Generating and modelling realistic worlds through tectonic simulation
Final Project Report

TU858
BSc in Computer Science International

Dmytro Kosynskyy

C21376161

Supervisors

Cathy Ennis,

Paul Laird

School of Computer Science

Technological University, Dublin

11/04/2025

Abstract

There are various methods of terrain generation for video games, including but not limited to Perlin noise, wave function collapse, or marching cubes. While each of these carry their own benefits and drawbacks, varying in speed, memory efficiency, and individual use cases, these methods don't usually follow realistic physics when generating the video game terrain. The goal of this project is to create a program that is able to create terrain that is based on a tectonic plate simulation that can be applied to both a fixed area map and a procedurally generated map.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

*Dmytro Kosynskyy*____________________

Dmytro Kosynskyy

11/04/2025

Acknowledgements

Thank you to my supervisors Cathy Ennis and Paul Laird for all their help and support.

Table of Contents

1. Introduction

The goal of this project, to create a system to generate and model realistic worlds through tectonic techniques. This project will delve into both fixed size map generation and procedural map generation through the use of these techniques. This report will explore aspects such as the various kinds of plate boundaries, the deposition caused by volcanic activity. The final implementation of this research will be a system that can be used in game engines such as Unity, or Godot to help create realistic terrain that does not rely on the limitations of Perlin noise, while maintaining its benefits of speed and versatility. The project Idea originated from watching the YouTube channel Artifexian by Edgar Grunewald [1] who goes into depth about all the minute details that make up a world in their worldbuilder's log series. This project hopes to bring to reality some of the aspects Artifexian explores.

This project does not aim to eliminate or replace the use of Perlin noise, as much as it offers an alternative solution for world generation that can be used in tandem with Perlin noise to help build more realistic worlds. This report will be comparing and contrasting the proposed techniques and implementations against Perlin noise, in terms of the efficiencies and benefits of both, or how both can be used in conjunction, such as helping create the initial landmasses to helping improve the erosion simulation there are many possible use cases.

1.1. Project Background

There are many methods of creating terrain for video games. These include, but are not limited to:

- Perlin noise
- Diamond-square algorithm
- Wave function collapse
- Voronoi cells
- Marching cubes
- Simulation based generation
- Handcrafted worlds

While all of these have their own benefits, drawbacks, and use cases. These can include speed, memory efficiency, and the final appearance of the terrain created. While these methods all have their uses, most of these do not typically follow any realistic physics when generating the worlds. This can lead to terrain that is floating with no other terrain supporting it, deserts right next to oceans, or terrain that repeats itself as well as other artifacts and anomalies that you would not find in the real world.

Perlin noise is a is a noise texture that has become the standard for game development. Perlin noise was first developed by Ken Perlin in 1982 after working at Disney on the motion picture Tron [2][3] and has become the prevailing technique for creating a gradient noise, for 2D, 3D, or even 4D applications. It allows game developers to create procedural textures for 3D models, more fluid animations [4], as well as many other possibilities. It is most common use in game development stems from its ability to be layered on top of itself allowing to not only build various levels of detail but also allowing for far greater control and complexity in generating certain features such as procedural terrain.

Perlin noise is one of the most common techniques used in video game development, it is a useful tool that can be used to create worlds, textures, animations, and more in a fast, and memory efficient.

Perlin noise offers a fast, inexpensive, low memory and storage requirement, and semi realistic option for procedural terrain generation, but unfortunately its ability to make realistic worlds is limited to how many rules and individual layers of Perlin noise are implemented, which is both time and cost intensive. Another drawback is that it does not follow realistic physics so the worlds that it can generate do not always make sense. An example of these techniques in action can be seen through Minecraft's world generation [5], " Biome generation in the Overworld is based on 6 parameters: temperature, humidity (aka. vegetation), continentalness (aka. continents), erosion, weirdness (aka. ridges), and depth. Except for "depth", the other 5 parameters are based only on horizontal coordinates", Minecraft's world generation is built up of several layers of Perlin noise at different scales to help model the 6 parameters.

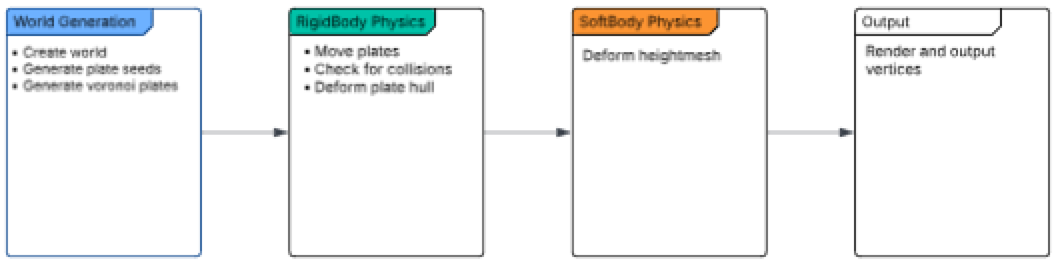
While Perlin noise is on average one the fastest to implement, to iterate different versions with, and one of the least memory intensive methods, it is not the only option for terrain generation. Another viable alternative is using pre-generated terrain that is either modelled by humans or created by a computer and then further handcrafted by humans. An example of this can be seen in games such as Ghost of Tsushima [6]. These games are typically a lot more handcrafted than games like Minecraft and are more focused on a linear storyline. Buildings, biodiversity, atmosphere, and even terrain can be changed to precise detail and specification, offering the developers the ability to change and create the world with the only limits being the computational hardware and their imagination. The limitation of the computational hardware is due to the larger disk and memory storage of these worlds as they must be stored with every little detail on the end user's device in contrast to procedural generation which only needs to store the explored or changed areas.

While there have been other options for procedural generation explored in other games such as Elite Dangerous [7], who use their "Steller Forge", to create their galaxies and star systems through first principles " Bodies are gradually aggregated over a very long simulated time from available matter, taking into account their chemical composition. Depending on the angular momentum, this might begin to form into a single central body, or into multiple co-orbiting bodies.", and appear to use a scientific model to create the planets, even modelling some tectonic plates "how this planet gets its specific, this how many mountains it should have, this is where the mountains should be, this is how active the planet is underneath the surface for the tectonic activity." [8][9]. While Elite Dangerous does offer a tectonic simulation model for their planet generation, this is still a very time intensive model, and is only generates and stores "discovered stars" on their server before sending the relevant data to the user, this helps reduce the load on both the Elite Dangerous servers, and user hardware, but makes it difficult to create infinite procedural terrain from the user end as it is a closed source software.

1.2. Project Description

A C++ program that can be used to generate a fixed area terrain for a video game world. Output in a dll format so that it can be called on through a third-party game engine such as Godot or Unity.

It will follow the steps of creating the initial world, simulating the rigid body physics, followed by simulating the soft body physics. and finally outputting as a render or set of vertices as required.



1.3. Project Aims and Objectives

Overall aim and some milestones along the way to achieve the aim

- Create a new approach to procedural terrain generation
- To match or exceed the speed of Perlin Noise
- To match or exceed the memory efficiency of Perlin noise
- To create a system that can be used in large game engines such as Godot or Unity
- To create landscapes and terrain that rely on a tectonic simulation

1.4. Project Scope

This project scope is that of a fixed state terrain generation, that is able to show case result of the tectonic methods as an alternative solution to world generation. This project does not aim to model hyper realistic terrain, it will not be able to model the likeness of Earth, but can be used to generate earth like planets, it should not be used in any scientific capacity due to the limitations of the implementation, and possible inaccuracies.

1.5. Thesis Roadmap

This report will detail the steps involved in researching, designing, testing, and building the tectonic simulation program that will be outlined in this report.

Firstly this report will go through the current technologies that are similar to this project. It will then detail the domain specific research required to be undertaken as this project falls outside of the typical scope of a computer science degree.

This report will then go through the design methodology of this project, as well as the feature table that was used to outline the requirements for this project as well as provide an overview of the system.

It will then detail the testing and experimentation done to achieve this project. The report will outline some problems that were encountered as well as detail some of the solutions undertaken to mitigate or solves these issues. It will also pose alternative solutions to the issues found as well as the current solutions were chosen.

After this, this report will detail both the testing and evaluation of the project, before introducing both the conclusion and the possibility of future work that could be implemented into this project given more time and resources.

2. Literature Review

2.1. Introduction

In this chapter I will explore other work that has attempted to tackle terrain generation through tectonic generation methods. I will look into the technologies that I have chosen to use for this implementation. As well as showcase some of the domain specific research that had to be undertaken.

2.2. Alternative Existing Solutions

There have been several prior solutions that looked to tackle problems similar to those I stated in our initial introduction one such example is detailed in a paper by L.Viitanen [10], where they follow the following steps in their algorithm: plates are moved, check for collisions, collision rules are applied, checks for if plates joined or not, filled empty spots with oceanic crust attaching to the plate that was there last. While this does manage a very impressive terrain simulation, it does mention some of the artifacts and issues that this paper hopes to improve upon, including sea floors being unnaturally smooth, as well as shorelines and mountain ranges lacking natural "jaggedness".

One method of creating procedural terrain is through erosion which can create natural jaggedness of mountains and shorelines. S.Lague cover this in their "Coding Adventure: Hydraulic Erosion" [11] using a water drop simulation. While their method works well for a fixed area terrain it would not translate well into an infinite terrain for several issues including complexity, and resource intensity. An alternative approach can be found using gradient descent and mixing it with fractals in a similar approach to what is detailed in "Better Mountain Generators That Aren't Perlin Noise or Erosion" [12]. Both methods offer their own benefits and draw backs and should be taken into consideration for continuing to create a more detailed world generation.

2.3. Technologies Researched

The main technologies that will be used throughout this project are C++ and the graphical framework Raylib which will primarily be used for debugging.

C++ was chosen due to speed, and memory requirements as I hope to implement a solution that can match the speed and efficiency of Perlin noise. I also chose C++ due to it still being an OOP language so that I can use more higher-level techniques which should help speed up development. Another benefit of compiling to C++ is that I will be able to create dll files which can be used through both Godot and Unity allowing this solution to be universal in its approach.

Raylib was chosen as it allows me to work a layer above APIs like OpenGL or Vulkan which helps reduce boiler plate while debugging and testing.

Some other technologies I have researched include Unity and Godot, and how to best implement the world generation.

Unfortunately, due to the requirement of getting both the speed and memory efficiency as close as possible to that of Perlin noise neither of these game engines were a good choice. Unity also carried the drawback of a long compile time which went against the Agile methodology.

I have also looked into python and java, which due to these both being higher level programming languages would speed up the development time. Unfortunately, this advantage was outweighed by their lack of speed when compared to C++, along with the fact that Unity does not support python or java which would hinder this project from being used on any game engine unlike compiling C++ to a dll.

2.4. Other Relevant Research

As this topic falls outside of the typical computer science educational curriculum, additional domain research was required to be researched so that a more thorough and accurate implementation can be achieved. While a lot of the information that is used can be adapted from Artifexian "Worldbuilder's Log" [1], a lot of additional research is still required.

The National Oceanic and Atmospheric Administration (NOAA) has an article detailing the 3 distinct kinds of tectonic boundaries [13] as well as some of the features that occur along these boundaries and upon the plates themselves, these include several types of volcanoes, different types of ridges and trenches, as well as continental rift zones.

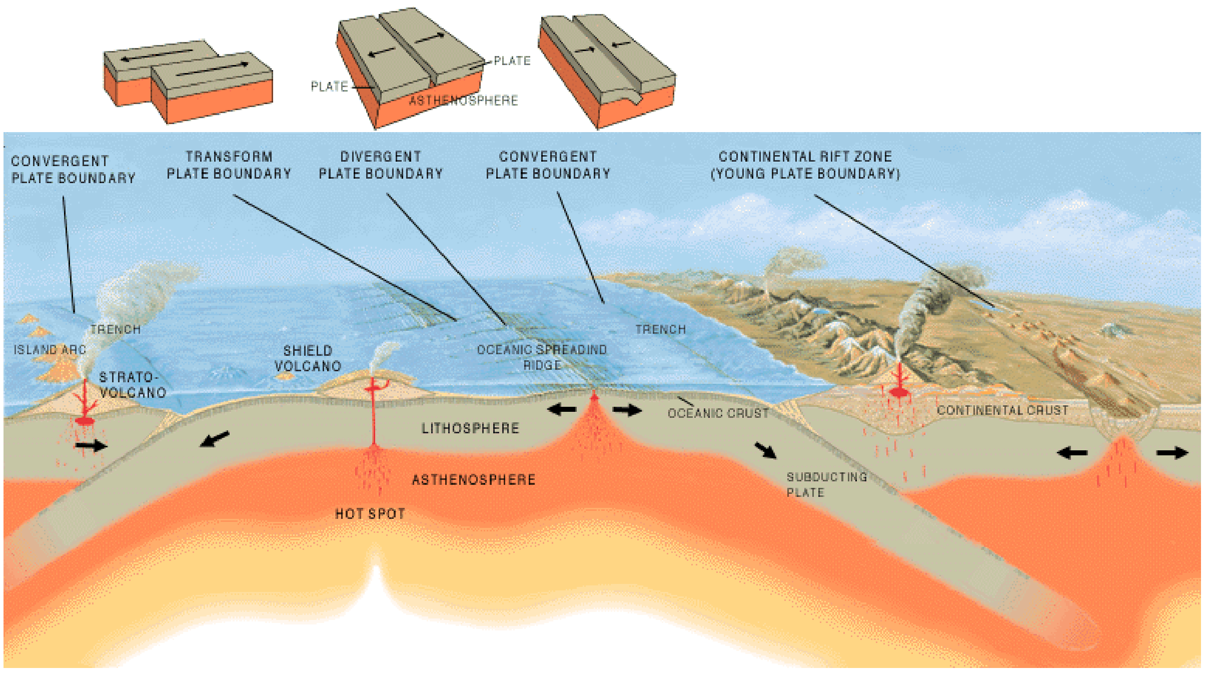


Fig 1, types of plate boundaries [14]

Prepp [15] also gives me a list of some of the types of mountains that can form on the continents, these include fold mountains, fault-block mountains, and dome mountains.

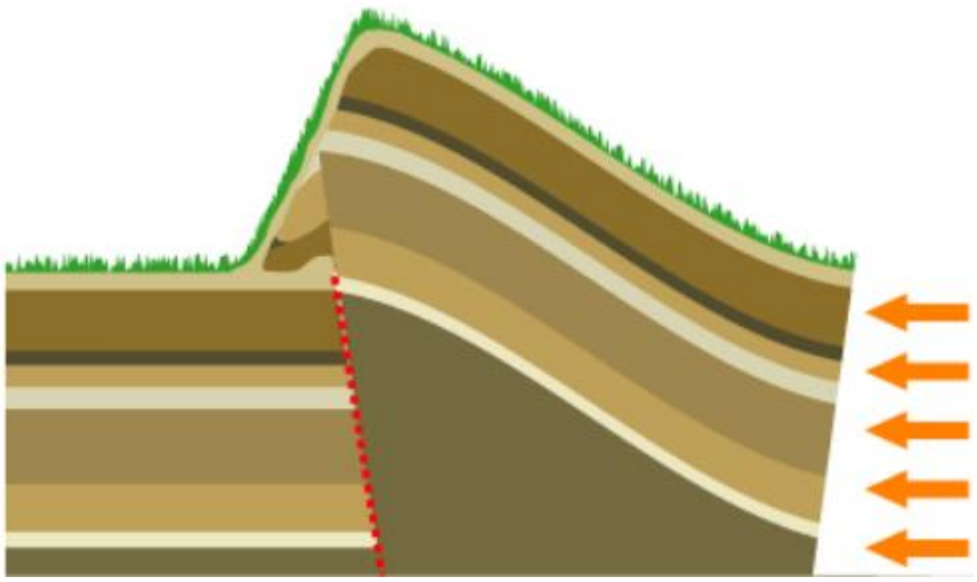


Fig 2, Fold mountains [16]

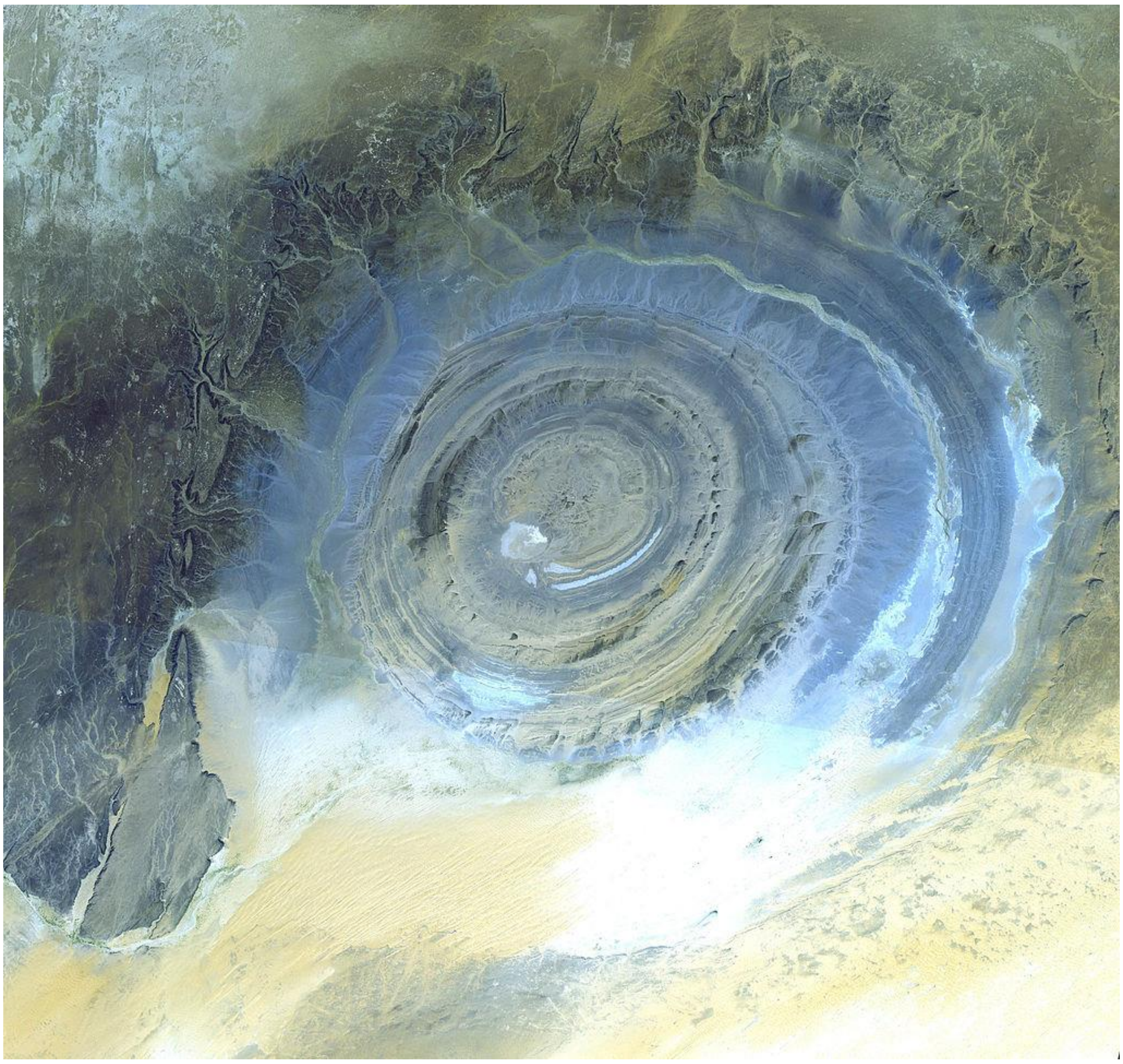


Fig 3, Dome mountain [17]

2.5. Existing Final Year Projects

There are currently no existing projects that look to tackle a similar problem in the TUD library that I have found, there are projects that do handle procedural generation.

[18] " Procedural Generation System for a 2-Dimensional Video Game World" by B.Willis uses Perlin noise to generate procedural world consisting of rooms for a 2D video game. While their method and results are different to what I plan to create for the procedural terrain simulation, it does help offer insight into how Perlin noise can be used to generate procedural maps, and a method of creating connections between the generated Perlin noise texture.

[19] " EvolVR - Investigating Procedural Ecosystems and Evolution" by R.Bryne also uses Perlin noise in their planet generation. They use Perlin noise to displace the planet vertices away from the center

of the planet which allows them to create mountains and valleys. This gives us a look into how Perlin noise can be used to generate unique worlds quickly.

2.6. Conclusions

This topic has a lot of potential but requires a lot of time and research to be dedicated to it to make it work successfully. While I have already done a large amount of research, I will have to continue to research as I explore both techniques and domain specific research to cover this topic. I will also have to continuously keep the time frame in mind to avoid scope creep and be able to work around any issues or setbacks that may arise.

3. Experiment / Software Design

3.1 Introduction

This section will outline the methodology selected, an overview of our system classes, a feature table describing what features we will have along with their difficulty and their priority.

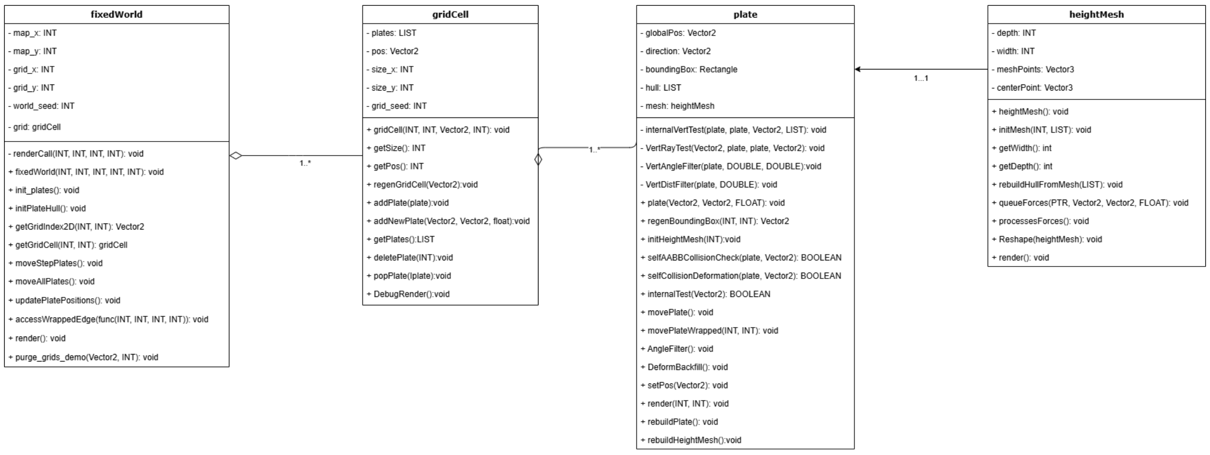
3.2. Software Methodology

I am using an Agile software methodology to quickly iterate over designs and features as I am required to test them quickly and determine their effectiveness and efficiency. This methodology will allow me to implement, test, and review my new features and prototypes quickly as I will be going over multiple different versions of the same methods and concepts.

3.3. Feature Table

Feature	Description	Difficulty Easy 1- 5 Difficult	Priority
Plate Generation	Initial plate generation	2	1
Wrap around for plates	Plates having the ability to wrap around the fixed world area i.e., going past the left edge and reappearing on the right	1	2
Collision simulation	Simulating plate collisions and movement	3	3
Plate deformation	Plate deformations as they collide and move	4	4
Height mesh deformation	how the height mesh is deformed from the collisions	4	5
Mesh Clean up	Final mesh clean up, transforming it into a useful output	4	6
World map size selection	Allowing for different world size selections and for different amounts of plates	2	7
DLL Output/exe output	Outputting to a dll or other format for use in other game engines	4	8

3.4. Overview of System


fixedWorld
- map_x: INT
- map_y: INT
- grid_x: INT
- grid_y: INT
- world_seed: INT
- grid: gridCell
- renderCall(INT, INT, INT, INT): void
+ fixedWorld(INT, INT, INT, INT, INT): void
+ init_plates(): void
+ initPlateHull(): void
+ getGridIndex2D(INT, INT): Vector2
+ getGridCell(INT, INT): gridCell
+ moveStepPlates(): void
+ moveAllPlates(): void
+ updatePlatePositions(): void
+ accessWrappedEdge(func(INT, INT, INT, INT)): void
+ render(): void
+ purge_grids_demo(Vector2, INT): void
1..*
gridCell
- plates: LIST
- pos: Vector2
- size_x: INT
- size_y: INT
- grid_seed: INT
+ gridCell(INT, INT, Vector2, INT): void
+ getSize(): INT
+ getPos(): INT
+ regenGridCell(Vector2):void
+ addPlate(plate):void
+ addNewPlate(Vector2, Vector2, float):void
+ getPlates():LIST
+ deletePlate(INT):void
+ popPlate(Iplate):void
+ DebugRender():void
1..*
plate
- globalPos: Vector2
- direction: Vector2
- boundingBox: Rectangle
- hull: LIST
- mesh: heightMesh
- internalVertTest(plate, plate, Vector2, LIST): void
- VertRayTest(Vector2, plate, plate, Vector2): void
- VertAngleFilter(plate, DOUBLE, DOUBLE):void
- VertDistFilter(plate, DOUBLE): void
+ plate(Vector2, Vector2, FLOAT): void
+ regenBoundingBox(INT, INT): Vector2
+ initHeightMesh(INT):void
+ selfAABBCollisionCheck(plate, Vector2): BOOLEAN
+ selfCollisionDeformation(plate, Vector2): BOOLEAN
+ internalTest(Vector2): BOOLEAN
+ movePlate(): void
+ movePlateWrapped(INT, INT): void
+ AngleFilter(): void
+ DeformBackfill(): void
+ setPos(Vector2): void
+ render(INT, INT): void
+ rebuildPlate(): void
+ rebuildHeightMesh():void
1...1
heightMesh
- depth: INT
- width: INT
- meshPoints: Vector3
- centerPoint: Vector3
+ heightMesh(): void
+ initMesh(INT, LIST): void
+ getWidth(): int
+ getDepth(): int
+ rebuildHullFromMesh(LIST): void
+ queueForces(PTR, Vector2, Vector2, FLOAT): void
+ processesForces(): void
+ Reshape(heightMesh): void
+ render(): void

3.5. Conclusions

In conclusion, this project follows an Agile software methodology to ensure that features can be added and tested quickly. I will implement features in order of priority as stated in the feature table while building the software to the UML class diagram that is described above.

4. Experiment / Software Development

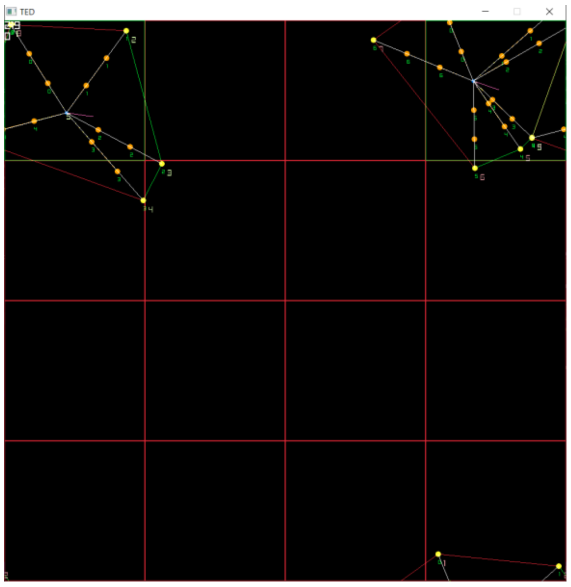
4.1. Introduction

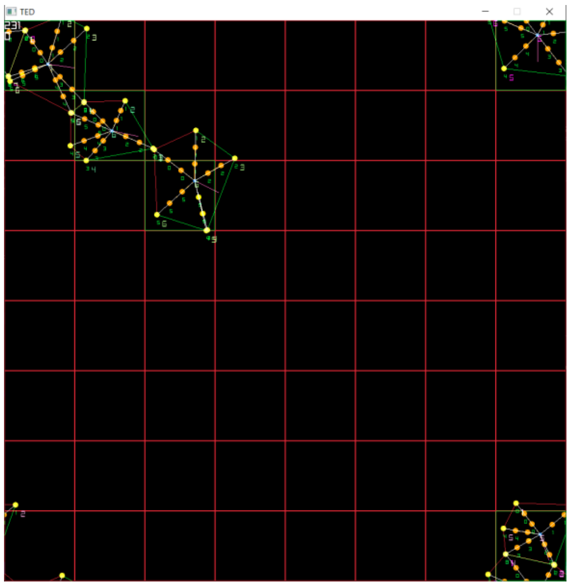
The development of the software required multiple different steps to achieve the required results. and was constantly iterated upon until a final solution was achieved.

4.2. Software Development

Grid cell generation

As this software needed to account for wrap around from when plates reached one side of the map and appeared on the other a grid cell approach was used





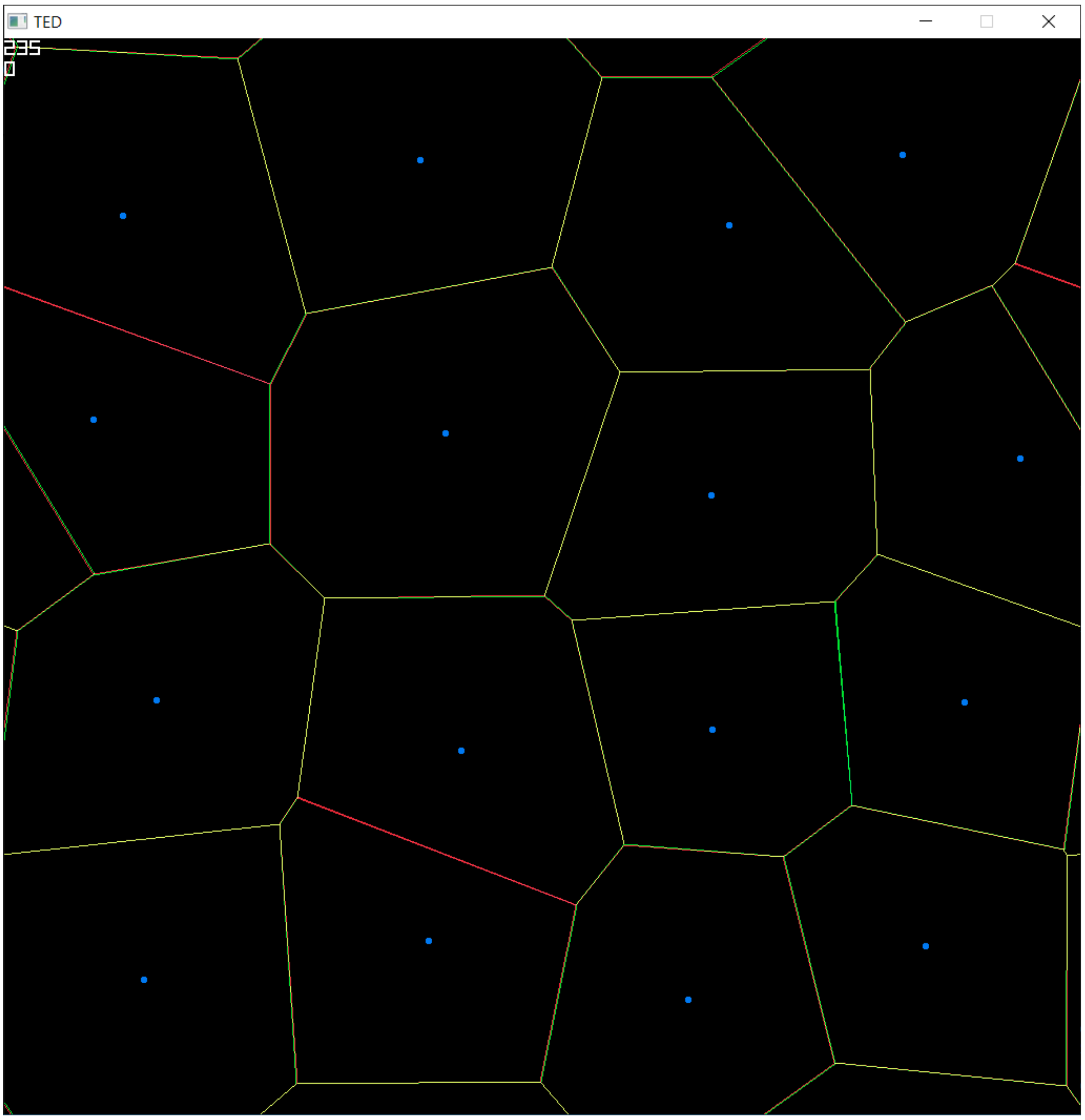
Both images are 1024x1024 pixels but have a varied amount of grid cells.

In the left image we have a world with a 4x4 grid cells approach, we can also see that the top right plate is also being wrapped around and shown on the bottom right of that world.

In the right image we have an 8x8 with several plates isolated. The larger amount of grid cells allows us for a varied amount of grid cells based on our technical requirements. with a higher amount of grid cells being far more useful for larger amounts of plates compared to fewer grid cells being better for larger plates overall.

Initial plate generation

For the initial plate generation, a Voronoi approach was used as it allowed us to generate unique plates based on randomly scattered seeds inside the grid cells



The vertices of these plates also need to be generated in a winding order as this allows us to later determine if a certain point is inside or outside a plate

```
START VORONOI_CONTRUCTOR

// Define winding order offsets
DEFINE array lx = [-1, 0, 1, 1, 1, 0, -1, -1, -1]
DEFINE array ly = [-1, -1, -1, 0, 1, 1, 1, 0, -1]

// Iterate through every grid cell
FOR each x FROM 0 TO grid_x - 1
    FOR each y FROM 0 TO grid_y - 1

        // Get the plate from the current grid cell
        SET p TO first plate in grid[x][y]
        DECLARE array pb_array of size 9 // To hold perpendicular bisectors

        // Getting perpendicular bisectors
        FOR each l FROM 0 TO PH_COUNT - 1
```

```
            // Find offsets for wrapped points
            SET offset_x TO -map_x IF (x + lx[l] == -1), ELSE map_x IF (x + lx[l] == grid_x), ELSE 0
            SET offset_y TO -map_y IF (y + ly[l] == -1), ELSE map_y IF (y + ly[l] == grid_y), ELSE 0

            // Get neighbouring plate
            SET p2 TO first plate in grid[(grid_x + x + lx[l]) MOD grid_x][(grid_y + y + ly[l]) MOD
grid_y]

            // get perpendicular bisector and store in array
            SET pb_array[l] TO PerpendicularBisector(p.position, (p2.position.x + offset_x,
p2.position.y + offset_y))



        END FOR

        // Find the intersection points
        FOR each i FROM 0 TO PH_COUNT - 1
            SET i1 TO Intersect(pb_array[i MOD PH_COUNT], pb_array[(i+1) MOD PH_COUNT])
            SET i2 TO Intersect(pb_array[(PH_COUNT + i - 1) MOD PH_COUNT], pb_array[(i+1) MOD
PH_COUNT])
            SET i3 TO Intersect(pb_array[i], pb_array[(i+2) MOD PH_COUNT])

            // Check for valid vectors
            IF i1, i2, and i3 are all invalid
                PRINT "FAILED MULTI"
            END IF

            // Find closest intersection point
            SET d1 TO distance(p.position, i1) IF i1 is valid, ELSE maximum float
            SET d2 TO distance(p.position, i2) IF i2 is valid, ELSE maximum float
            SET d3 TO distance(p.position, i3) IF i3 is valid, ELSE maximum float

            IF d1 is the smallest SET res TO i1
            ELSE IF d2 is the smallest SET res TO i2
            ELSE SET res TO i3

            // Adjust result relative to plate position
            SET res TO (res.x - p.position.x, res.y - p.position.y)

            // Check for duplicates in hull
            SET temp_skipper TO FALSE
            FOR each v IN p.hull
                IF res equals v
                    SET temp_skipper TO TRUE
                    BREAK
                END IF
            END FOR

            // Add result to hull if unique
            IF NOT temp_skipper
                ADD res TO p.hull
            END IF
        END FOR
    END FOR
END FOR


END VORONOI_CONTRUCTOR
```

Moving plates & collisions

for each timestep all of the plates need to be moved according to their speed and direction.

This is done by simply looping over the grid cells and then by looping through all the plates currently in the grid cell and applying the move function to them.

The movement system follows the following steps

1. Move all the plates
   We loop through all the grid cells and move all the plates inside them

2. We back fill by extruding the plate in the opposite direction of its movement
   This allows us to fill in any gaps that may have formed when the plates have moved
3. Checks the grid-based distance filter

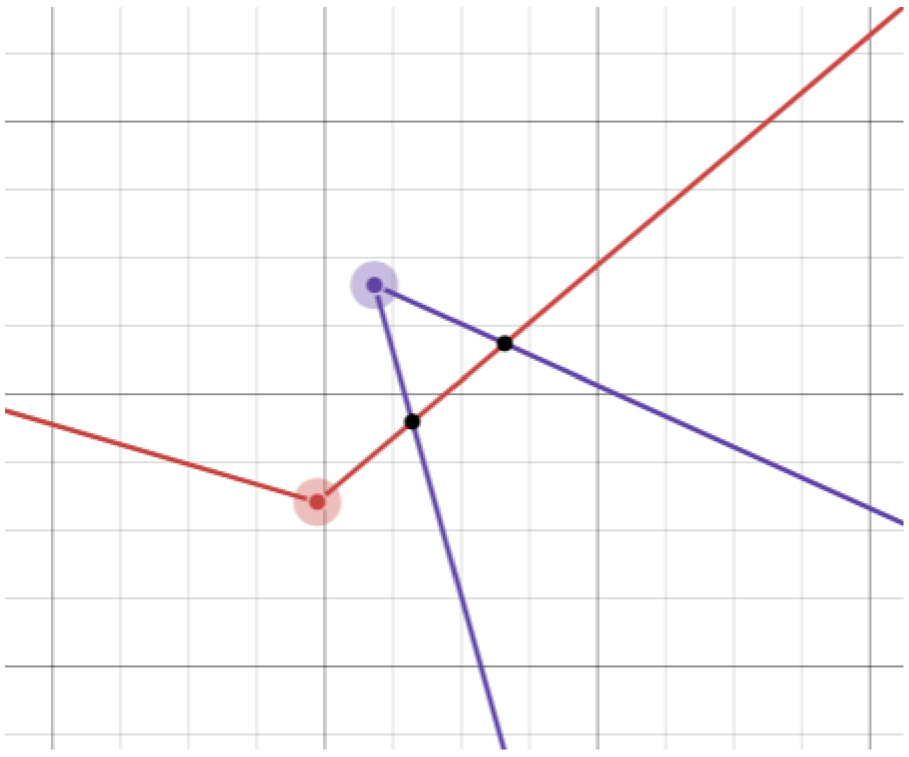
   The grid-based filter checks the neighbouring cells of the current cell the plate is in. This allows us to reduce the number of plates that need to check against in the next step by only checking the neighbouring cells' plates

4. Checks the Axis Aligned Bounding Box (AABB) detection

   AABB detection uses a boundary box that is aligned to the coordinate axis. This bounding box is created by checking the width and height of the plate, which can then be used alongside the plates position to determine if it intersects with another plates bounding box. This allows for a fast and cheap collision detection.

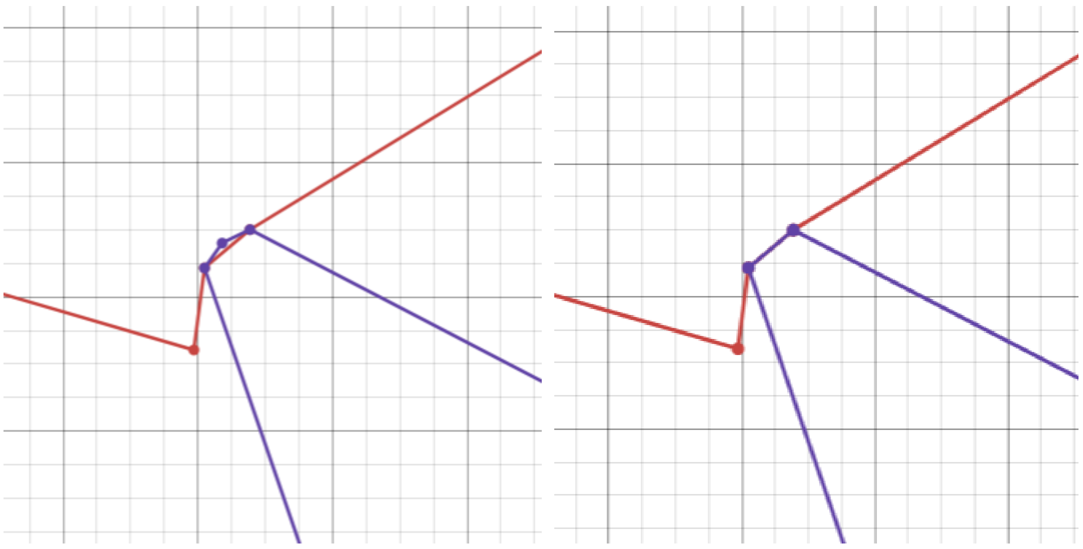
5. We then perform our collision deformations

   We first need to check for any lines that intersect and where they intersect using the point of intersection formula



Point of intersection formula [20]:

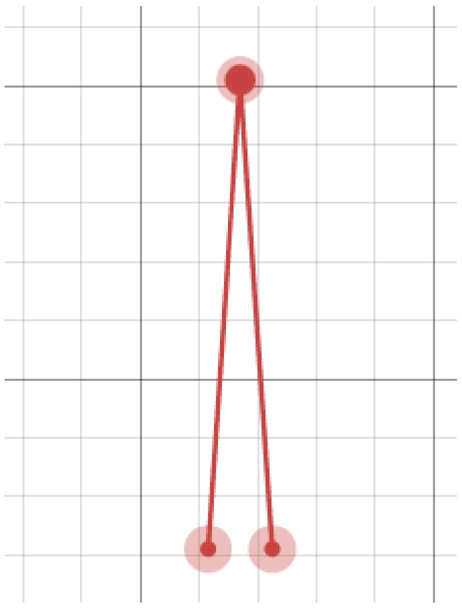
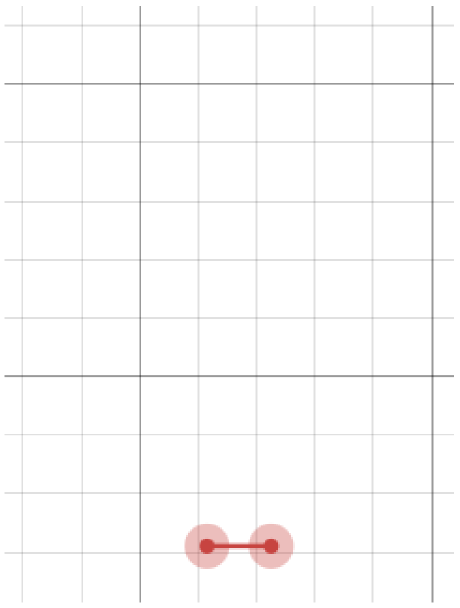
(x, y) = ((b1*c2-b2*c1)/(a1*b2-a2*b1), (c1*a2-c2*a1)/(a1*b2-a2*b1))



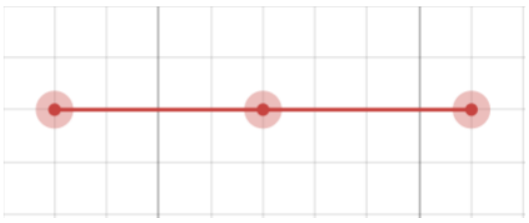
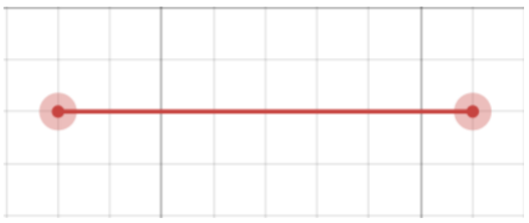
This allows for the creation of new vertices for the plates. These vertices can also be offset based on the momentum of the plates, as seen in the left image.

After this we can remove any vertices that are inside the other plate. We determine if a vertex is inside another plate if we can cast a ray to the plates center and check how many edges it intersects, if the number is even or 0, we can remove it as its inside the other plate. This approach was chosen as it works for both concave and convex shapes, but it is sometimes inaccurate or has odd exception cases such as if a point is on a line

Currently our only method of removing vertices is when they are inside another plate, and while this does remove some vertices it will still lead to memory issues. This is why we also include two other methods of removing unnecessary vertices. The first method is an angle filter that defines a maximum and minimum angle between three vertices, allowing us to remove vertices if an angle becomes too sharp or shallow. In the example of the image on the bottom left we have an angle which is too sharp so we must remove the point leading to the result on the left. We do this to avoid any random spikes that may occur when the plates are compressed at certain angles.

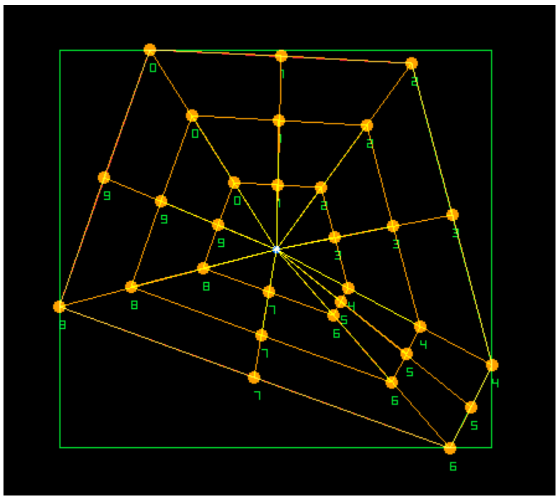
Another reason for needing the angle filter is for longer lines of very shallow angles, or straight lines with multiple points on them as can be seen on the bottom left image.

These filters help reduce the memory requirements of each plate and tends to reduce the number of vertices needed on any one plate to below 20 on average.

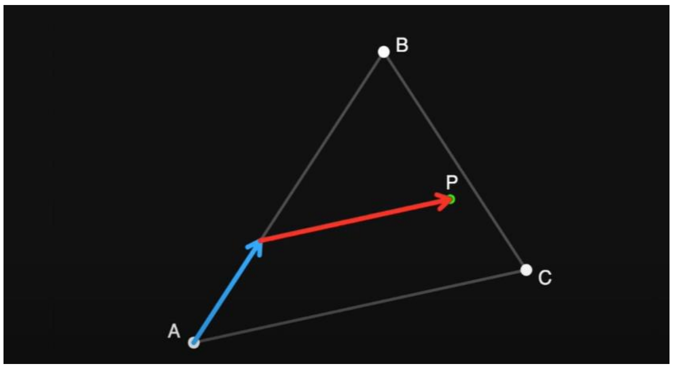
The second filter we use to help reduce the number of vertices is the distance filter, this filter takes the distance between itself and its two neighbours and based on the combined distance determines if it should be removed or not. This filter requires a lot of testing as you need to ensure that the distance is sufficient to capture any excessive vertices, but not overly aggressive so as not to remove necessary defining vertices.

These functions handle the deformations of the plate hull, but these do not affect the actual height of the plate or form any mountains or ridges, that is handled by the soft body like height mesh.



The Image on the left shows an example of the height mesh (Orange), this height mesh is used to determine the height of certain points which can then be used to build a mesh for the terrain for a video game.

As the outer edges of the mesh are not link to the plate hull, they can often separate. To avoid this, we can create a new mesh on the next timestep based on the current hull shape, then get the heights of closest points from the previous mesh using a method that is usually used to check if a point lies within a triangle.



The image on the right shows a point within a triangle. Using the formula found in the video by S.Lague[23] we can use the values that would typically help us determine if the point is within a triangle to determine the weighted height of the new point in relation to three of the points from the previous mesh.
This method works well for all the points leading up towards the center of the height map. The only exception to this is the points on the outer edge of the height map, these are simpler as they only require a weighted average as there is only two points.

This is a very simple approach for creating a height mesh for the plates, and was chosen due to the time constraints and scope of the project, the

4.3. Exploration of other methods

While this project has chosen a specific approach and methodology, other approaches have also been explored throughout the testing and design process.

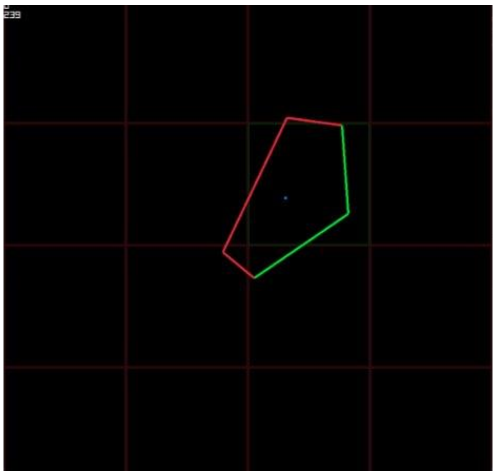
One noteworthy example is the use of AABB collision detection in place of SAT (Separating Axis Theorem) collision detection. SAT is especially useful for detecting collisions between polygons such as the ones that are being created in the simulation, but it does carry drawbacks. These drawbacks include that it is only effective for convex polygons and not concave polygons, it is slower than AABB as it has to project each of the vertices onto a hyperplane that is parrel to an edge, for every edge, compared to AABB which needs to only check the minimum and maximum widths and heights of the shapes.

Another example is the use of a ray to determine if a vertex is inside or outside a neighbouring polygon. Another method that was explored is the use of cross product and the winding order of vertices to determine if a vertex is to the left or right of the line segment, unfortunately this had drawbacks of concave polygons as when edges buckled they created spaces that would have been counted as outside the polygon despite being inside it. While the ray cast approach does help minimise this problem it does not solve it entirely. An example of this is when the center of the polygon lies outside of itself such as that of a "C" shaped polygon.

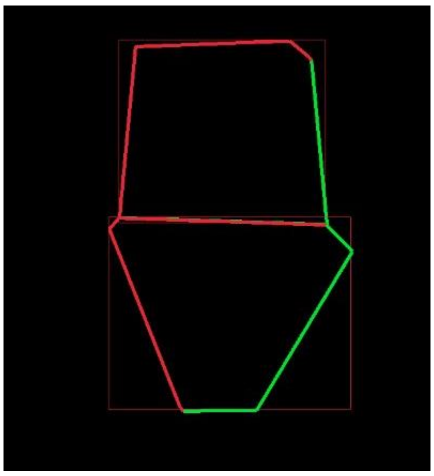
The final output was decided to be that of a simple PNG heightmap as it was faster and simpler to implement into Godot due to the time constraints. Other options for this would have been compiling it alongside Godot and passing through a list of vectors which would have been used to build the terrain.

4.4. Optimisations

As previously mentioned, this project employed a grid-based distance system to help reduce the number of plates need to be compared against each other as can be seen on the below on the left-hand side.



Grid based system

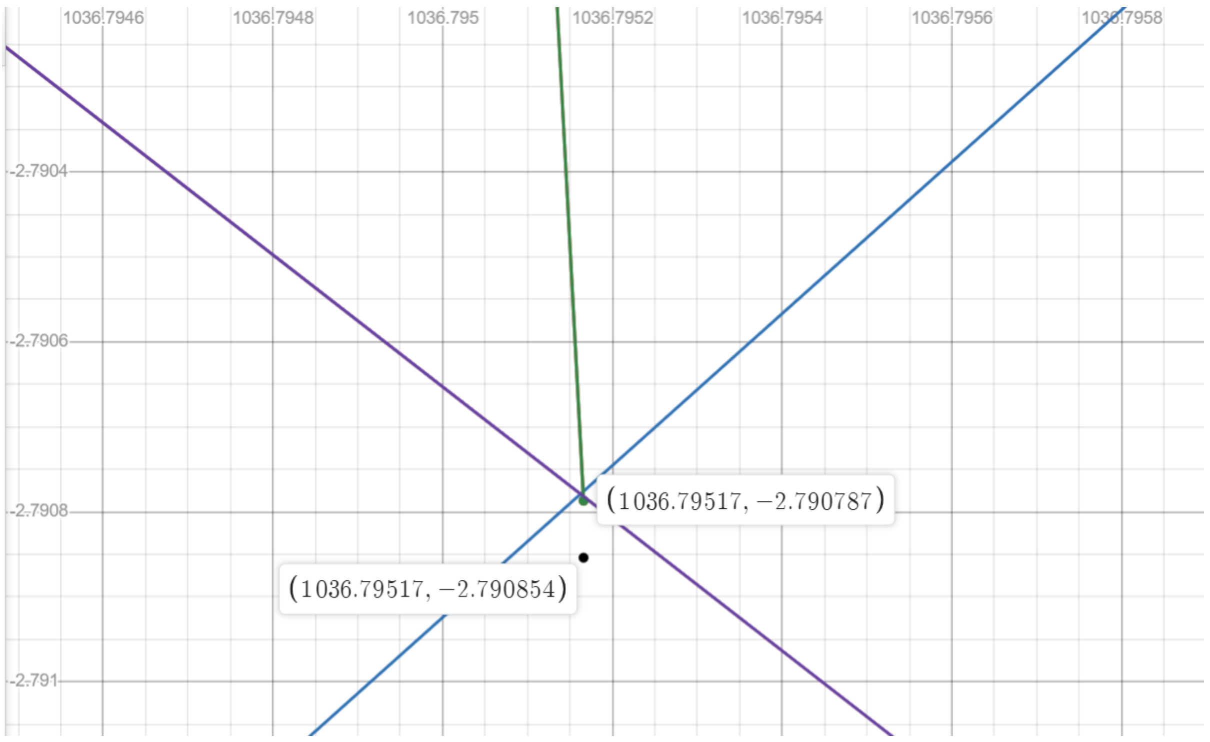


AABB Collision detection

This project also uses AABB collision detection to further reduce this, this was chosen instead of SAT collision detection as it offered a simpler and faster detection, an example of this collision system can be seen in the image above.

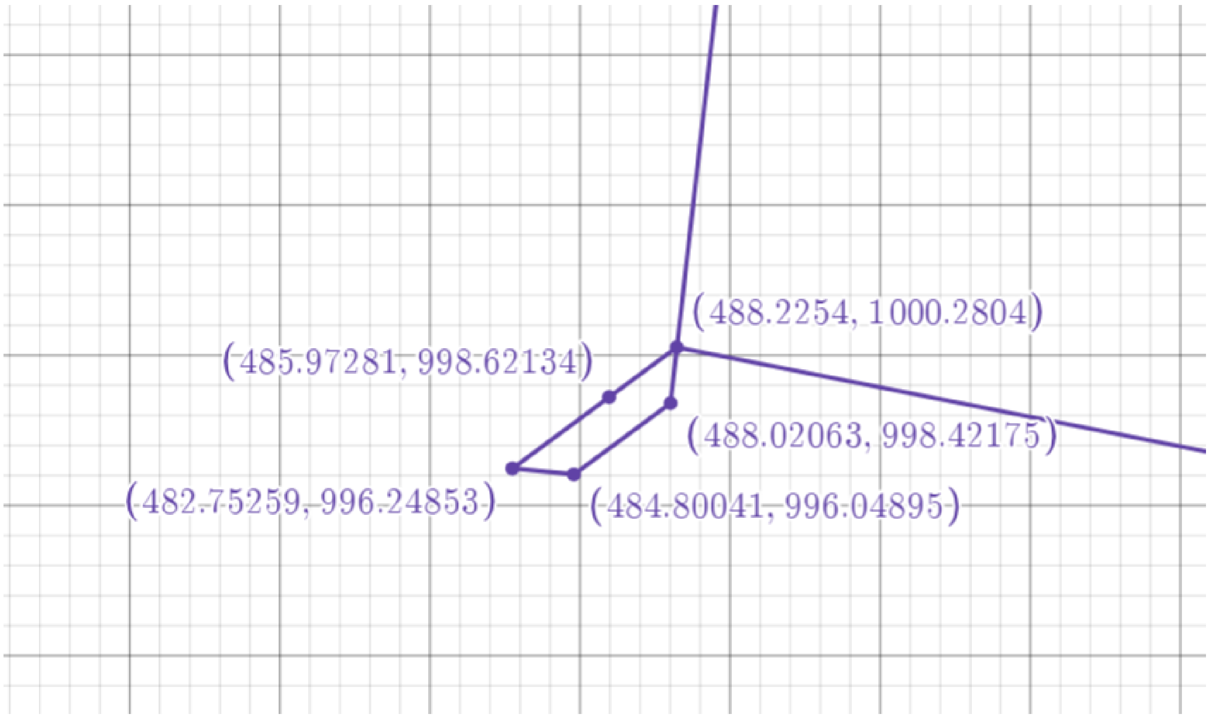
This project also used the bounding boxes of AABB to help reduce the number of vertices that needed to be checked from other plates scaling up the bounding box slightly and checking which points are inside.

4.5. Potential Issues and solutions



Floating point error can cause certain calculations to give an output that is slightly off and causing the program to not work as intended i.e. in the image above where we have a difference in Y value of 0.000067. This causes an issue where a new intersection point is not created, which causes problems later on when the vertices are checked if they are internal or external as any internal vertices are removed.

One potential solution is to create a boundary area where the new intersection point, even if it is not on the line segment is still allowed to be created. This helps avoid the issue but creates a further issue.



As can be seen above, the continuous accumulation of the errors mentioned above creates self-intersecting areas in the polygon. When self-intersection occurs, these errors can grow exponentially and cause issues for other parts of the simulation.

Some solutions include checking the distance between vertices to see if any vertices are close to each other. This approach helps minimise the frequency of these errors, but this has its own drawbacks and does not eliminate the risk completely. The main drawback of this approach is that it requires a lot of fine tuning due to the risk of it being overly eager and removing vertices where they are still needed.

Another approach is using the Bentley–Ottmann algorithm [21], which is a sweeping line algorithm which can be used to find the points of intersection.

This program will use a rebuilding approach where every few time steps the polygon is rebuilt using the minimal defining vertices of the polygon. This approach helps avoid the majority of issues caused by floating point errors or other issues and helps create a fall back if these problems do occur.

4.6. Conclusions

In conclusion, this program has a lot of parts that are interacting with each other, from the initial world generation to both the rigid body and the soft body physics simulation. Due to the large number of parts and intersections the possibility of floating-point errors grows exponentially and when one occurs the effects stack and causes more issues over time.

5. Testing and Evaluation

5.1. Introduction

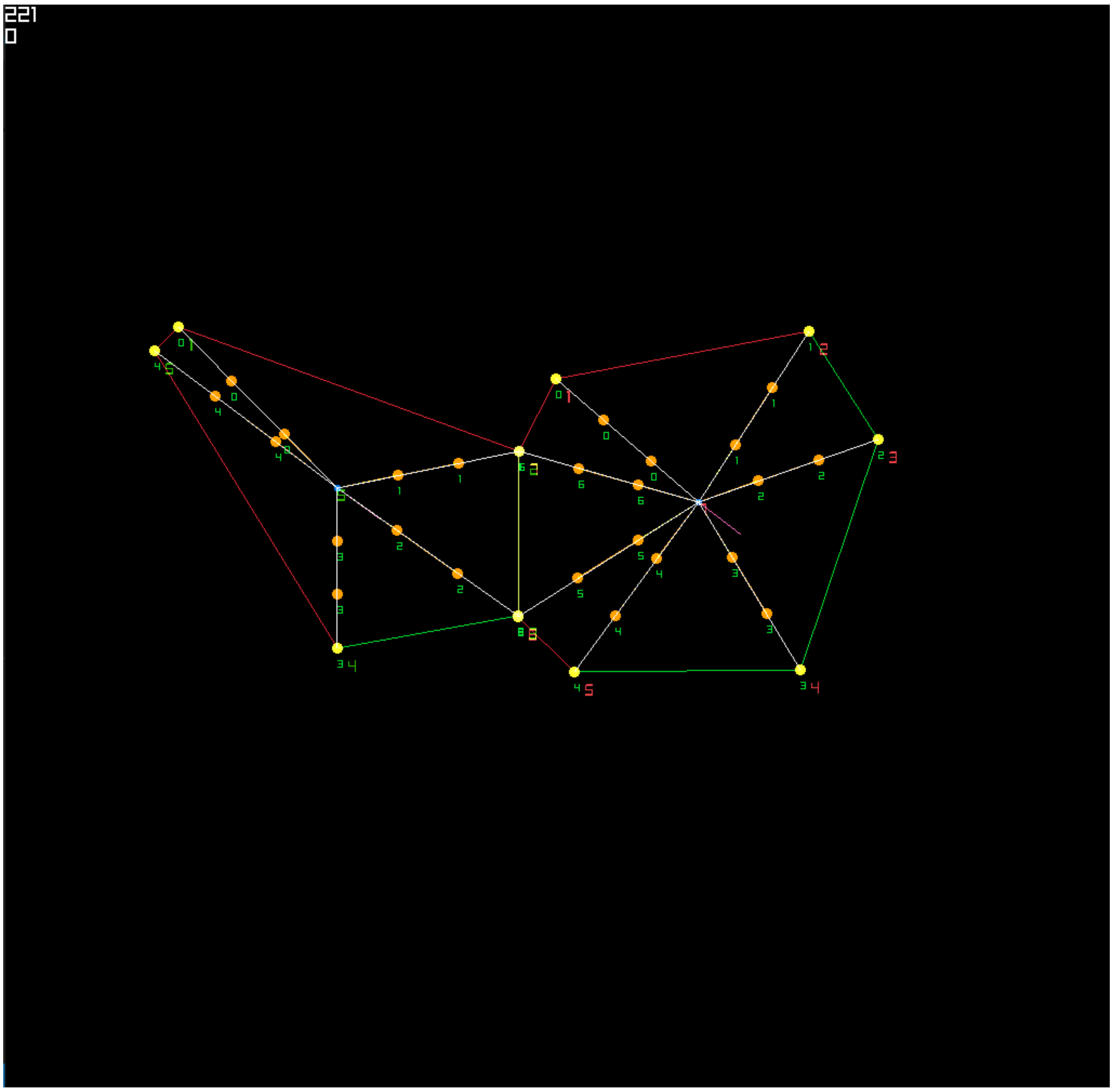
Testing and evaluation is a valuable factor that must be considered while completing this project. The main points I need to test for include, speed and memory efficiency, especially in comparison to Perlin noise. I will also attempt to do some user testing to ensure that the final application is easy to use with the appropriate game engines as an additional tool

5.2. System Testing

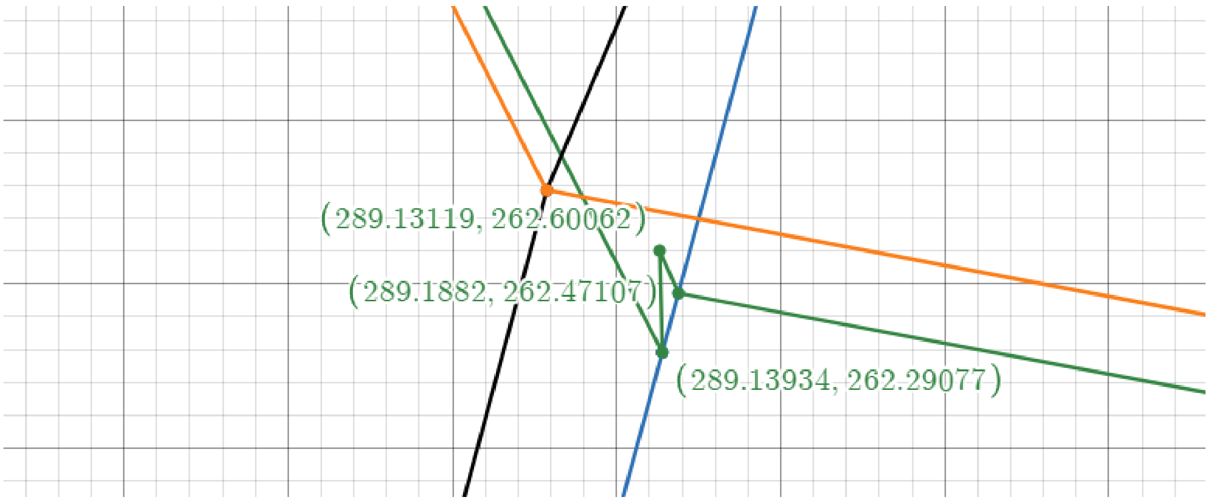
This project follows an Agile methodology and will need to be continuously redesigned, developed, tested, deployed, and reviewed at each step of the development process. For testing I needed to ensure that the application remains fast and memory efficient at each step so as to reduce the inevitable slowdown as features are added.

Each of the features need to be tested separately and once again together with all of the other features that can be affected by it. In Isolation these features will be tested through their own independent test functions. When testing the functions against other functions a series of debugging functions need to be made, these include isolating plates, moving the plates a certain amount of timesteps forward, and moving the plates around, as necessary.

All of these tests are conducted using the Raylib Framework as well as Desmos so as to reduce the compile times and allow for faster testing. Desmos is used as it allows for an easier method of looking at individual vertices and edges and determining where the issues may have arisen.



Example of isolating plates for testing collision and deformation between two plates. This allows to evaluate the collisions in a less cluttered manner as with many plates it becomes difficult to assess what is happening at any one moment in time.



One of the best tools that was used in the creation and testing of this project was Desmos [22]. This tool allowed for checking vertices and edges at far closer levels as the vertices of the plates could just be pasted into Desmos and then zoomed into. As can be seen above, there is a difference of less than 1 between all the vertices which would have been impossible to make out without some sort of zoom feature which was outside the scope of this project. This allowed for faster development and testing as there was no need for additional development time as the tool was already there.

5.3. System Evaluation

The evaluation of this project was a bit more complicated than most projects as there were very few obvious points to evaluate. The primary points of evaluation included speed of the program, perceived realism of the final output, and ease of use in other platforms such as Godot or Unity.

The programs speed is tested by running a certain amount of timesteps for specific amounts of plates. These tests were ran 3 times for each of the following and the time was averaged.

Amount of plates	Size of world	Steps	Average time (ms)
4x4	2048x2048	200	553.06
8x8	1024x1024	200	1744.21
4x4	1024x1024	100	389.07
4x4	1024x1024	200	546.94
4x4	1024x1024	300	711.94
4x4	1024x1024	400	895.79
4x4	1024x1024	500	1059.77
4x4	1024x1024	0	158.00
8x8	1024x1024	0	243.43

As we can see the biggest contributor to any increase in simulation time is the total amount of plates. If we remove the initial load times from both of 8x8 and 4x4 simulations as seen in the 0 step examples, we get:

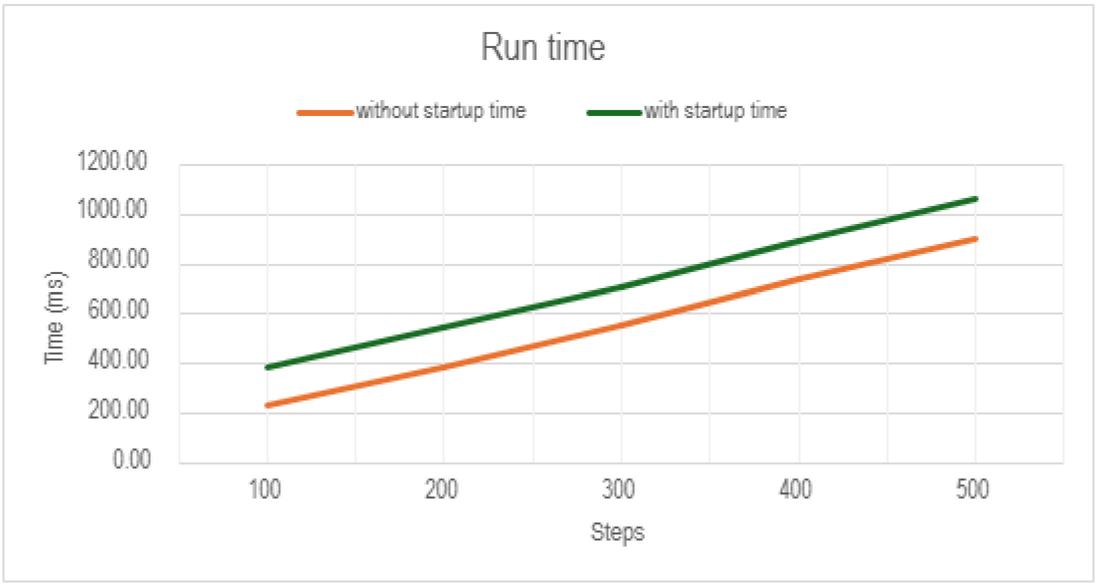
546,94 - 158.00 = 388.94
1744.21 - 243.43 = 1500.78
1500.78 / 388.94 = 3.86

The 3.86 times matches our expected result of 4 times as we have quadrupled the number of plates that are being simulated.

Additionally, the fact that changing the final render resolution does not affect the process time is expected as the plates are done using vertices and the size of the output only matters at the very end.

Below we can also see that the time taken increases in a linear fashion with the number of steps.

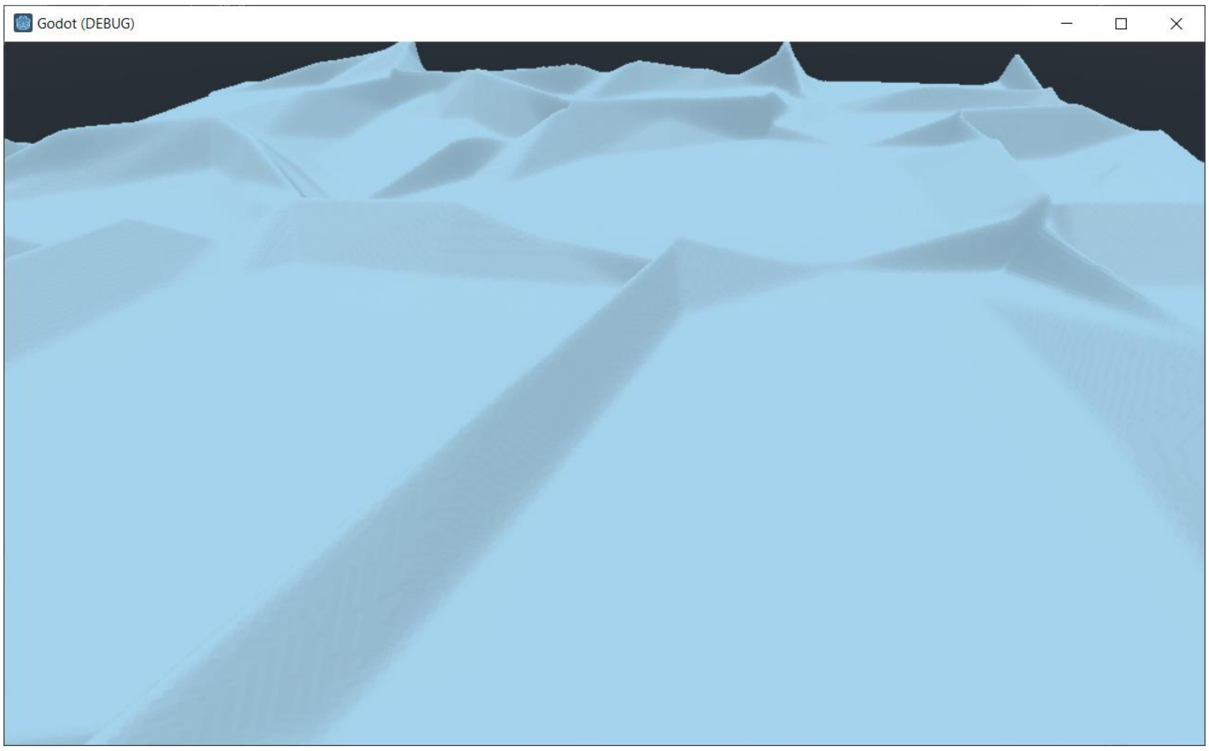
This is the expected result as the number of plates on average shouldn't change and the number of vertices should stay on average the same also.



5.4. Conclusions

This project requires a lot of testing and evaluation throughout its entire development process as we are using an agile methodology which will allow us to iterate rapidly throughout different versions of this project.

6. Conclusions and Future Work



heightmap created through the use of the application and used in Godot

6.1. Introduction

Information about potential work and features that can be done or added to this project as well as the final conclusive remarks.

6.2. Future Work

While this project covered the basic aspects of terrain generation through tectonic simulation, there is still a lot of work that can be done with this project.

Some aspects which that were not covered due to time constraints and scope of the project include

- Splitting and merging of plates
- Differentiating between oceanic and continental plates
- Plate speed and direction being affected by a separate mantle simulation
- Effects of Mountains and valleys on a separate air current simulation
- Erosion through rivers, oceans, or other events
- Using multiple threads
- Procedural generation with infinite scrolling

6.3. Conclusions

In conclusion, we see that the amount of plates has the largest impact on the time taken to process a timestep, and we can also see that it grows at the same rate as the amount of plates, we also see that the time taken grows linearly in respect with the amount of timesteps taken. We also see that the actual rendering of the plates does not affect the final processing time.

Bibliography

[1] Artifexian, "Artifexian," *YouTube*, 2024. Available: https://www.youtube.com/@Artifexian. [Accessed: Nov. 23, 2024]

[2] K. Perlin, "Chapter 4 In the beginning: The Pixel Stream Editor." Available: https://userpages.cs.umbc.edu/olano/s2002c36/ch04.pdf. [Accessed: Nov. 23, 2024]

[3] K Perlin, "An image synthesizer," *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 287–296, Jul. 1985, doi: https://doi.org/10.1145/325165.325247 [Accessed: Nov. 23, 2024]

[4] B. Bodenheimer, A. Shleyfman, and J. Hodgins, "The Effects of Noise on the Perception of Animated Human Running," 1999. Available: https://repository.gatech.edu/server/api/core/bitstreams/248afd31-c14d-41a9-9d4b-72c66056e543/content. [Accessed: Nov. 23, 2024]

[5] "World generation – Minecraft Wiki," *Minecraft Wiki*, 2022. Available: https://minecraft.wiki/w/World_generation. [Accessed: Nov. 23, 2024]

[6] B. Bencomo, "How It's Made -- The world of Ghost of Tsushima - ESPN," *ESPN.com*, Jul. 23, 2020. Available: https://www.espn.com/esports/story/_/id/29524519/how-made-world-ghost-tsushima. [Accessed: Nov. 23, 2024]

[7] "Elite: Dangerous Newsletter #36," *Archive.org*, 2014. Available: https://web.archive.org/web/20171027110746/http://us2.campaign-archive2.com/?u=dcbf6b86b4b0c7d1c21b73b1e&id=76df98203b. [Accessed: Nov. 23, 2024]

[8] J. Williams, "How Planets are Made in Elite: Dangerous Horizons," *Wccftech*, Oct. 16, 2015. Available: https://wccftech.com/planets-elite-dangerous-horizons/. [Accessed: Nov. 23, 2024]

[9] On, "On The Horizon: How to Make a Real World," *YouTube*, 2024. Available: https://www.youtube.com/live/-Et5Ivi_yIg. [Accessed: Nov. 23, 2024]

[10] L. Viitanen, "Physically Based Terrain Generation : Procedural Heightmap Generation Using Plate Tectonics," *Theseus.fi*, 2024, doi: urn:NBN:fi:amk-201204023993. Available: https://www.theseus.fi/handle/10024/40422. [Accessed: Nov. 24, 2024]

[11] Sebastian Lague, “Coding Adventure: Hydraulic Erosion,” *YouTube*, 2024. Available: https://www.youtube.com/watch?v=eaXk97ujbPQ. [Accessed: Nov. 24, 2024]

[12] Josh's Channel, “Better Mountain Generators That Aren’t Perlin Noise or Erosion,” *YouTube*, 2024. Available: https://www.youtube.com/watch?v=gsJHzBTPG0Y&list=PLjal0X3O1Eg9eEA7KN9ylgHjsW5S2RzTD&index=5. [Accessed: Nov. 24, 2024]

[13] “What are the different types of plate tectonic boundaries?: Exploration Facts: NOAA Office of Ocean Exploration and Research,” *Noaa.gov*, 2022. Available: https://oceanexplorer.noaa.gov/facts/plate-boundaries.html#:~:text=There%20are%20three%20kinds%20of,of%20the%20U.S.%20Geological%20Survey.. [Accessed: Nov. 25, 2024]

[14] J. F. Vigil, “Earthquakes - General Interest Publication,” *Usgs.gov*, 2016. Available: https://pubs.usgs.gov/gip/earthq1/plate.html. [Accessed: Nov. 25, 2024]

[15] A. Patil, “Tectonic Mountains - Geography Notes,” *Prepp*, 2024. Available: https://prepp.in/news/e-492-tectonic-mountains-geography-notes. [Accessed: Nov. 25, 2024]

[16] T. Torao, “geological processes that underlie the formation of mountains,” *Wikipedia.org*, Nov. 18, 2004. Available: https://en.wikipedia.org/wiki/Mountain_formation#/media/File:Mountain_by_reverse_fault.gif. [Accessed: Nov. 25, 2024]

[17] “deformational feature in structural geology,” *Wikipedia.org*, Mar. 28, 2006. Available: https://en.wikipedia.org/wiki/Dome_(geology)#/media/File:ASTER_Richat.jpg. [Accessed: Nov. 25, 2024]

[18] B. Willis, “Procedural generation system for a 2-dimensional video game world,” Tudublin.ie, 2016. Available: https://library.tudublin.ie/search?/XProcedural+Generation+&SORT=D/XProcedural+Generation+&SORT=D&SUBKEY=Procedural+Generation+/1%2C5%2C5%2CB/frameset&FF=XProcedural+Generation+&SORT=D&2%2C2%2C [Accessed: Apr. 10, 2025]

[19] R. Byrne, “EvolVR: investigating procedural ecosystems and evolution,” Tudublin.ie, 2020. Available: https://library.tudublin.ie/search/?searchtype=X&searcharg=EvolVR+&sortdropdown=-&SORT=DZ&extended=0&SUBMIT=Search&searchlimits=&searchorigarg=XProcedural+Generation+%26SORT%3DD [Accessed: Apr. 10, 2025]

[20] “Point of Intersection Formula,” Unacademy, Jul. 13, 2022. Available: https://unacademy.com/content/point-of-intersection-formula/#:~:text=The%20point%20of%20intersection%20formula,of%20three%20or%20more%20lines. [Accessed: Apr. 10, 2025]

[21] Bentley, None, and None Ottmann. “Algorithms for Reporting and Counting Geometric Intersections.” IEEE Transactions on Computers, vol. C-28, no. 9, 1 Sept. 1979, pp. 643–647, ieeexplore.ieee.org/abstract/document/1675432, https://doi.org/10.1109/TC.1979.1675432 Accessed 7 Apr. 2025.

[22] “Desmos | Beautiful free math.,” Desmos.com, 2025. Available: https://www.desmos.com/ [Accessed: Apr. 10, 2025]

[23] Lague, Sebastian. “Gamedev Maths: Point in Triangle.” Youtube.com, 14 June 2017, www.youtube.com/watch?v=HYAgJN3x4GA. Accessed 13 Apr. 2025.

Appendices

#######Steps to generate the terrain

It's fairly simple to generate the terrain as all you'll need is the main.exe file, the dll files that accompany it, and have it in the right folder, as well as the code below.

####### Godot code to generate the heightmap

```
extends Spatial

# Declare all variables at the top
var heightmap_image : Image
var terrain_width = 1024
var terrain_height = 1024
var height_scale = 50.0  # Increased from 10 to make height more visible

func _ready():
    # Run external program (make sure path is correct)
       # Variables are amount of gridcells, amount of mesh layers, spread of force
                        width, spread of force depth, screen width, screen height
    OS.execute("main.exe", ['4','3', '3', '6', '1024', '1024'])


    var heightmap_texture = load("res://output.png")# Load the heightmap

    heightmap_image = heightmap_texture.get_data()
    heightmap_image.lock()

    # Debug: Check image dimensions
    print("Image size: ", heightmap_image.get_width(), "x",
                          heightmap_image.get_height())
    if heightmap_image.get_width() < terrain_width or heightmap_image.get_height()
< terrain_height:
        push_error("Heightmap image is smaller than terrain dimensions!")
        return

    # Create mesh
    var surface_tool = SurfaceTool.new()
    surface_tool.begin(Mesh.PRIMITIVE_TRIANGLES)

    # Generate terrain
    for x in range(terrain_width - 1):
        for z in range(terrain_height - 1):
            # Get heights (using red channel)
            var h1 = heightmap_image.get_pixel(x, z).r * height_scale
            var h2 = heightmap_image.get_pixel(x + 1, z).r * height_scale
            var h3 = heightmap_image.get_pixel(x, z + 1).r * height_scale
            var h4 = heightmap_image.get_pixel(x + 1, z + 1).r * height_scale
            # Create vertices
            var v1 = Vector3(x, h1, z)
            var v2 = Vector3(x + 1, h2, z)
            var v3 = Vector3(x, h3, z + 1)
            var v4 = Vector3(x + 1, h4, z + 1)

            # First triangle (v1, v2, v3)
            surface_tool.add_vertex(v1)
            surface_tool.add_vertex(v2)
            surface_tool.add_vertex(v3)

            # Second triangle (v2, v4, v3)
            surface_tool.add_vertex(v2)
            surface_tool.add_vertex(v4)
            surface_tool.add_vertex(v3)

    # Generate normals for proper lighting
    surface_tool.generate_normals()

    # Create and display mesh
    var mesh_instance = MeshInstance.new()
    mesh_instance.mesh = surface_tool.commit()
    add_child(mesh_instance)

    # Center the terrain
    mesh_instance.translation = Vector3(-terrain_width/2, 0, -terrain_height/2)

    # Unlock the image when done
    heightmap_image.unlock()
```