but removes the ability to make dynamic lengths by making them only have step-based values. With that clarified, we started discussing the following methods for the rest of the generation.

29.2.1. Four-Prefab Utilization

This concept is fairly straightforward, we used an analogy by using Lego pieces or sets, since their purpose is very similar to prefab generation or assembly into a three-dimensional virtual space. We would create four different prefabs—which we called “Lego Sets”—that hold predefined subsegments for the maze; these would be a straight, a right-angle turn, a three-way, and a four-way intersection.

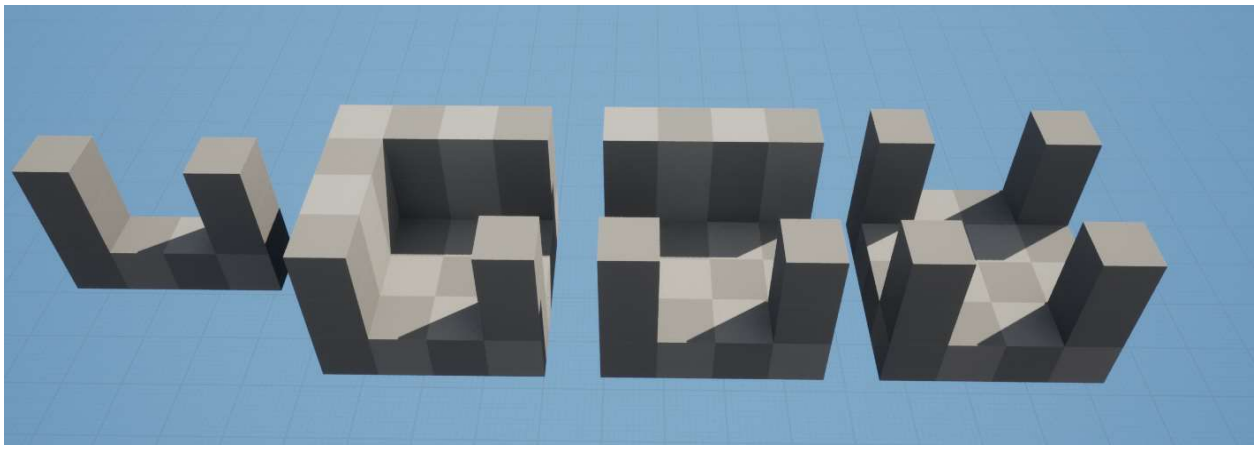


Fig. 56. From left to right: Straight, right-angle turn, three-way intersection, and four-way intersection.

As mentioned before, straight segments would be placed a set number of times based on how long the algorithm generates a straight segment. While the right-angle turn and the three-way intersection look like they only lead the user to specific directions, outside of the generation, the asset placement code would handle rotation of these two assets appropriately based on what was the previous piece placed and what the algorithm states it has generated, such as a left turn, or a three-way intersection where the two paths that the user can choose are either to keep going straight or left, rather than left or right, such as what the image above shows. Four-way intersections do not need to be rotated, since they are symmetrical and would make their place in the code redundant. Going back to our analogy, we see this way of implementation as placing entire “Lego Sets” to build our world. Another thing that has to be pointed out is

that walls are built into these assets; this is being mentioned explicitly due to how the next method of grid-based generation is used.

29.2.2. Three-Prefab Utilization

This might make one confused, since using the aforementioned four prefabs from above looks like the most logical and conceptually understandable approach to modularly creating a maze, even a lot of online blogs go about random generation for mazes like the previous method, but we thought we could optimize our storage consumption by reducing the amount of prefabs by one, and instead, we would use a straight and a four-way intersection again, but now we would have a wall piece.

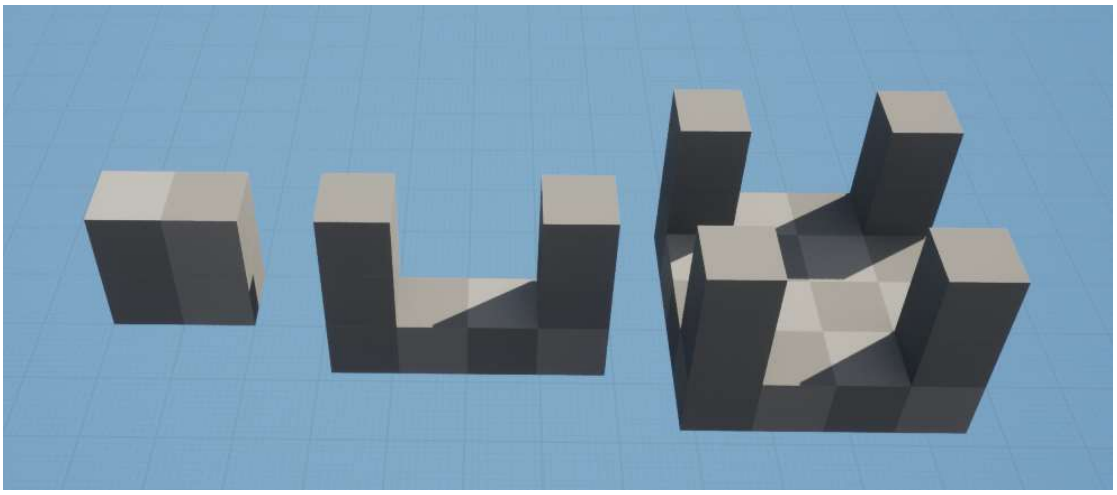


Fig. 57. From left to right: Wall, straight, and four-way intersection.

This would have a different method of implementation than the four-prefab utilization technique. For all types of turns and intersections, a four-way intersection would be placed, and based on what the algorithm generated, it fills out the walls required to make the desired pathway. For example; if the next generated path was three-way intersection the makes the user enter from the center and decided to either pick left or right, it would load the four-way intersection, and the fill out the north wall—of course, dependent on the relative orientation of how the maze has been generating so far and the endpoint of the previous segment generated. The analogy used here would be the same as placing “Lego sets,” and then adding individual pieces to modify it.

One complication that we realized while we were coming up with the concept for this implementation was figuring out how to place walls. The algorithm only generates what pieces should be spawned, and walls are part of the visual aspect of the maze, and to save a bit on computation, it would be preferable to only work on a

two-dimensional coordinate system. This means that placing the walls would possibly be in positions outside of the step size of the originally created grid, and working with objects that are outside of a discrete and repetitive space sounded like it could lead to imprecise placement, or overlapping constructs.

We realized that we were looking at our problem the wrong way; we were looking at our wall placement under the lens of a more mathematical or traditional computer science approach, where we would be working on developing our systems and data structures from scratch. Since we are working with Unity for our development environment, it has a lot of pre-made systems in place that would help us with this current problem, but also any future ones. The way objects exist in Unity is through the implementation of a hierarchical tree data structure where they could be nested. Objects can be a plethora of things, from a physical shape, to a sound emitter, and two important things about them is that they all exist in the three-dimensional environment, even if they have no contribution to the senses of the user, and most basic type of object is what people call, an “empty” object, where the most basic thing that it has is a Transform component, which corresponds to the object’s scale, rotation, and primarily, world-space coordinate position, which can also be converted into a local coordinate position based on treating its parent’s position (which is the game object above it in the tree hierarchy) as the point of origin. Therefore, by adding four children that are “empty” to the base four-way intersection, and setting their positions above the floor location, where walls would be placed, we can then reference those world-positions to load the appropriate amount of walls to create our desired shape that the algorithm created, and since the walls have just been loaded in code, they can be easily be set as new children for the segment, which is convenient for us if we have to delete the entire segment. This also allows for wall pieces to be as tall as the researcher wants it to be, since it is hard-modeled with the floor pieces.

With now the concept of using set coordinate points to spawn walls, this led us to creation of our third and last grid-based implementation method, and since the naming pattern has been concise, the name for it is of no surprise.

29.2.3. Two-Prefab Utilization

With this new usage of loading coordinates, we began discussing how much we could abstract our assets in order to have primitive shapes that use a bottom-up approach to create any shape the algorithm fabricated. We ended up creating two very similar shapes, a longer horizontal line that would make up floors, and a smaller vertical line that would be used for loading walls.

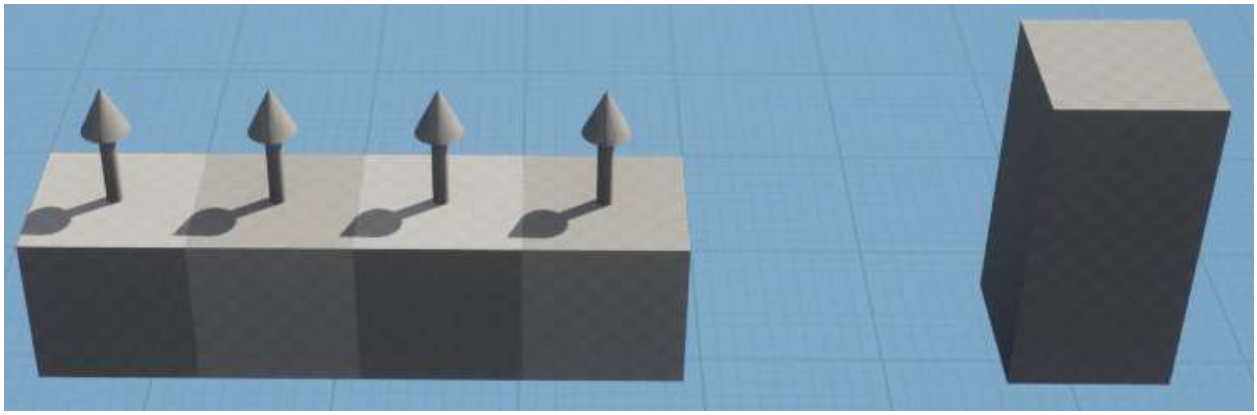


Fig. 58. From left to right: Horizontal floor line, vertical wall line.

The floor tile is longer because we want to give some space to the user to traverse in, and as for the reason why there are arrows on top of the horizontal floor line, this is supposed to indicate the empty child that holds the world-space coordinate to spawn walls in, but they would not actually be visible.

The way this implementation operates is very similar to the previous prefab utilization method, however, the wall piece has been modified to have two rows and one column as a template—after all, if the researcher wanted to change this asset, they would be able to—this is done because now the horizontal floor line has an empty child in every grid step amount. In theory every segment in the maze generation only creates walls at edges, which would imply that there should only be empty children on them, this then brings another problem where we would have to rotate and place the floor lines in such a way that all the empty children are at the edges; this was not possible when we had to consider pieces that have wall corner pieces, for example: if we had a four-by-four segment, with the upper left of that segment being the position of the zeroth row and the zeroth column, if we wanted to generate a right-angle right turn segment, a small part of the visual loading would require to have three walls pieces, one in the zeroth row, zeroth column, another in the zeroth row, first column, and lastly, first row, zeroth column, the wall piece with neighbors to its right and below should simply not exist, since there would be no prefab that could allow it to spawn; trying to fix this was originally sending us back to using three prefabs by having a single block floor piece with an empty child coordinate to cover this common edge case, in the literal sense. Instead we opted in having children for every grid step of the floor piece, this also allows the ability to create any desired segment by only rotating the floor piece based on the relative orientation of the maze generation and the previous segment that it had generated. We ended up realizing that this is the same as creating a subgrid and then loading whatever we wanted.

With this method of implementation, we discovered that this allows us to create more unique segments for mazes with more available degrees of turns, such as creating diagonal straights by loading a four-by-four square and having two opposing corners have walls, while the other two are open, which allows the user to traverse through. As well, with the simplistic design that these two template prefabs would allow for researchers to implement their own prefabs with faster iteration to test more aspects that they might desire.

We lastly thought about the possibility of even simplifying this system to a one prefab utilization technique by having only a single cube with an empty child coordinate above it, but we quickly realized that that is simply a voxel generator, so we swiftly discarded the idea.

29.2.4. Theoretical Performance Trades

After we had developed our three methods of utilization, we began considering which would be most optimal and easy to implement, however, we have a lot of leeway since we are not planning on developing this toolset to be a high-fidelity visual experience, we only wish to have a small amount aesthetics for which their assets would not take up too much storage, even before this project has been built into binaries. We went over a high-level idea of how each implementation would look like in code. The common thing that all these implementations would have are that they would take in the new coordinates in the grid to load the shape that has been requested by the algorithm.

By looking at all three utilization methods from four assets downward, we go down from having more cached memory taken up because of the number of assets that have to be saved and serialized into whatever object that will handle asset loading, and less loading would be done because less calculations have to be done, to having much less cached memory, but more loading and placement calculations to be done since we do not have walls, and shapes are too primitive. This told us that the number of assets that we have is inversely proportional to the number of calculations and operations that have to be done.

29.3. Decided Implementation

After careful examination of each of our conceptual implementations, we decided upon using the Three-prefab Utilization for our generation, but instead of working on a grid, we opted for being in a gridless world, and instead, we would have a snapping system between components through the use of Unity's box collider components, this allows it so that even if the algorithm where to generate a segment at a very small yet noticeable offset from its previous one, it could snap the new segment to make it look flush. The overall reason for why we opted to use this was so that we could still have variable-sized straight segments of even floating-point values, without having to repeatedly have to place smaller straights. This decision was made at the time when it came to developing a prototype for the prefab generation, so this goes into more detail of the Prefab Generation Prototype section, in its subsection, Straight Segments.

After careful examination of each of our conceptual implementations, we decided upon using a Two-prefab Utilization, and instead of working in a grid, we would be working in the xy-plane, where x equals zero, and y also equals 0.

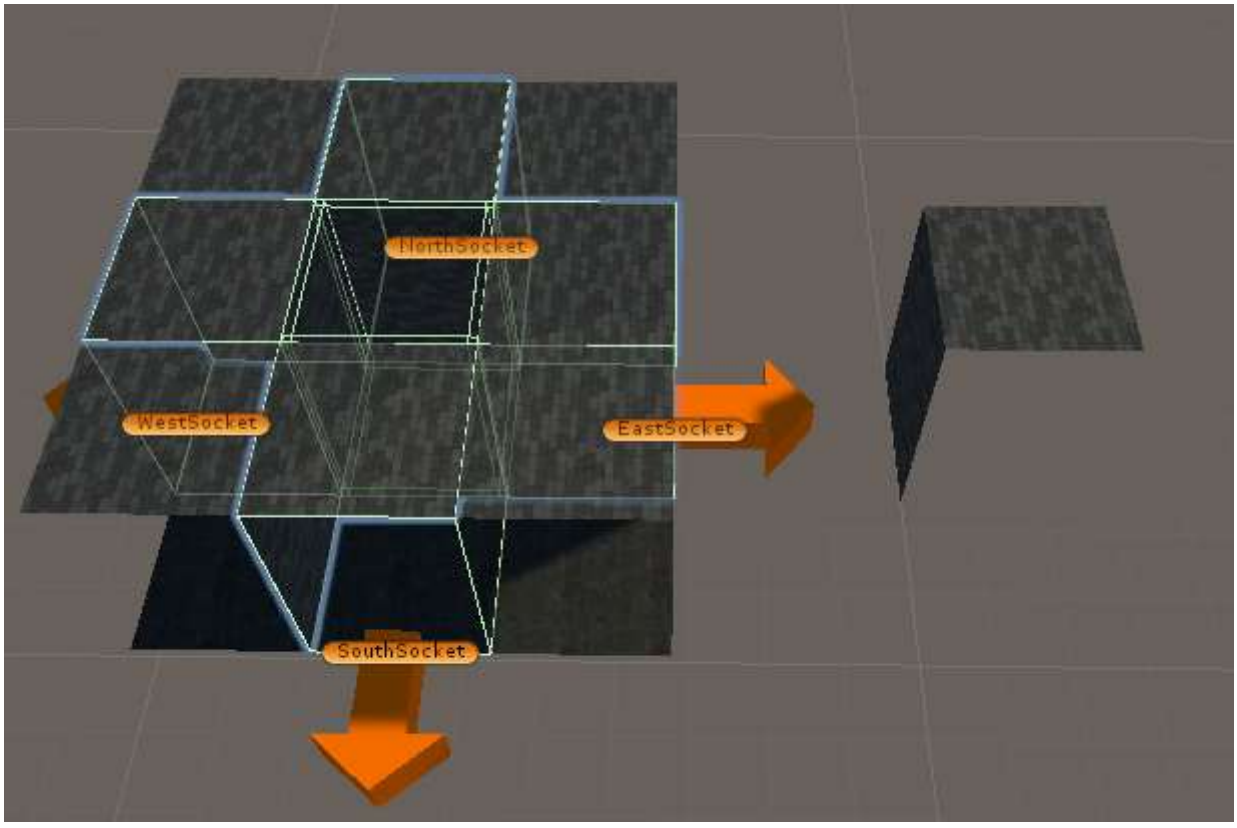


Fig. 59. From left to right: A Four-Way Junction with all its walls closed (highlighted), and a Wall Piece, both prefabs are textured.

We created a system that snaps newly generated pieces into the sockets of old ones; as the orange arrows of the left prefab shows, when a new piece is created, it takes the position of that arrow, and the arrow itself becomes now disabled, which in turn makes it invisible and no longer has any interactions with the simulation environment, either physically, or through scripts.

29.3.1. Wall Placement

Walls are placed similarly to the originally proposed Three or Two-Prefab Utilization, where the root would have placed empty child objects, which in this case is four, where they are placed in each absolute cardinal direction, however instead of loading a wall prefab at the desired location, the empty children would each have one child that *is* the wall prefab, and it's default state is to be disabled, which in turn makes it invisible.

29.3.1.1. Algorithm

We placed all the walls into an array of size four with the following indexing: North is zero, East is one, South is two, and West is three. We wanted to have the following piece types: Four-way intersection, three-way intersections (rotated variants as well—exits going either left or right, left or forward, right or forward), left turns, right turns, and straights; we determined the following patterns for closing walls of a newly loaded piece to create the desired one, based on the cardinal direction of the previous piece's socket:

- If the piece is straight, leave the opposite wall opened, and close the two adjacent walls.
- If the piece is a left turn or a right turn, close the wall in the opposite end, and close the wall in the opposite direction of the desired piece—left turn closes the right wall, and right turn closes the left wall.
- If the piece is a rotated three-way intersection, close the opposite walls from the left or right turn piece logic.
    - Otherwise, simply close the opposite wall from where the previous piece's socket index.

Now, it was a matter of determining the direction of generation for proper rotated placement of the desired piece. We knew that since we have each socket be the same index as the cardinal direction, we simply had to find the index of the next or previous

socket in such a way that it would not go out of bounds. We did this by using the size of the array and the modulo operator to stay within the bounds of the array and determining neighboring sockets as such:

$$left\ neighbor\ =\ (parent\ index\ +\ 1)\ mod\ 4$$

$$right\ neighbor\ =\ (parent\ index\ -\ 1\ +\ 4)\ mod\ 4$$

And for finding the opposite wall:

$$opposite\ wall\ =\ (parent\ index\ +\ 2)\ mod\ 4$$

Where the *parent index* is the array index representation of the socket that is loading the new piece. Afterwards, we optimized the logic path of which walls had to be closed by looking at which ones commonly shared this behavior between the piece types.

29.3.2. Visual Optimizations

After we had our algorithm for wall placement, the initial design of the prefabs was originally composed of rectangular prisms. However, we noticed that in the special case when we had mazes with very long stretches of hallways, there would be a slight drop in frames, since the user would be pushing the generation system to place and create a vast amount of pieces at the same time. One main issue was the amount of vertices, indices, and texture coordinates that had to be loaded from our prefabs; where they could range from 40 vertices per piece, which is the case for a four-way junction with no ceiling (more on that later), up to 64 vertices, which for a piece that is either a left or right turn, and it also has a ceiling. Since we realized that there will be sides to these rectangular pieces that will never be seen by the user, such as the underside of the floor, or the behinds of every wall, we could optimize the prefabs even further by converting them to one-sided quads. We were able to achieve this by removing all of the unnecessary vertices from the prisms, which automatically updates the indices and texture coordinates. By doing this visual change, we were able to obtain a substantial performance increase in loading prefabs in situations that would require a lot of them since the system had less data to load. This is why figure 59 looks as such, although some sides may look invisible, it is simply because of the one-sided property of the quads.

30. Algorithm Researcher-Defined Parameters

One of our biggest requirements is that we want this toolset to allow the researcher in their own user interface to customize and set certain requirements for the generation algorithm. We ended up coming up with a hierarchical model that has certain overrides for the most basic maze generation, in which the lowest priority type of parameter would be Randomness, where when there is nothing else for the algorithm to specifically meet, it will randomly generate segments that connect properly.

30.1. Distance and Time Parameter

The first parameter that the algorithm looks at is the Distance Parameter, it depending on what type of prefab generation method will be used, it allows the researcher to set a specific distance for user to have to traverse before the maze has been completed, the algorithm will then have to keep track of distance traveled while not keeping track whenever the user is backtracking. This next parameter can also be considered that first, which will be discussed after its explanation, it is the Time Parameter, this allows the generation tool to keep track of a countdown timer that the researcher sets before hand, when the timer is over, the algorithm has to create the exit of the maze in the next generation. These two parameters are both the first because they are mutually exclusive, since having one of them prioritize the other would then make one of them nullified, for example: if the researcher wanted the user to travel five hundred meters in three minutes, the user should not be rushed, so if three minutes have passed, the next segment the user will discover will be the exit, but they might have not traveled the five hundred meters, inversely, if they traveled the five hundred meters in less than three minutes, then the timer was useless. While these two parameters are also in an unique state where although they are the next lowest in the hierarchy, once a requirement for either of these two is met, it tells the algorithm to finish the session as quickly as possible.

30.2. Periodic Parameters

There are three parameters in this category. The first one is the Maze Windingness, where we could have a range number that could either be zero to one, or zero to one hundred, where the higher it is, the more turns the user will encounter, this could inversely be names Maze Straightness to determine the likeliness of straights

happening, however, we concluded that Maze Windingness seems more appropriate, since researcher will probably want to test out locomotion with more effort from the user. The second and experimental parameter is the Impossible Space Likeliness, similar to neighboring parameters in this hierarchy, it could have a range from zero to one, or zero to one hundred, where it determines how often the algorithm created paths that lead to impossible spaces. Lastly, the third parameter would be the lengths of straights, rather than having a single value to have a consistent size all the time, we would have a number range that determines what should be the minimum length and the maximum length of a straight, so when a new one is created it, it picks a random number from that range.

30.3. Asymptomatic Parameters

These are the highest parameters in the hierarchy, and influence mostly only the parameters below them, they all work by having a switch that contains the word “at least” or “at most,” and an input integer; when the switch word is “at least,” the algorithm has to create the minimum amount of segments while considering the lower hierarchy parameters so that this one is not ignored, if the researcher had set the user to traverse the maze for two minutes, and its windingness is very low, and they also set so that there needs to be at least ten left turns, no matter how low the probability of turns of intersections is, it has to create at least ten left turns; This applies to all segments that are selected, straights, left turns, right turns, three-way intersections, four-way intersections, and something that is not a segment, and impossible space. When the switch word is “at most,” the algorithm would follow its other parameters while having a counter that keeps track of everything has been generated, and the moment whatever segments has been assigned reached the number for their “at most” parameter, the algorithm cannot generate more of that prefab for the rest of the session, this same restriction applies to impossible spaces.

30.4. Granular Parameters

This last set of parameters is very conceptual and might not end up being implemented. We discussed the possibility of making our maze generation as granular as possible by making an isolated set of parameters that override everything in the generation, and tell the algorithm to load specifically what the researcher created. This would be an expandable list of parameters that would first contain an integer input box,

then the type of segment, and another type of element. This allows the researcher to make certain specific insertions to the algorithm, for example: if the researcher wants that after the third right segment that is generated, they want to force the algorithm to generate a 10 meter long straight segment. We opted from implementing this because it could hypothetically allow one to insert so many of these parameters, that one could build the maze by hand, which went against our design philosophy, and the entire purpose of the project.

30.5. Parameters Graph

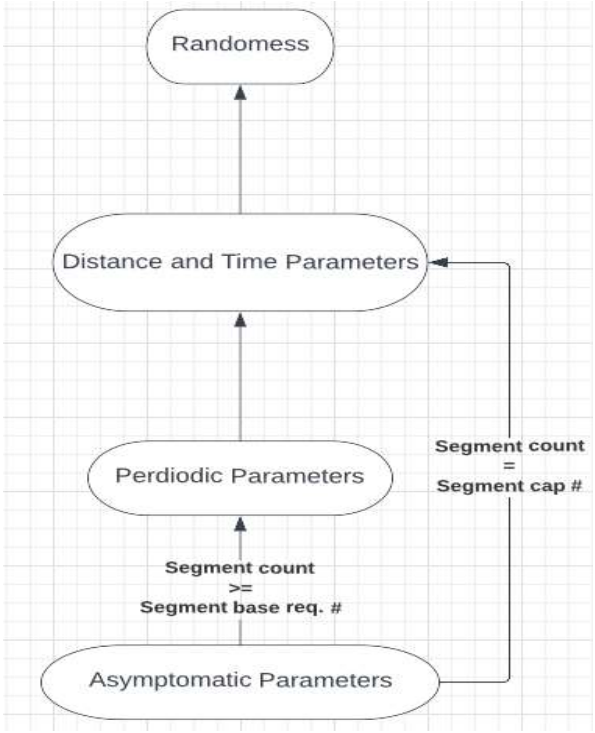


Fig 60. The parameter graph

30.6. Decided Implementation

30.6.1. Visual Parameter

The second set of parameters is what we called *Visual* Parameters, these do not have any effect on the generation itself, and they theoretically could be more important for affecting the user's experience inside the maze itself. We first added the option to

toggle whether all prefabs should have ceiling or not, however, we did notice that this would create very dark environments, as such we decided to add spot lights that point towards the floor for clearer better lighting and visibility.

Next we had wall height adjustment, while we originally wanted full control of the overall wall height, we realized that this could allow the researcher to set a height of zero or negative numbers, this caused a few problems, the first one was zero height would cause for for quad faces to overlap, and thus cause flickering as the user traverse. The second issue can also be applied to zero height again, but also negative height values, this would visually make it so that the user only sees the floor of the generated maze, but based on the visibility system that we implemented, they would be able to see the generation happen in real-time from afar, which defeats one of the systems of the project. We made the design choice of giving walls default height of 3 meters, and the researcher adds extra length to that height, the only caveat to this was that the only way to stretch an object in a respective axis, which in this case would be the z-axis, the object also stretches in the opposite direction. The solution to this was by translating the wall coordinates in the positive z-axis by half the stretch value.

31. Prefab Generation Prototype

As a team we have decided to work on the main components of this project by developing isolated prototypes, this was done because some systems took longer to document and flesh out. However, we are developing these prototypes so that they can easily be coupled with one another, and keep close communication for what inputs and outputs we would have in order to create templates to test with. So while the algorithm is not being developed in tangent with the prefab generation, it is still possible to start work on the latter of these two systems.

31.1. Algorithm and Prefab Intercommunication

The first thing that we discussed when we started development on any of the two main systems for this project, which in this case, was the prefab generation; it was to define a method of communication between them, since prefabs are dependent on what the algorithm creates. We ended up coming up with three ways, from most complex, to the simplest one:

Fig. 60. Method I: Intermediary Dynamic File.

This first method was inspired by also the brainstorming for how to develop a templated system to test prefab generation independently from the algorithm, but that will be talked about in a later section. The purpose of this implementation was that the algorithm would send its generated data to an intermediary file that would be changing dynamically, both in modifying entries, and expanding its contents in real-time. This makes it easy to keep both of the main systems isolated, and thus quarantining whichever one ends up having erroneous behavior, or checking to see which one specifically crashes. This implementation though, was tossed out as fast as it was thought of; its main issue was what it mainly was about, the intermediary file, since editing data entries, read, and writing all at the same time would seem to be very troublesome, since we would have to be very careful telling the file what to do at a specific position, and so, it would most likely end up creating errors that in the least worrisome scenario, the user could end up soft-locked in a fully closed of environment, and in the worst case-scenario, the system would crash with hard to analyze reasons as to why it did, since we only thought of using this file in run-time, and not save it afterwards.

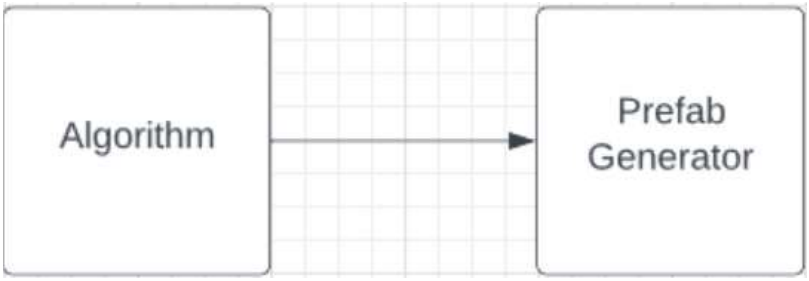


Fig. 61. Method II: Direct Communication

The second method that our team developed was to ameliorate the drawbacks that the first one's Intermediary Dynamic file had, which was to remove the specificity and finickiness that was required to write and update data of the generated maze, and have faster time and space complexity, since no extra functions or libraries would be required to handle input and output streams. We still also keep the two main systems

isolated from one another, in case one of them is faulty. As for the actual way this implementation would work, we would still create two different classes for each system, but the algorithm class could call an event that the prefab generator is listening to. That said event would also send the generated raw data for the second system to interpret and load into the virtual environment. An alternative but still similar implementation would that the invisible object that holds the algorithm system would have a direct serialized reference to the invisible object that holds the prefab generator, so once the first system is done generating data, that data could be sent to the referenced object.

31.1.1. Decided Implementation

After conceptualizing and carefully looking at both of the first two implementations, and looking at their use-case scenarios, we end opting for third method that end up being a combined solution:

Fig. 62. Method III: Unification of both systems

Rather than having both systems be isolated, we would put both of them as a single unified system that handles both the data generation, and its interpretation and loading in the virtual world. The first thought that the reader might have when looking at this implementation method could possibly be red flags all around, since the idea of having both systems leads to very tight coupling, which could be detrimental when working it, since that would mean that testing one system depends on the other system working as well. But the benefits of this implementation outweighs the detriments of it. Since the purpose of the algorithm is just to create and send data *only* to the prefab generator, we thought that there would be no purpose in having both of these aspects be in their own respective classes, and while it could be beneficial to have independent classes, based on the time and scope of this project, we have not planned the possibility of leaving open ends for other conceptual systems that could end up relying on any of the two core systems for communication querying, also, we could solve the

issue of testing systems independently through the use of preprocessor directives, for example: we could create one called USE_ALGORITHM_SYSTEM, and instead of using premade data for the prefab generator, it will start relying on the algorithm instead; the opposite could be done by having one called USE_PREFAB_GENERATOR while testing the algorithm to check if the system is generating the data at the right position and orientation. Our last reason for why we ended up using this implementation was because it provides the fastest method of communication and execution, from the data generation, to its visual representation in the virtual world.

31.2. Conceptual Premade Maze

Before looking into writing a premade maze, we had to draw a conceptual one to give us a clearer idea of what are the most common cases, and special cases. We came up with the following criteria:

1. There needs to be a start location.
2. There needs to be either a left turn segment or a right turn segment.
    2.1. Following up, the next turn segment that should be loaded, even if there is a straight segment in between, should be another turn of the previously loaded one, this is to show that the prefab generator understands the heading direction of the data generation.
3. A three-way intersection.
4. A four-way intersection
5. At least two differently-sized straight segments.
6. Branching paths.

31.2.1. Straight Segments

For something that sounds so simple, in the case of placing straight segments, this actually needs to have its own subsection here, due to its complexity. Quickly mentioning again, originally, when dealing with a straight segment, for a Three-Prefab Utilization method, when the algorithm sends a straight of a specific length, that length would have to be of a discrete size in relation to a grid, and the prefab generator would then take that size, and simply place repeated straight segments until the overall total length that was originally sent was reached; we thought that this way of loading to what would represent a single straight creates too much redundant data that is not needed.

Towards one of our last meetings, we thought that it would be better to load and place a single straight segment that would be placed, and it would be stretched along the axis where the user would walk on. A small issue with this is that since scaling of any object takes place at the center of the object, scaling at that location for the walking axis of a straight segment would mean that it would stretch equally on both sides. Ideally, we would prefer if the stretching for a straight would only be from one side, since it would allow its loading and spawning to be placed at the edge of the previous segment, and then it would reach to the desired length. One solution that we thought of, in case we could not figure out how to stretch a segment only from one side, would be to load it half of its desired length away from the edge of the previous segment, and then it could be stretched out evenly from the center to fulfill the length that was sent from the algorithm.

With all the information provided, we were able to create a simple concept maze that covered all of the points that were mentioned at the beginning of the Conceptual Premade Maze section:

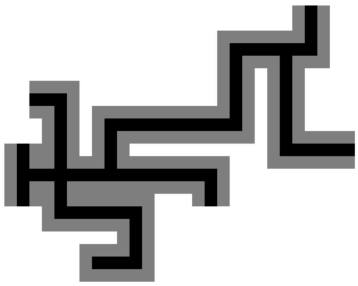


Fig. 63. Concept premade maze

The lower most segment of this maze is the starting point, right after, it shows points two and two point one by having two left turns in different heading directions, afterwards there is a right turn that leads to a four-way intersection, which is point number four, the right turn of that segment then leads to a three-way intersection, which is point number three; two or more differently sized segments, which is point number 5 is easily seen. After having sketched out this conceptual maze, we started to discuss how to write data for it.

31.2.2. Making the Data

When coming up with how to develop the actual data that would be used to recreate the conceptual maze that was drawn in the previous section in order to test the prefab generator, we ended up looking at two approaches for the methods in writing said data.

31.2.2.1. File-Based Data Implementation

This method of data implementation had two sides; it could start as something as simple as a custom made file that could have rows in which column corresponds to the x and y position of a segment in the virtual world, perhaps an integer indicator that tells the program what type of segment it is, but this could also be changed to be a string instead with the name of the segment instead, and possibly as well, and the heading direction of that segment, which we could tie to the world plane that is being used for spawning, and such, the heading directions could be negative x, positive x, negative y, and positive y; we think it is better to not use cardinal directions such as North, South, East, and West, because they are more dependant on other physical factors that are not related to the space of the virtual world. The more complicated file implementation would be through the use of a Json file, where again, each entry in the file would have similar attributes to what was mentioned for the previous file system, but it would look more organized since these attributes would have name tags also attached to them, which makes each entry probably easy to parse, however, the issue of using this file system is that it would take more time to write the functions required to parse the data properly, we would have to learn how to use Json files in general, and it also could be troublesome for trying to write branching paths, which will be discussed later in this parent section.

31.2.2.2. In-Engine Data Implementation

This second approach seems more practical, and it solved a recurring theme for whenever the team would come to a halt due to an implementation problem, rather than looking at the problem in a more traditional Computer Science approach, where we would have to make all the data structures and algorithms from scratch, we could use our the software that we are using to our advantage to facilitate the problem. Unity Engine has a built-in system for displaying class objects through a user interface which allows to view what data that object holds before running a project, what data the object

is holding in runtime, and all of this could be edited through that same user interface; the types of data that it shows are the following:

- Integers.
- Booleans.
- Low-precision floating-point numbers.
- High-precision floating-point numbers.
- Arrays.
- Characters.
- Strings.
- Enumerations.
- Vector types
    - 2D, 3D, and 4D.
- Other object references.

With all this information, we could develop an object class called “MazeSegment” for data that holds a segment’s world position as a two-dimensional vector, the type of segment that it is through an enumeration that indicates if it is a straight, a left turn, a right turn, a three-way intersection, or a four-way intersection, and its orientation would also be an enumeration that could have the values of the positive x-axis, negative x-axis, positive y-axis, or negative y-axis. Then, we would create another object class called “PremadeMaze” that can hold an array of MazeSegment objects, the array could then be easily written by hand through the aforementioned Unity user interface system. We think that writing data this way is the fastest way to write it, and not only is it easy for Unity or C# to understand, it is also easy for the developers to write human-readable and easy to follow data. So far, we have decided to use this method for implementing the premade data.

31.2.2.3. Data Accommodation for Branching Paths

When looking at how data for either a straight, a left turn, or a right turn, the natural path that the user takes would be from where they enter the segment, to where they exit it, because of this, data for these three types of segments could be placed in the object array in a linear fashion, such as having the data for a right turn, then a straight of an arbitrary length, and lastly a left turn. Now, problems start to arise when discussing three-way intersections, or four-way intersections, since although the

previous data points from the object array can be lead in a linear manner, the moment we reach any kind of intersection we lose all sense of order due to the non-linearity that these two segments present, as they both lead to different data sets, which are represented as paths. To solve this, a tree-like data structure could be used for the data could be developed as such:

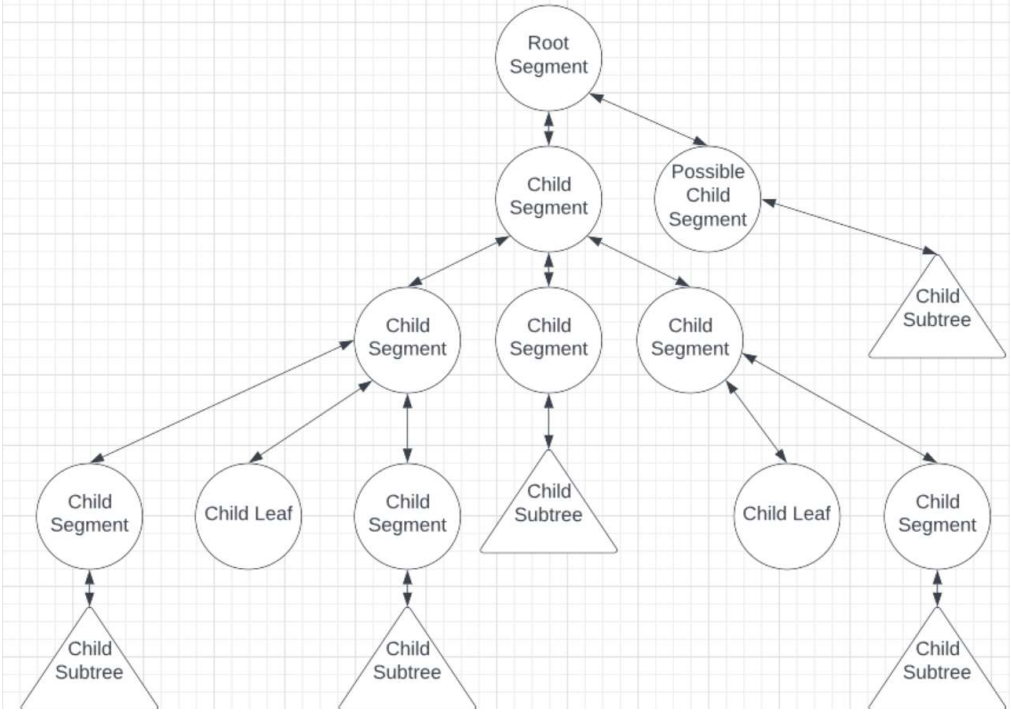


Fig. 64. Premade Maze Data Structure

Hopefully this graph displays enough context. Again, this structure has the properties of a tree, where there are parent-child relationships, but what makes it standout is that it also has the property of traversing back up to a node's parent like a doubly-linked list; nodes can have from zero to three children, and one parent, with the exception of the root node, since it would have not parent. We also quickly realized that in terms of visuals, the tree is a very close representation of what the maze will look like in the virtual world, with straights and turns being nodes only with one child and a parent, and three-way and four-way intersections would be the nodes with more children.

By adding this data structure to the original PremadeMaze class, this solves the issue of having branching paths, by instead removing the object reference array to be the tree that can be expanded to view the nested children that either make up linear segments or split into branching paths. This also gives insight as to what type of individual data should the algorithm system send to the prefab fabricator. One would initially think that the algorithm would send the entire tree to the prefab system, but

instead, the former system would create a segment data point at a time, and the latter would receive it, the prefab system would then create the tree little by little, as the user keeps traversing the maze, which makes the algorithm generate more data. This again might lead the reader puzzled, as they might then question why a tree needs to be used for the premade maze, if the prefab generator will be basically generating it as it is reading that said premade tree. This is actually useful for us in testing since we can compare the original tree with the generated one to see the system is following the flow of sent data correctly, and secondly, the tree is premade because this part is for specifically testing the prefab generator isolated from the algorithm itself, and thus we need consistent data.

31.2.2.3.1. Tree Structure Special Cases

Although the sketch drawing that shows how the tree data structure would generally look like, the root node has another child node labeled *Possible Child Segment*, this is because there are special cases that would make more uncommon type of root node for the tree, and considering how many of these cases there are, they are more likely to be what the algorithm will generate more often, and they are as follows:

- If the user appears in the middle of a straight segment with no dead ends on it, a left turn segment, or a right turn segment, the root node of the tree needs to have *two* children.
- If the user appears in the middle of a three-way intersection, the root node of the tree needs to have *three* children.
- If the user appears in the middle of a four-way intersection, the root node of the tree needs to have *four* children.

31.2.2.4. Passing the Conceptual Data from the Algorithm to the Prefab System

Now that we have all the necessary information for how is it that data should be interpreted and used, we have to take that information for each individual segment from the algorithm and pass it to the prefab generator; this could be done in two ways: either the algorithm creates tree nodes inside its functions and passes them to the prefab generator directly, so the latter simply assembles from each node that it receives, or the algorithm creates and sends a MazeSegment object, and the prefab generator takes that object and makes it into a tree node which then gets added to the maze segment tree.

There is a more complicated issue, which relates back to the already complicated branching paths, that because of them, we had to create this tree data structure. Going back to a three-way or four-way intersection's nonlinear nature, we then thought about how both the algorithm and prefab generator behave after branching out, since at this point, when data is being created, it is simply added as a child of the previous segment that was generated, but when we have branching paths, at the time when we first thought of the tree data structure, we did not have defined behavior for handling the creation of multiple paths. We thought of two solutions that are very similar; first of all, these two solutions begin the same, when the algorithm decides to create a three-way intersection or a four-way intersection, create a tree node with the appropriate number of children for it. The part where both approaches differ is when creating the branching paths; the first approach is simply creating each child branching path iteratively with a depth-first approach, once a child path is completed, go on to the next, once all child path are completed, send the completed node to the prefab generator; the second approach simply creates all of the branching paths in parallel, and while Unity Engine does not allow multi-threaded behavior, we can still use its equivalent called Coroutines, which allows for the same behavior through the use of virtual tasks, again, once all the child paths are completed by the algorithm, the parent node for them is then passed to the prefab generator. However, like most things in this project, problems end up creating smaller problems the more we expand on an implementation, in this case, it was figuring out when to stop generating child paths from branches.

One of the most imperative features about this project is that the user should never know that the maze that they are traversing is being created as they go, because of this, they can never see an area that is being generated, this is due to the fact that it would break the immersion for the user, and they could quickly realize that no matter what path they take, it is not a traditional maze; they might also realize that they are being lead to a particular direction, rather than making their own choice, since they would not be aware of the random generation that is going on in the background. This type of issue can be mostly seen when the user is in a three-way intersection or four-way intersection, where a worst possible case scenario would look like the following:

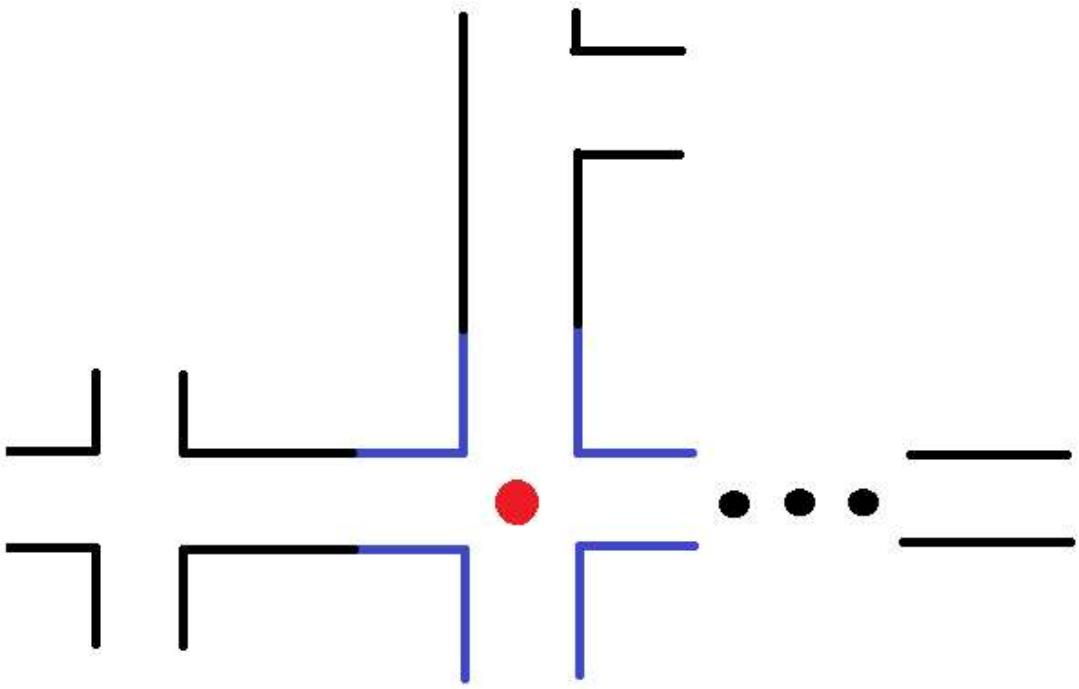


Fig. 65. Worst Possible Case Scenario for Generation Occlusion

In this case, the user is the red dot, and they are standing in the middle of a four-way intersection segment, which is colored blue, if we consider that the human eye can see from very far distances, and the far clipping plane that is used for discarding objects from rendering when they are too far from a set distance is set to be also at a large distance, we have four cases where the user could see past what is being generated. The left case shows that another four-way intersection would still allow the user to see past what was generated. The top case shows that a rotated three-way intersection that has either a left branch or a right branch, and a forward branch also allows the user to see past what was generated. The right case conveying through the use of ellipsis shows that if there is a long straight path, the user can still see past what was generated. Lastly, the bottom case simply shows that there was nothing generated for the user to see, perhaps because the algorithm was not designed for generation of multiple children. In all of these cases, this is because if the user were to be looking towards the direction of the generated branch, there is no wall at the end of the forward direction that they are looking at. With fine tuning of the virtual camera that the user sees through in the virtual world, there has to be a distance golden distance where users cannot discern if there is something generated or not, and the camera is not rendering past that distance, but this small solution only works for extreme distances, and is not the most common case scenario for these kinds of situations.To solve this issue in a more generalized manner, we refined this behavior of the algorithm by still

utilizing doing a depth-first approach at generating each branching path, either iteratively or asynchronously, but the generation of each path would stop the moment the algorithm either creates a left turn segment, a right turn segment, or a three-way intersection that leads the player to pick either a left path, or a right path, of course, the rotation would be relative to the heading direction of the algorithm.

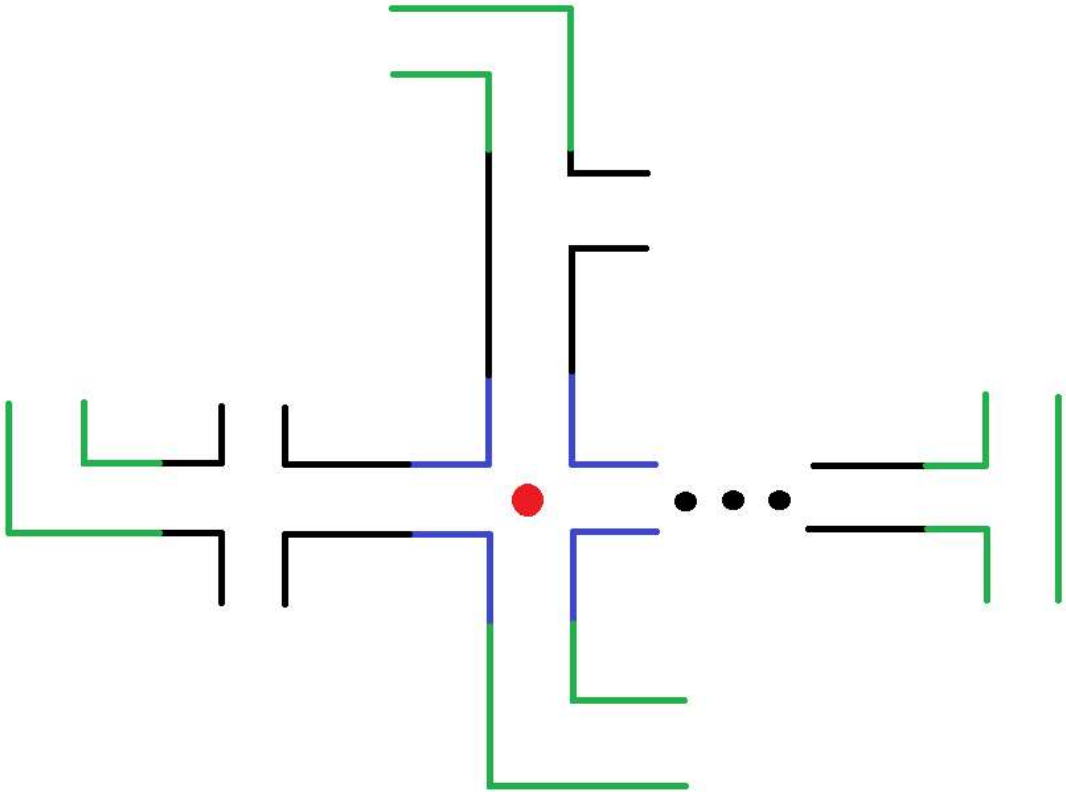


Fig. 66. Ameliorated version of branched path generation.

As the above drawing shows, while this still the same original paths that were created for a four-way intersection, now, the user's view will always be obstructed by the wall of a generated segment from the point at which they are standing, no matter how far the generated segment is. After briefly discussing the previously mentioned solution, although it correlates with what the user visually sees, this is concerned more with the actual algorithm, rather than the prefab generator, because of this, it is of no interest to expand more on this in *this* particular section.

Now that all is clarified with how the tree-like data structure is generated, posited two ways as to where would this structure exist in the virtual environment, since as mentioned before, every script that we develop for this project in terms of the actual user testing and data gathering has to exist in some kind of way in the virtual world for it to be able to execute. The tree data structure could either be an attached component script to the prefab generator object, or also an attached component script, but instead, it would be to the user object.

31.2.2.5. Tree Node Parent Attachment

The last aspect of the tree-like data structure that has not been talked about for a while is a node's parent, since we still have not talked about any defined behavior for designating them. The solution that we came up with consists of creating another node pseudo-pointer that holds a reference to node, which would be the node for where the player is currently inside of. This node pointer would exist as part of the user object inside of the virtual environment. There are two ways this pointer would be updating, but they both have very similar behaviors; the most common one would be that the moment the user enters a new segment through a function made in the user object itself, ideally, in less that a frame, first, it would grab the reference of the newly entered segment, then it would set the parent of the new segment reference to be the current segment pointer that is built into the user object, and lastly, it updates the current segment pointer to be the newly entered segment. The second use of this approach would be when the user spawns for the first time into the virtual environment; since the current node pointer right at the beginning before the user spawns by default is a null value, when the first node of the maze is loaded, which is also the root node of the maze tree, the first entered node's parent will be null.

The purpose of node parents for nodes is for when the user is doing backtracking. In a similar manner to when entering new segments, again, in the same function for checking when one has entered a new segment, we could simply check if the entered node is the current node's parent, before checking if it is instead a child node of the previous node the user was standing on.

In general, in order to check for any kind of collisions in Unity engine, there is a built-in function that is provided for virtual object that has a collider component on it, this function is called OnCollisionEnter, it is a void function, and its only parameter is a Collision Object, which is created by the built-in physics system, this object type contains all the information that is needed to perform on collided objects, and more. So, for the pseudocode of how the function would execute, it would perhaps look as the following:

```
// Global MazeSegment to this class for storing current
// segment the user is in.
public MazeSegment currentSegment;

void OnCollisionEnter(Collision other)
{
    // Check that the collided object that the user entered
    // is a segment
    if(other.gameObject.tag != "segment")
```

```
        return;

    // Get a reference to the entered maze node
    MazeSegment enteredSegment = other.gameObject.getComponent<MazeSegment>();

    // Check if user is backtracking by seeing if the entered
    // segment is the current segment's parent
    if(currentSegment.parent == enteredSegment)
    {
        currentSegment = enteredSegment;
        return;
    }

    // Otherwise, check if the new segment entered is a child of
    // the current segment.
    foreach (MazeSegment segment in currentSegment.children)
    {
        if(segment == enteredSegment)
        {
            enteredSegment.parent = currentSegment;
            currentSegment = enteredSegment;
            return;
        }
    }
}
```

Of course, these names and variables are all simply placeholders.

31.2.2.6. Handling Impossible Spaces

When describing what impossible spaces can exist, there is one kind that is more common than others: Backtracking impossible spaces are the type of impossible spaces, these would occur if the algorithm were to generate a dead end on purpose, and so when the user turns around, either when they turn around, or after they traverse backwards for a few segments, there will be different segments from what they had previously seen.

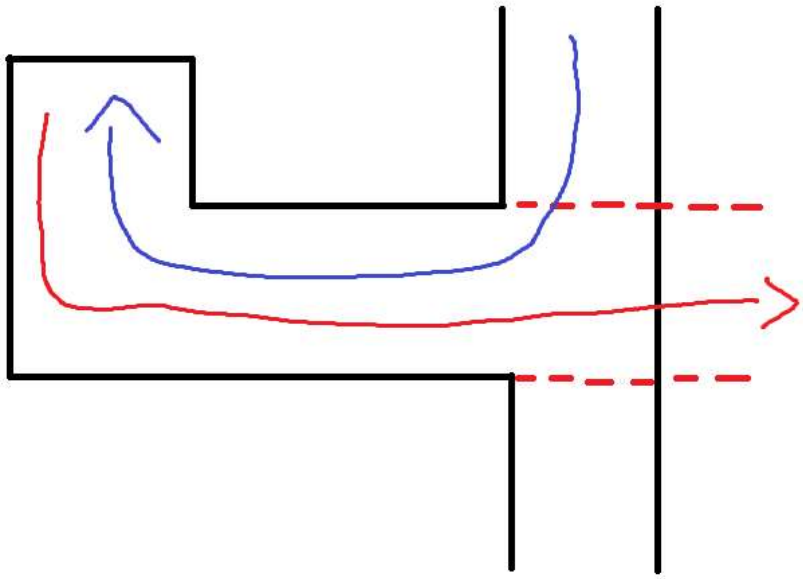


Fig. 67. Direction of user path for backtracking impossible spaces.

As the image shows, if the user were to enter a three-way intersection, where they take a path that leads them to a dead end, the original path from where the user enters is the blue arrow, once they reach the created dead end, the algorithm and prefab generator would overwrite the maze so that the original three-way intersection is now a straight, and the new path that the user takes is indicated by the red arrow.

In order to perform this kind of operation, we simply have to snip off the parent of the current node pointer and set the parent to be null, which now makes the current node the root of the tree-like data structure, and expand the list of its children to accommodate for adding the new entered segment as an extra child.

32. Testing and Evaluation

32.1. Introduction

In any project, it is important to ensure the goal of the project is met, even after initial development. We want our project to be useful to the VR research community and hope to make further improvements to it when necessary. As we are not actual users of the project and are biased as developers of the project, it is necessary that we utilize input from users when looking to improve the project.

To achieve this, we will look at two methods of gaining insight into what can be improved: surveying and testing. With these methods in mind, we can hopefully gather

the information necessary to build and maintain the most useful tool for researchers and users.

32.2. Testing

To test our project, we will use a variety of different methods. Each of these will hopefully give us insight into various aspects of the project, whether it be improvements to the UI/UX, maze generation algorithm's performance, or simply bugs that we didn't catch.

32.2.1. Algorithm Testing

The process of testing for the algorithms portion of the program will involve generating multiple mazes to test out if the expected output occurs. The testing of this part will be broken down into two parts: probabilistic testing and preset testing.

The first testing phase will consist of preset testing where the maze generation algorithm will be created and rendered in a pre-defined manner. The purpose of this is to not only test if generated pieces are correctly following a predefined input pattern but also if junctions and segmentations are connected properly. Segments come in many different shapes and sizes, and as such there contains things like loops/swirls and zigzagging and testing out different paths that include multiple of these allows for the detection of any unintended clipping when an impossible space occurs or even just the next segment/junction not spawning in the correct location.

The next testing phase will be testing if our probabilistic generation is occurring as expected. Once all the bugs have been squished and generation works as intended then the team will be able to focus on allowing for probabilistic generation of next path pieces. To test this out the team plans on generating very long mazes multiple times and counting each type of segment and junction and checking if the ratios align with their respective probabilities. Generating long and large amounts of mazes will allow us to take advantage of the "big number" theory where if you repeat an experiment enough times the random probability of an action should match its expected probability. (Example: flipping a coin four times can lead to inaccurate measurements in regards to the probability of heads and tails but flipping a coin 500 times leads the probability of each side closer to 50/50.)

32.2.2. Experimental Runs

One method of testing involves running tests with willing participants who are unfamiliar with our tool. Often, being a developer of a tool gives you a different perspective of a tool as opposed to someone who has never seen your tool before. This shift in perspective can make it difficult to spot bugs and find pain-points that affect the user experience, among other things. To circumvent this, we can introduce unfamiliar users and get as much information from them as possible.

For example, we could bring in willing participants to test the maze traversal experience. We would introduce them to what the experience will be like, but be careful not to influence them in a significant way as any influence may impact their view on the experience.

After they finish traversing the maze, we could collect information by asking them about various parts of the experience:

- Were the prompts indicating what you had to do as a player clear?
- Were there any issues involving the maze that you noticed throughout your experience? Especially things like clipping, pop-in, or other visual bugs?
- What is your general feedback on the quality of your experience after testing?

Likewise, we could conduct a similar trial run with users administering the player as well. To collect information on this aspect of the tool we would introduce the user to the purpose of the tool and ask them to read any relevant documentation on the usage of the tool before prompting them to start administering a maze traversal.

Afterwards, we could ask them about the parts of the experience we are most interested in:

- Was the tool easy to understand and intuitive to use?
- Was the view of the player during the maze run informative and clear?
- Was the admin view of the real-time statistics showing everything you would be interested in?
- Was the player data easy to collect post-maze?
- What other suggestions or feedback do you have on the quality of the administrating experience?

Finally, before utilizing unfamiliar participants, we ourselves could run these experiments throughout our development process. This would be useful to test features as we incrementally add them throughout our development. While testing new features, we would also be additionally testing the older features for bugs and improvements. This would help keep the tool polished as we develop, and reduce the need for significant amounts of bug fixing and polishing towards the end of the development cycle.

The method in which the developers would test the tool is important to specify, as it gives good insight into how we plan on moving forward. Each time a major milestone is met during development, we would do an end-to-end test of the entire tool. End-to-end tests are very effective at ensuring the entire system is working as intended from start to finish.

During each test, we would pinpoint any detail we want changed and analyze the overall experience for any improvements. However, our end-to-end test wouldn't capture the entire start-to-finish process of the tool as we develop since the entire system wouldn't be in place. Even with this caveat, it would be an effective method at polishing and improving the tool as we work through it.

32.2.3. Using Our Telemetry

Our tool is inherently a data collecting tool used to evaluate locomotion techniques in a variance free environment. Because the tool collects a significant amount of data to achieve our goals, that data may be useful in testing the effectiveness of the tool itself. Throughout development, we can use our telemetry systems to analyze the effectiveness of the tool and compare different systems within the tool.

For example, if we observe that with a given implementation of the maze generation algorithm the data shows a large variation in time to maze completion, we might reconsider if that algorithm is effective.

The time to complete a maze in our tool could be considered as a random variable that exhibits a normal distribution. The goal of the algorithm could be to adjust the mean of this normal distribution to a value close to the time-to-completion that is requested of the algorithm, while also minimizing its variance. Thus, for a given time-to-completion parameter, we can measure a quantitative value that represents how good the algorithm is. This quantitative value can be obtained by a number of statistical methods and tests that compare distributions. Using this, we can make an informed decision on the effectiveness of an algorithm implementation and use that to further improve our tool.

Time-to-completion is just one of the various random variables that the tool collects information on. With a combination of time-to-completion and many of the other variables, we can create an informed picture of how effective our tool is at creating a minimally variable maze. This would be immensely useful in our decision making processes during development, but also in proving to others that our tool is effective.

32.2.4. Surveying

Another insightful source of information would be actual user experiences. Given the tool is complete and an outside party uses it, we could prompt them for input on their experience. We could ask questions similar to what we asked during the development experiments, and collect the information digitally.

The method of information collection could be by online survey. We could utilize technology like Google Forms or a Qualtrics survey to gather information from the users. We could add a link to a form like this where we publish the tool, or provide a link to it in the tool itself for a more convenient way to send feedback.

This information would be useful in creating future updates for the tool. We could incorporate feedback on bugs, quality of life improvements, and additional features to make a more versatile and useful tool for researchers.

32.3. Tutorial Section

At the beginning of the experience, the user could be unfamiliar with the locomotion technique that the researcher is testing. They could also be new to virtual reality and will need a minute or two to get the proper bearings before being able to traverse the maze. Essentially, a space for the player to become familiar with the experience may create a more welcoming experience.

Additionally, creating this space for players to become familiar with the experience would be beneficial for data collection. A user that is unfamiliar with the locomotion technique may produce data that is unfit for analysis because factors outside the locomotion technique itself was influencing this instance. Thus, ensuring a player is comfortable and familiar with the locomotion technique is essential for collecting useful, informative data.

So, we introduce a *tutorial section*, where the player is placed at the beginning of any trial. This area has a few technical considerations, which we will discuss further. Firstly, the location in world space where this section is located is important to discuss. There are also a few implications with our telemetry system that need to be worked through. Finally, the behavior of this tutorial section before, during, and after the player goes through it is important as well.

32.3.1. World Location

The world location is the position in which the tutorial section is placed. We originally planned on beginning the maze itself at (0, 0) on the XY plane, but now with a tutorial section we must reconsider. Our first approach was to begin generating the maze at (0, 0) as originally planned, but rather than using a dead-end junction, we could use a straight segment, with the tutorial on one end and the opening to the maze on the other.

This approach seemed to work, but after further consideration we found that the tutorial section itself should be placed at (0, 0) instead. This is to support future users in creating their own tutorial sections, unique to the locomotion technique they are testing. If the tutorial section is at the origin of the world, it is much more convenient to place objects and define interactions within that space. Thus, we decided the tutorial section should be centered at the origin and the maze should be generated beginning outside that section.

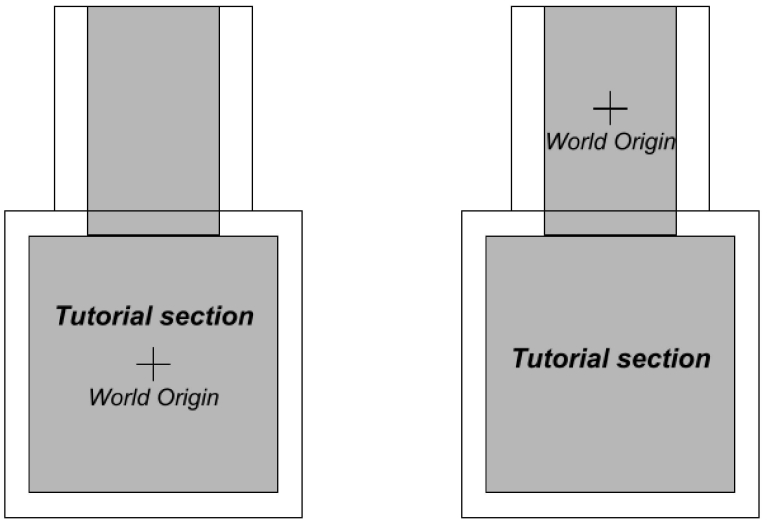


Fig 68. World origin at Tutorial Section (left) versus world origin at first maze piece (right)

32.3.2. Telemetry

As the tutorial section is meant to be a space for the player to become familiar with the locomotion technique the user is studying, it is important that anything they do within the section does not have an impact on the overall data collection of their maze traversal. For example, we would not want to start the timer in a time-to-completion trial until the player leaves the traversal area. This is because the player may spend significant amounts of time in the tutorial section familiarizing themselves with the locomotion technique rather than traversing the maze.

Thus we propose an additional phase of telemetry. This phase collects information on the player's experience in the tutorial section, while also blocking any other forms of data to be collected. For example, some data points could be how long do they spend figuring out the locomotion technique, or how far do they travel with it before feeling comfortable enough to traverse the maze. This data could be useful for understanding how some locomotion techniques could be favorable over others in terms of simplicity of use. Once the player leaves the tutorial section, this phase of telemetry is concluded, and we transition into collecting data on the maze traversal.

32.3.3. Tutorial Section Behavior

We need to carefully define the behavior of the tutorial section as without doing so we could negatively impact maze generation. A tutorial section that is left behind may introduce overlap with maze pieces if it is not accounted for in the generation algorithm. So, it is worth defining the behavior of this area to clarify for future developments.

As the player begins the experience within the tutorial section, it must be spawned in at the beginning. The player will be prompted with instructions on how to use locomotion techniques through a UI that is user-definable, in an effort to support additional locomotion techniques in the future. The player will then have unlimited time to test the locomotion technique and become familiar with using it to traverse the room. Once the user is satisfied, they will be able to exit the tutorial space through a door that leads to the initial maze piece.

Once the player leaves, the tutorial section is closed from the point of view of the player. Once the door responsible for closing the tutorial section is closed completely, the tutorial section is simply removed from the world. This will allow the maze algorithm to spawn pieces in the same location as where the tutorial section was once located, further contributing to a sense of impossible spaces. It also avoids any collision issues that might occur if the space was to remain loaded.

33. Project Milestones

Date To Complete	Milestone
Week 1	Refine socketing logic, integrate into maze generation
	Layout menus and parameters for the Create Maze Screen
	Develop white box versions of the prefabs.
	Path framework set up (data structure to hold path information)
	Define all possible parameters that can impact maze generation algorithmically
Week 2	Implement compound percentage for maze piece generation, record actual percentage in telemetry
	Connect maze configuration data to intermediate data structure for maze to utilize for generation at runtime.
	Attach hints to player controller's so they are aware what their inputs are.
	Develop a prefab attachment system for the virtual environment.
	Piece generation working with socket system

Week 3	Telemetry polling tests, integrate into existing scripts, export to csv
	Test prefab straight generation.
	Test prefab tree system.
	First draft of impossible space detection: update model prefabs to accommodate linked list structure and prefab level data collection
	Player movement influences piece generation to the proper direction
Week 4	Telemetry decouple for per-session data, record VR data
	Connect impossible space detection to the algorithm and send information to the telemetry module.
	Insert stylized graphical assets to the prefabs.
	End piece placement implementation
Week 5	Decouple maze parameters into external class, serialize class for hard disk storage
	Create a tutorial area prefab with corresponding prefabs to make easy signs to tell the user instructions.
	Untraversed paths are properly invisible and impossible spaces working
Week 6	Assist making grid pieces navigable
	Make the load maze screen using the

	newly serialized class saved to disk.
	Algorithm interfacing with prefab system
Week 7	User notification system, create pipeline with telemetry system
	Second draft of impossible space detection: improve upon the algorithm to allow for back-tracking
	Code cleanup and optimizations
	Taking research of Unity standards with maze generation and integrating it into the existing system
Week 8	Contribute to ARXIV documentation, integrate doxygen into project
	UI screen creation: participant waiting screen, pause menus, researcher tutorial screen, confirmation screen.
Week 9	Iterate telemetry output with sponsor feedback, generate mock study date
	In-Depth testing of UI components and impossible space detection
Week 10	Finalize documentation, code cleanup + unit testing
	Last minute changes brought about by testing.

Works Cited

Yin, P. Y. (2020, February 13). *Research on design and optimization of Game UI framework based on Unity3D*. IEEE Xplore. Retrieved November 20, 2022, from https://ieeexplore.ieee.org/document/8990972

T. M. Porcino, E. Clua, D. Trevisan, C. N. Vasconcelos and L. Valente, "Minimizing cyber sickness in head mounted display systems: Design guidelines and applications," 2017 IEEE 5th International Conference on Serious Games and Applications for Health (SeGAH), 2017, pp. 1-6, doi: 10.1109/SeGAH.2017.7939283.

Yao, R., Heath, T., Davies, A., Forsyth, T., Mitchell, N., & Hobberman, P. (2014, March 17). *Oculus VR Best Practices Guide*. Resources. Retrieved November 20, 2022, from https://developer.oculus.com/resources/

Buck, J. (2011, March 4). Maze Generation: Weave mazes. The Buckblog. Retrieved November 6, 2022, from https://weblog.jamisbuck.org/2011/3/4/maze-generation-weave-mazes.html

Elliott, M. H. “Some Determining Factors in Maze-Performance.” *The American Journal of Psychology*, vol. 42, no. 2, Apr. 1930, pp. 315–317., https://doi.org/10.2307/1415287.

LucidChart. Lucidchart. (n.d.). Retrieved December 5, 2022, from https://www.lucidchart.com