Procedurally Generated Space

Student: Saadi,Saadi

Supervisors: Boaz Sternfeld, Yaron Honen

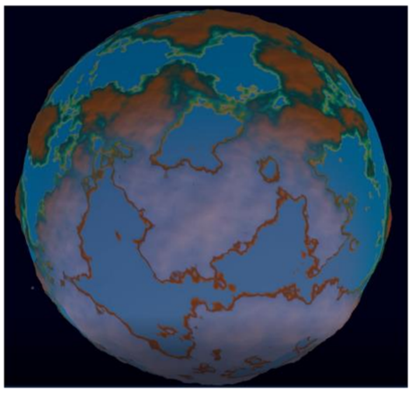
Completed in collaboration with the Center for Graphics and Geometric Computing as well as with the Geometric Image Processing Lab at the Technion – Israel Institute of Technology

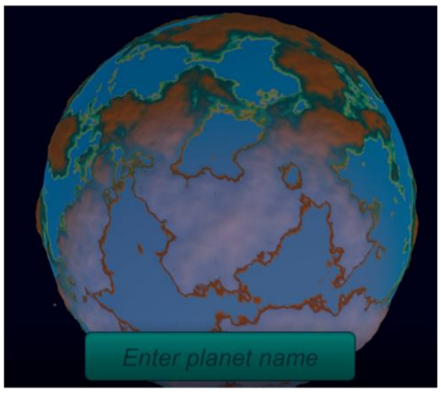
Introduction

This project was carried out in the CGGC/GIP lab at the Technion, under the supervision of Boaz Sternfeld and Yaron Honen. The objective of the project was to create planets using simplex noise, with a focus on developing a level-of-detail (LOD) system that allows for rendering small details while maintaining optimal performance.

Next, create a system that generates an endless space, with planets we made before serving as the building blocks.

Afterward, create a planet-naming system and a feature for locating lost planets within the vast expanse of space.





Features

1) You are in control of a spaceship that can move in all axes and rotate around all axes.
2) You have the ability to approach planets, allowing you to observe the terrain of the planet up close, as well as travel to other planets located at varying distances.
3) The planets are visually diverse with distinct colors and terrain features.
4) Planets have different temperature zones and regions.
5) The scene has a realistic ambiance due to the accompanying music and sound design.
6) A warning sound alerts you if you approach too closely to a planet, helping to avoid any collisions.
7) In the event of a collision with a planet, the spaceship's screen will break, but the system will recover after a certain amount of time.
8) An AI assists you throughout your journey, providing helpful guidance and updates.
9)You can display information on the screen that provides details about your location in space and the nearest planet in relation to your position.
10) When discovering a new planet, you have the ability to name it, and if lost, you can search for it by its designated name.
11) The game allows you to select the scene with the appropriate level of complexity based on your computer's performance, ensuring optimal gameplay experience.

Technologies and Platforms

1)Unity 2020.3.23f1 for app development
2)Inkscape for 2D art creation
3)SVG REPO for obtaining flat 2D art for the main menu
4)Audacity for sound manipulation
5)ElevenLabs for generating AI speech
6)freesound.org for sound effects
7)Kenny assets for additional sound effects

GENERATION PIPLINE

PLANETS

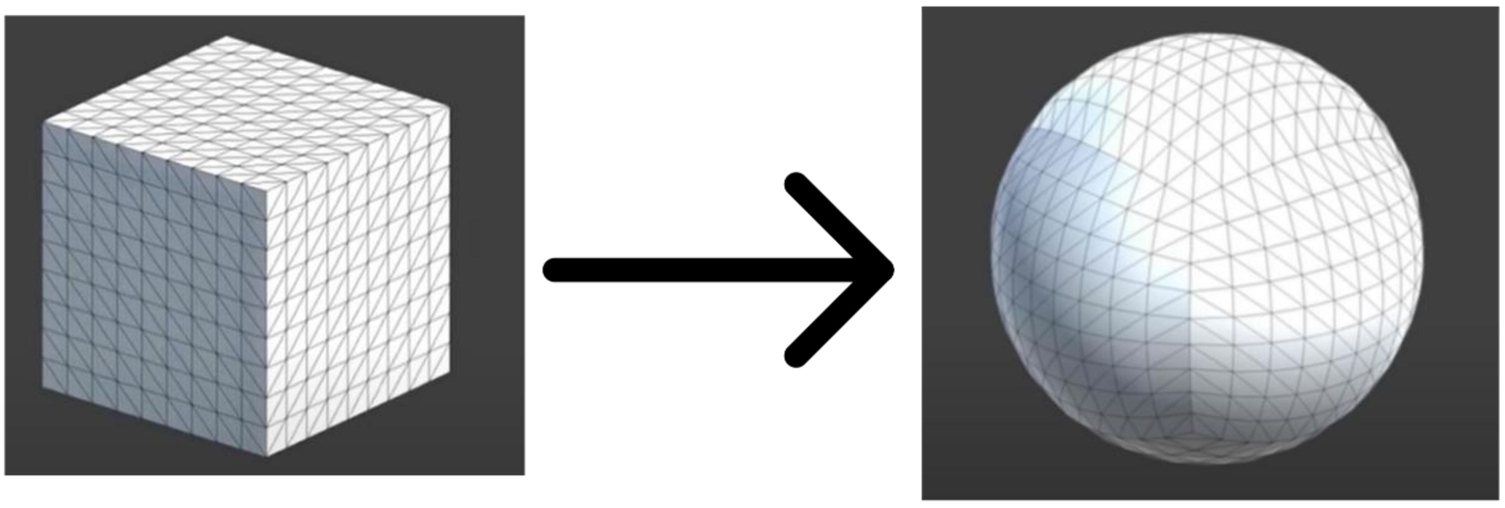
$$\begin{tikzcd}
\text{CUBE} \arrow[r] & \text{SPHERE} \arrow[r, "\text{noise}"] & \text{PLANET} \arrow[r] & \text{PLANET+=color} \arrow[d] \\
 & \text{PLANET+=CULLING} \arrow[d] & & \text{PLANET+=LOD} \arrow[ll] \\
 & \text{PLANET+=COLISSION DETECTION} & &
\end{tikzcd}$$

LOD

$$\begin{tikzcd}
\begin{array}{c}\text{ONE MESH}\\ \text{FOR EACH}\\ \text{FACE}\end{array} \arrow[r] & \begin{array}{c}\text{COUPLE OF MESHES}\\ \text{FOR EACH FACE}\\ \text{(ONE MATRIX)}\end{array} \arrow[r] & \begin{array}{c}\text{MESHES CAN}\\ \text{HAVE DIFFERENT}\\ \text{RESOLUTIONS}\end{array} \arrow[d] \\
\begin{array}{c}\text{WE DO REMESHING ALSO BEFORE}\\ \text{CHANGING MATRICES SMOOTHLY}\\ \text{BY DISTRIBUTING MESHES LINEARLY}\\ \text{OVER TIMLINE (REGULARLY JUST}\\ \text{WHEN CHANGING LEVEL OF DETAILS)}\end{array} & & \begin{array}{c}\text{COUPLE OF}\\ \text{MATRICES FOR}\\ \text{EACH FACE}\end{array} \arrow[ll]
\end{tikzcd}$$

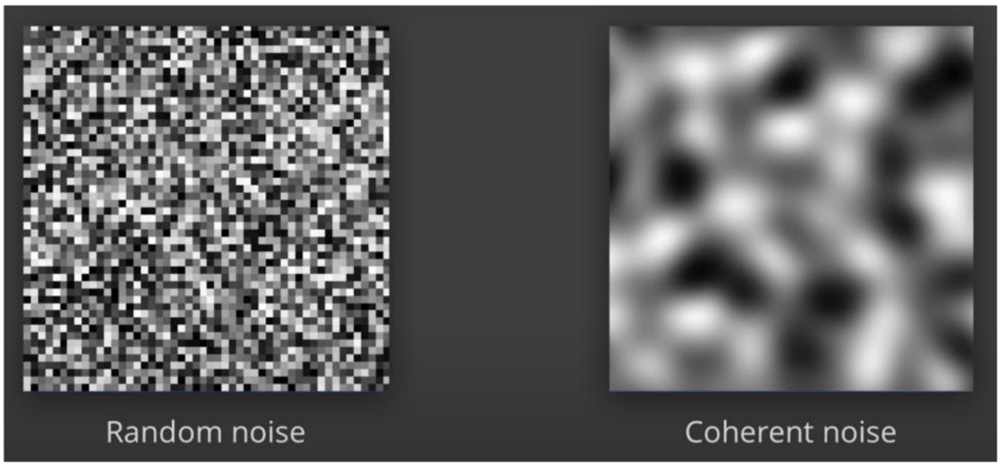
Planet Generation

We begin by defining a center point, and using this point as a reference, we construct 6 faces. For each face, we create a mesh with a specific resolution, and we arrange vertices in rows and columns based on this resolution.
Within each square formed by the vertices, we add 2 triangles. Each vertex is represented by a vector that originates from the center point. We normalize all of these vertex vectors in order to generate a sphere. This approach of using 6 faces and normalizing the vectors to transform a cube into a sphere will be used throughout the project. However, the number of meshes and the distribution of the vertices will vary, as we will discuss in the subsequent sections.

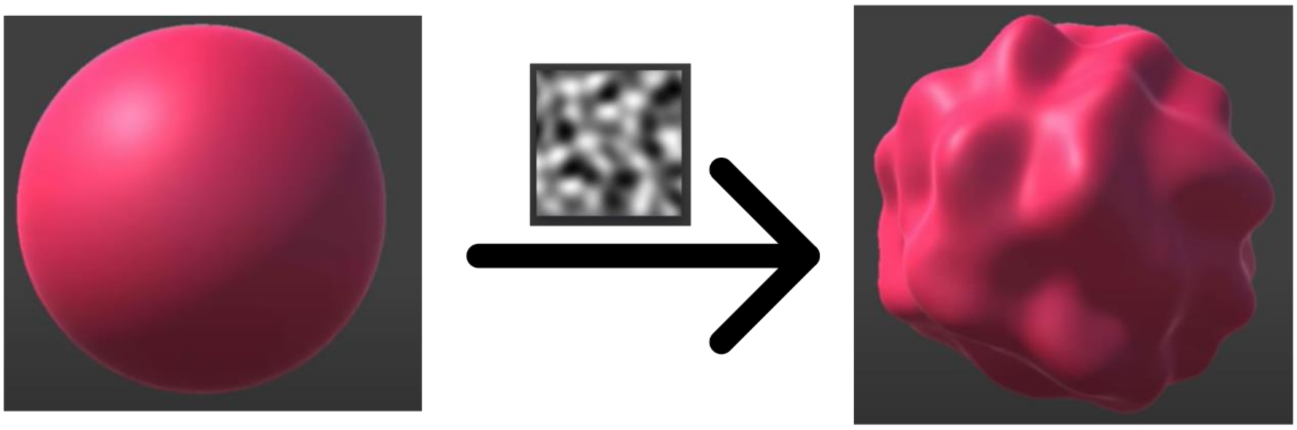


Simplex Noise

After constructing the sphere, we calculate noise using simplex-like noise. This noise takes a vector3 as input and outputs a random float between -1 and 1 based on the position of the vertex relative to the center of the planet. Simplex noise differs from regular random noise in that values change smoothly from point to point. We chose this noise because real terrain values also change smoothly between points.



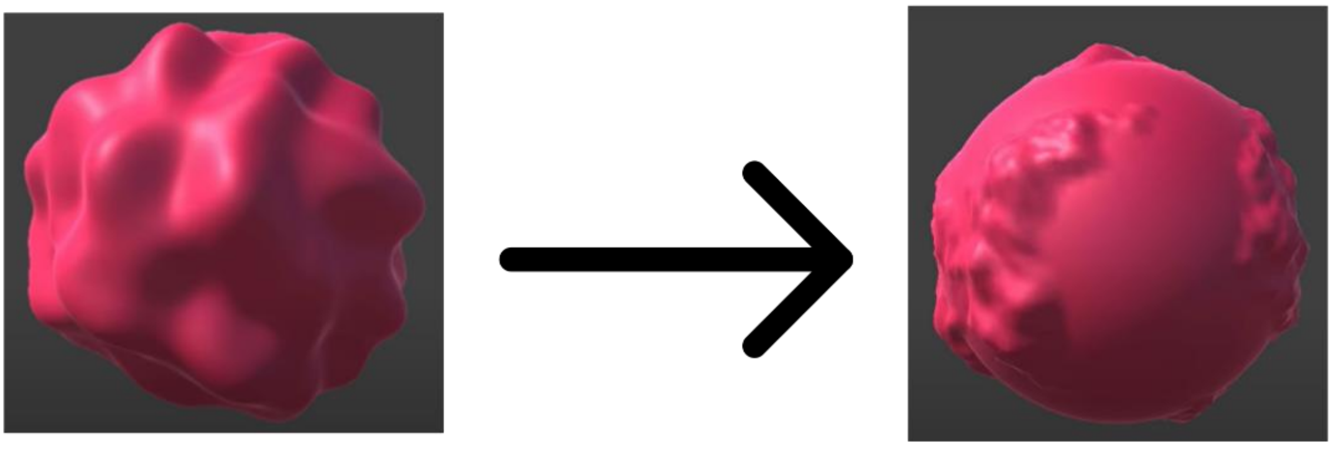
We remap the float value from [-1, 1] to [0, 1]. We then multiply the vertex vector by (1 + value) * radius, resulting in a transformed sphere that looks increasingly like a planet. However, there is still work to be done to complete the terrain.



Noise Controling

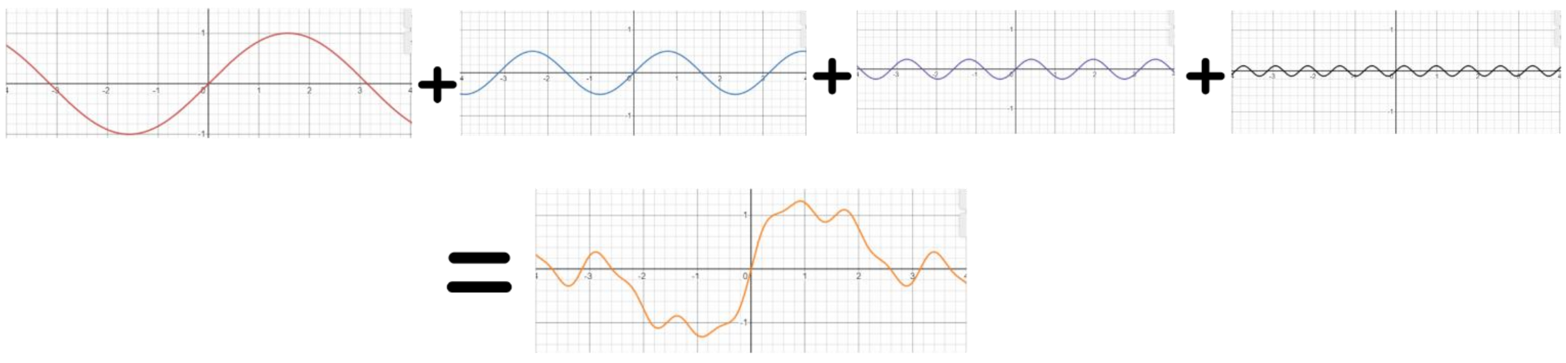
To gain greater control over the noise, we define a set of key parameters: "frequency," "amplitude," "reducer," and an animation curve named "curve." The noise is then calculated using the following equation:
Max(0, noise(curve(vertex * frequency)) * amplitude - reducer)
The names of the parameters themselves reveal their function; "frequency" and "amplitude" set the amplitude and frequency of the noise, while "curve" remaps the values of the vertex * frequency to create diverse planets.

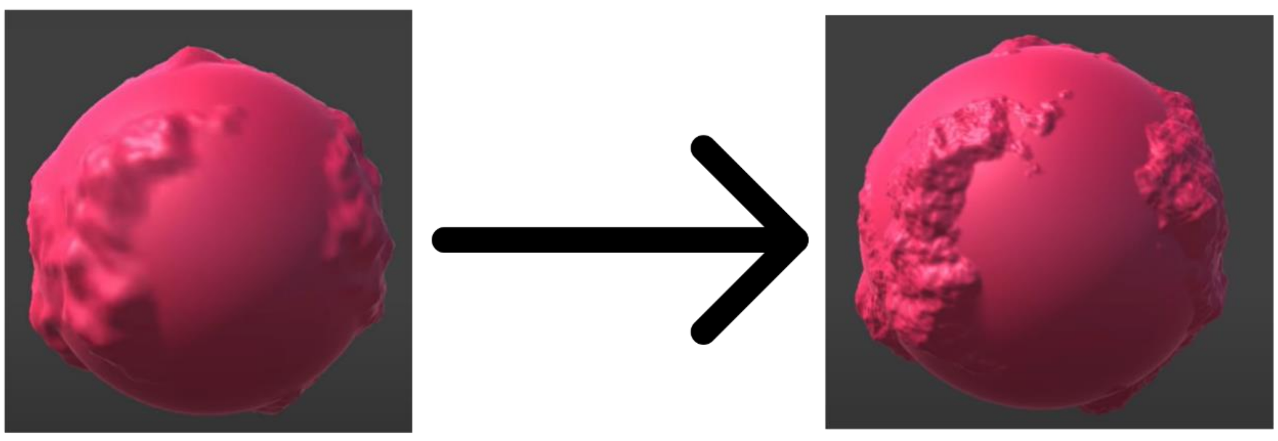
The "reducer" parameter is then used to reduce values to zero, resulting in oceanic features. This approach allows us to create unique, realistic planets with diverse features, lending to a more immersive experience.



Layered Noise

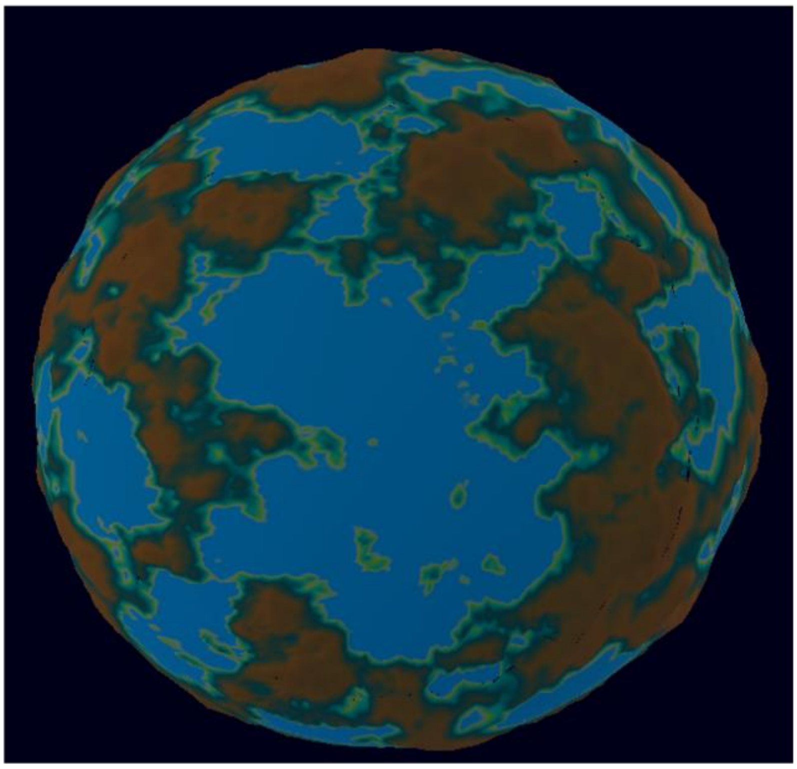
To achieve more detailed terrain, we can incorporate multiple layers of noise instead of just one, as we did previous page. Each layer has a higher frequency than the previous layer, while the amplitude is smaller.
After generating the noise for all the layers, we sum their outputs to obtain the final noise value.
 Although there are other techniques for calculating terrain height, they tend to have a negative impact on performance. Hence, I decided not to use them in the final version of the project.
It's worth noting that the higher the number of noise layers, the more performance-intensive the height calculation function becomes. This function is used across all the vertices, meshes, and faces of all the planets. Thus, to optimize performance, I chose to use only 4 layers in the project."

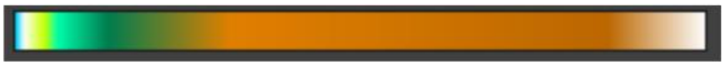




Color

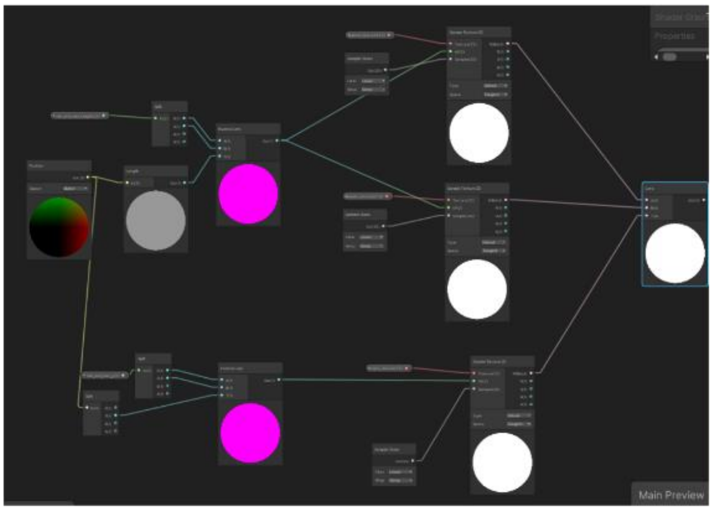
We have now come to a crucial point in our project - while we have successfully generated the planets we set out to create, they lack the diversity and realistic coloring of actual celestia bodies.
To address this, we turned to Unity's mesh coloring feature, which allows us to color meshes by coloring their vertices, with the triangles being colored by blending the colors of their three vertices. To implement this, I used Unity Shader Graph to create a shader that defines a color gradient.
The approach we took was to remap the height of each vertex to a corresponding color in the gradient, using the minimum and maximum heights of the planet as the range for the remapping.
The colors in the gradient are ordered based on height, ensuring that each color corresponds logically to its corresponding elevation. With this technique, we can now achieve diverse and realistic coloring for our planets.

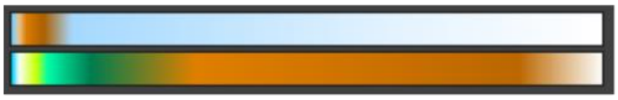


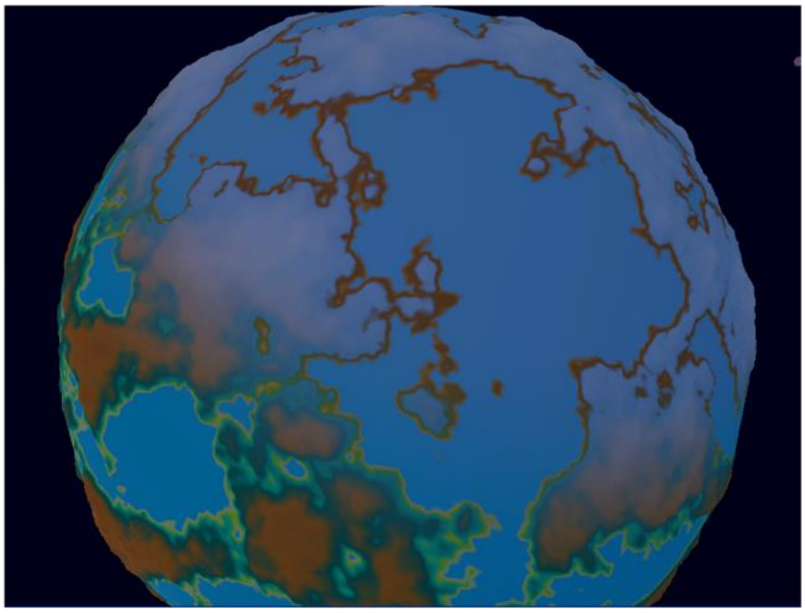


Temperatures

now we have colored the planets but they still boring,
to make the planets look more realistic, we need to give different regions different temperatures, to achieve that cold and warm regions should have different colors.
so I came with this implementation where we have 2 color gradients one for the cold and one for the warm areas what I do is taking the height of the vertex and take the Y value of the vertex and as we did in the previous page I get the color according to the height from both gradients, then make a noise layer like the one in the section of calculating the height , and we feed the Y this time to the noise and we remap the output to acolor between the two colors we got from the gradients.

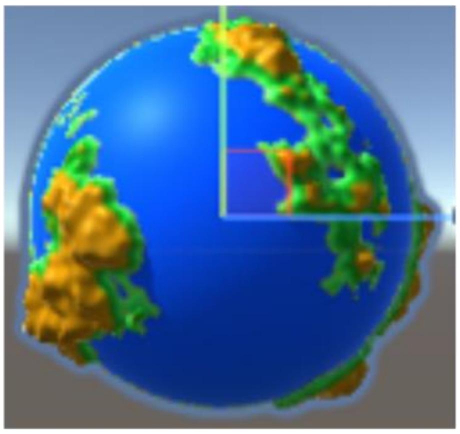




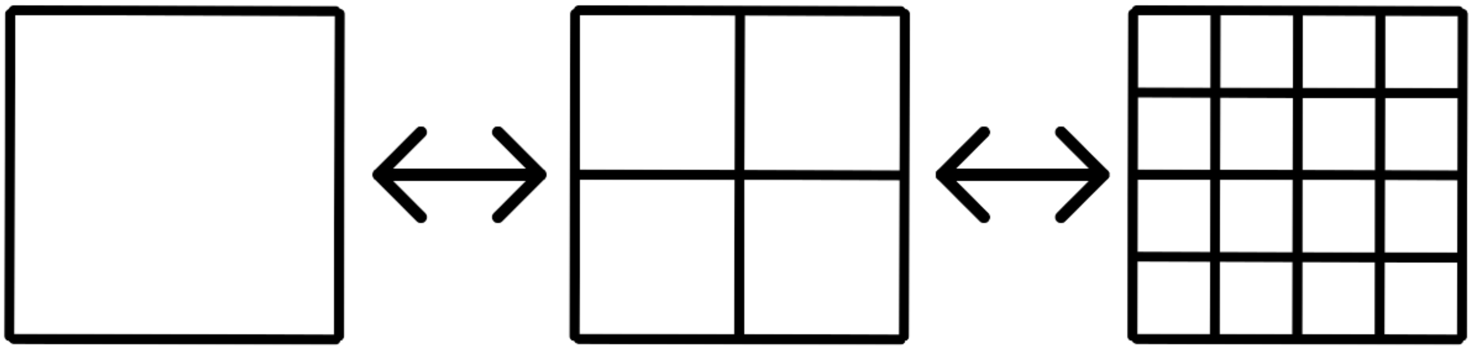


LOD

Now that we have shaped and colored our planets, we encounter a problem with proportion.
Mountains appear to be as large as continents due to the single mesh with a single resolution, this results in having a few large terrains instead of many smaller ones, and planets look too small like King Kai's Planet in Dragon Ball.



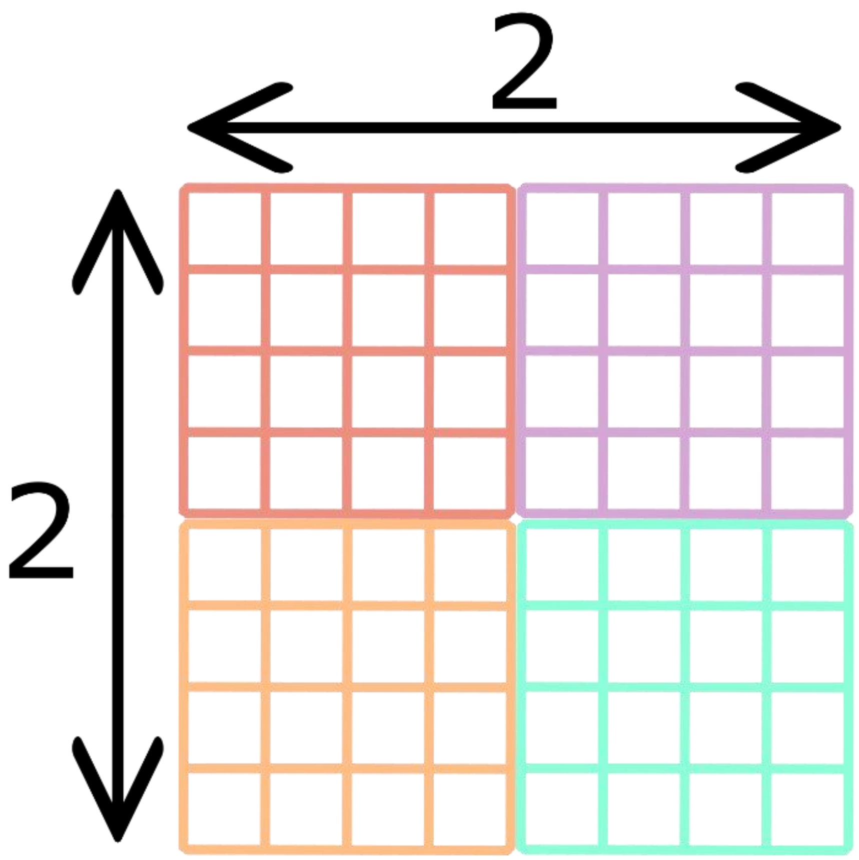
To address this issue, we begin implementing a level of detail (LOD) system.
First, I calculate the distance between the camera and the planet's center and determine the resolution based on the distance. The closer the camera is to the planet's center, the higher the resolution will be.
I use distances-resolutions lookup table.
Based on the current distance, I check the distances int the table and select the corresponding resolution from the resolutions in the table.

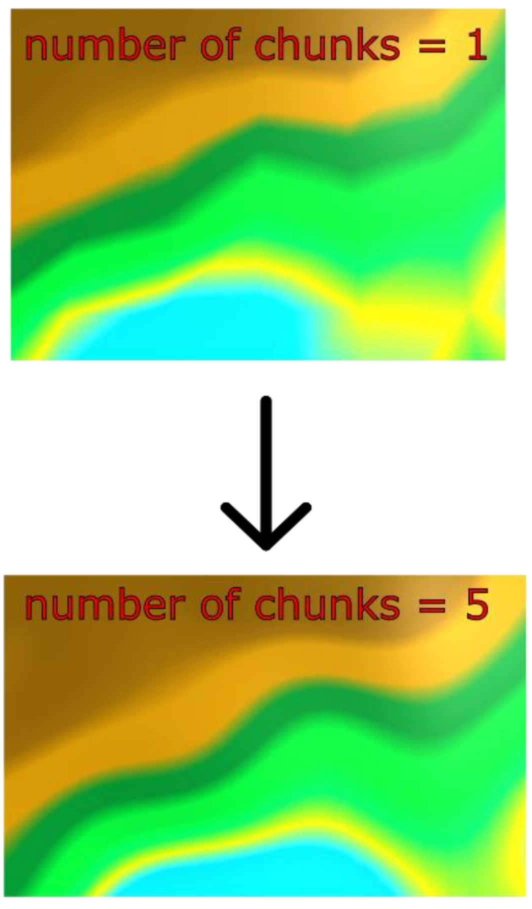


Multiple Meshes In Each Face

It appears that simply changing the mesh's resolution is not sufficient for our needs, as Unity only allows up to 256 * 256 vertices, which is too low for our purposes. To address this, we can use multiple meshes for each face instead of just one. We introduce a variable called "number of chunks" and create a matrix of meshes with dimensions equal to that variable. The illustration below shows an example of how a face would look when the variable is set to 2.

With this approach, the planets appear more detailed, larger, and more realistic. However, there is a performance issue with this method. When the "number of chunks" is increased, the app's performance rapidly decreases. For example, when the number is set to 4, the application experiences high frame drops and is almost unusable.

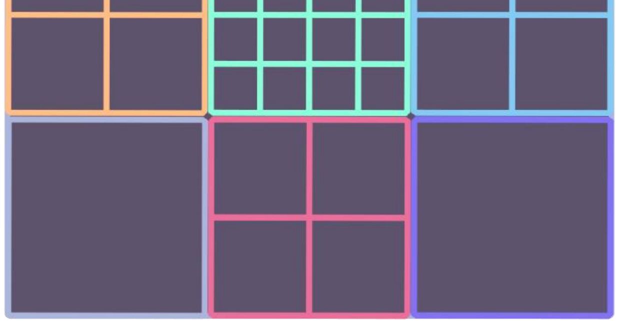


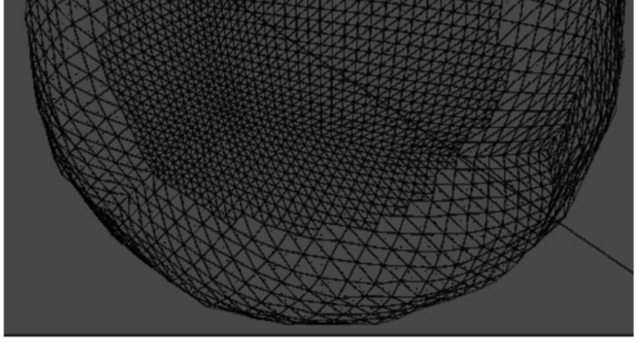


Different Resolutions

To maximize the benefits of changing resolutions for each mesh , we need to adjust the resolution of each chunk independently. Instead of basing our lookup table on the distance from the camera to the center of the planet and setting the resolution for all meshes across all faces, we need to base our lookup table on the distance from the camera to the center of each mesh. By doing so, we can increase the number of chunks to 25 without any issues.

Initially, I was hesitant to take this approach due to the gaps between meshes and lighting problems that typically arise.
I spent nearly a month searching for alternative solutions, and after not finding anything , I started thinking about leaving this project, but ultimately decided to give this approach one more chance.
In upcoming sections, I'll address the gaps between meshes and lighting issues in more detail.





Automatic velocity Change

Here's a possible rephrased version of the text:

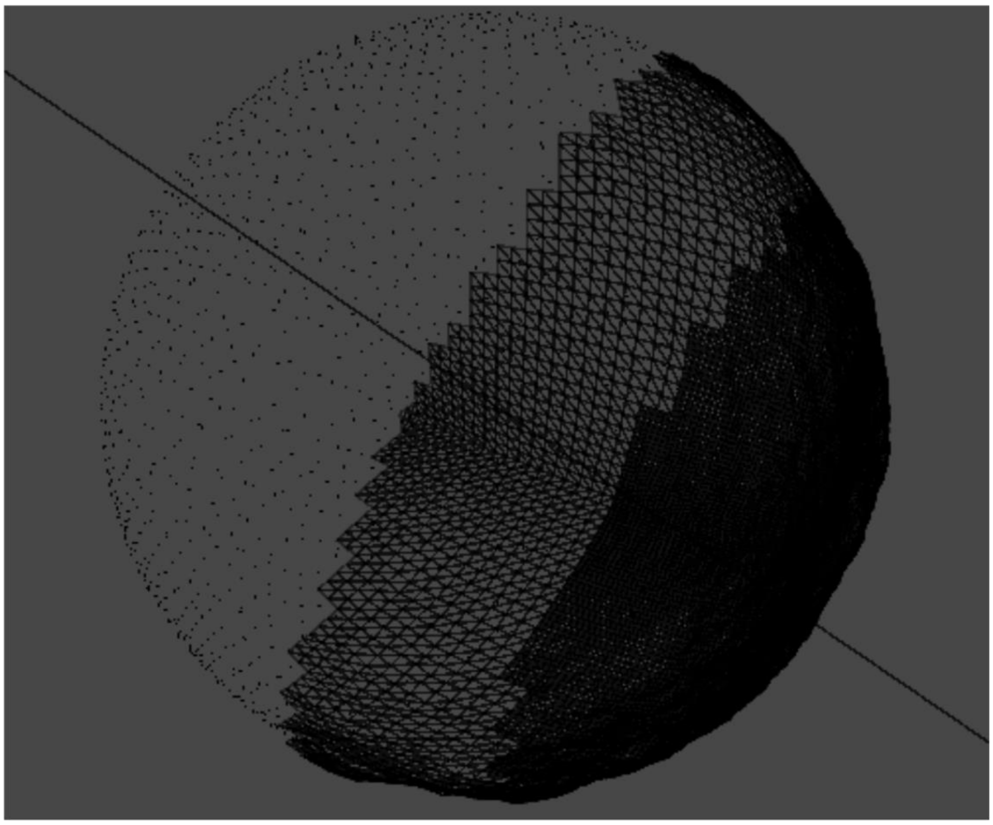
In order to account for the different velocities of various modes of transportation, I developed a system that adjusts the speed of the spaceship based on its proximity to the nearest planet. The closer the spaceship is to the planet, the slower its speed becomes.
Conversely, when the spaceship is in empty space, it can move at a much faster speed.

This system serves a dual purpose. Firstly, when the spaceship approaches a planet, the resolutions of the meshes of the planets increase significantly, resulting in a high number of calculations.
To avoid performance issues such as frame drops and lag, it is necessary to move the spaceship at a slower pace in such cases.
Secondly, by adjusting the speed of the spaceship based on its distance from the planet, we ensure that the spaceship does not collide with the planet or any other obstacles, as the speed is appropriately reduced when the spaceship is in close proximity to any objects.

Back Face Culling

To optimize performance even more, we can add another technique to our LOD system called backface culling. Backface culling is a method where you cull the faces that the camera cannot see from the back and sides so that the GPU doesn't render them.

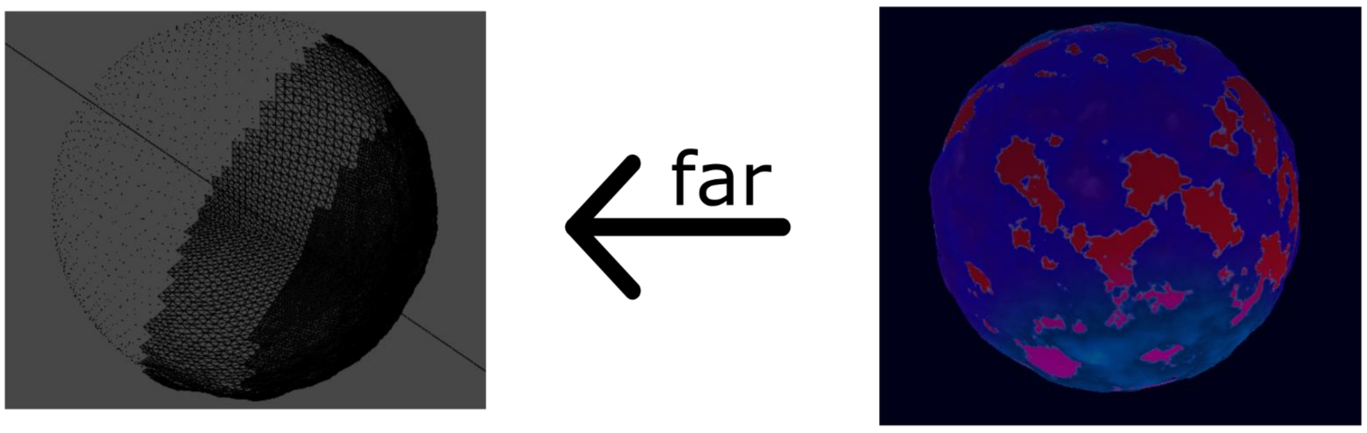
In my implementation, I cull the meshes that the camera can't see by calculating the spaceship-planet vector and the mesh-planet vector and checking the angle between them. If the angle is greater than a certain number, for example, 90 or 100 degrees, we cull them by disabling the mesh renderer component. As they are culled, we don't change the level of detail of that chunk until they are shown again. Then we check the current LOD that should be shown and the LOD before the culling. If they are different, we remesh. This approach works well performance-wise because the angles change smoothly.

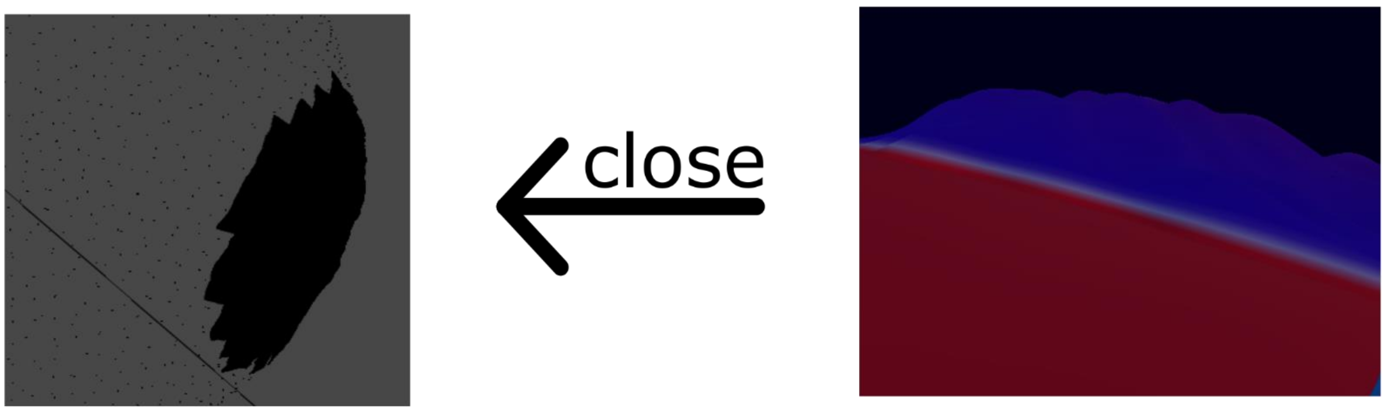


Advanced Culling

I observed that when the spaceship is near the planet, fewer meshes are visible within the camera's frustum. As a result, we can optimize performance by culling more meshes when the spaceship is close to the planet. This is particularly beneficial because the resolutions of the meshes are higher when viewed up close, requiring more calculations.

To achieve this, I add the angles to the lookup table.
We set a variable called "min lod" to represent the minimum level of detail for all meshes on the planet. Using this variable, we select an angle from the lookup table to cull meshes based on their distance from the camera. When the "min lod" value is smaller (i.e., when we're closer to the planet), we choose a smaller angle from the lookup table and cull more meshes accordingly.





Space Generation

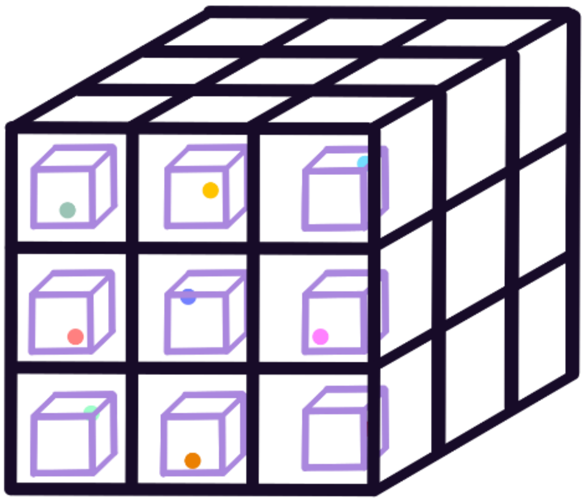
So far we have discussed the process of generating one planet, but now we need to consider how to build the entire space while maintaining optimal performance. There are a few key considerations we need to keep in mind to avoid breaking the game:

1) Planets should be placed far enough from each other to avoid too many calculations for the LOD system of both simultaneously.

2) Distant planets should have fewer vertices to ensure that rendering is not too expensive for one planet only.

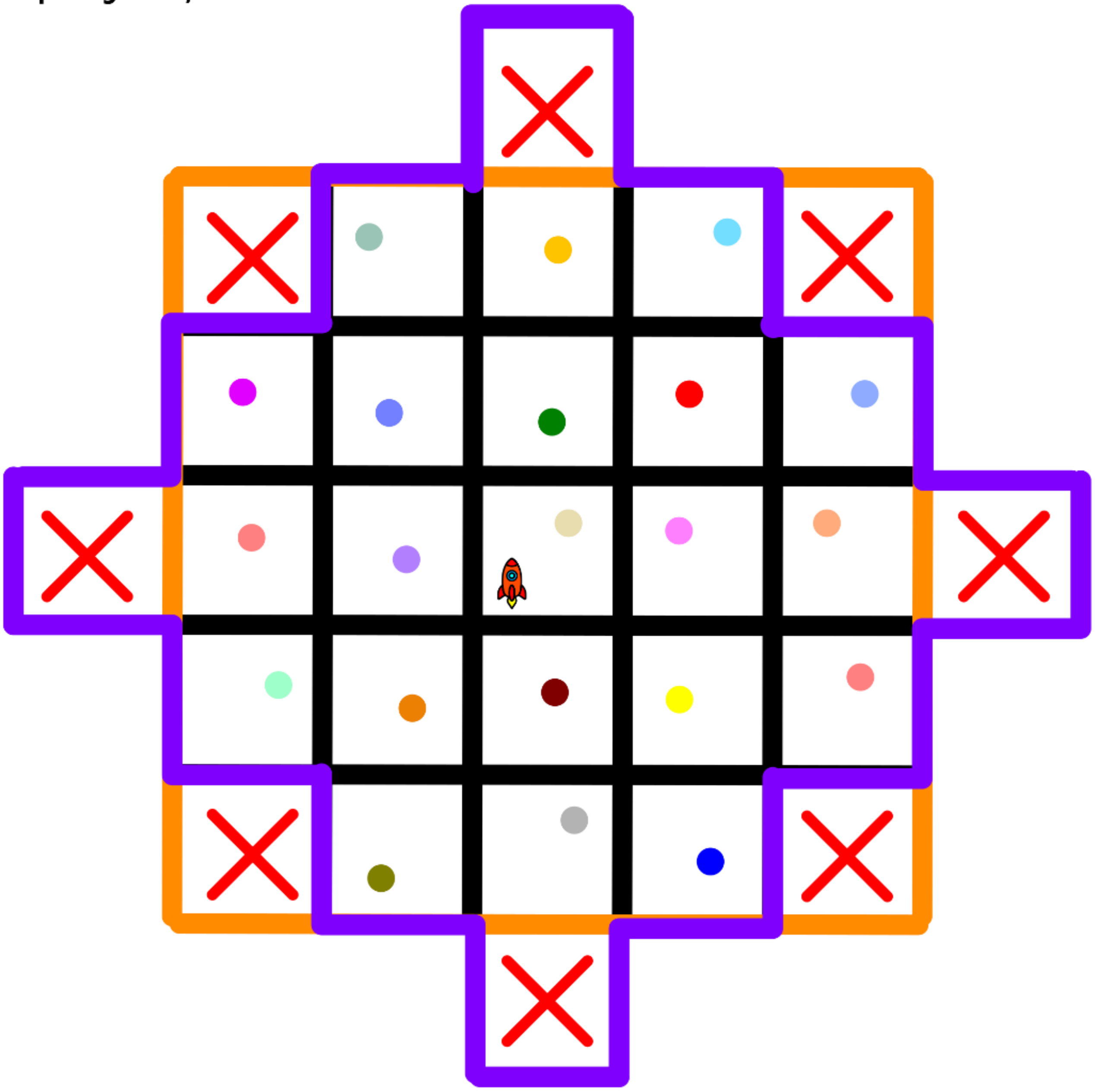
3) We cannot generate an infinite number of planets from the beginning of the scene, so we need an optimized way to dynamically generate planets.

To accomplish this, I divided the space into boxes, with each box containing an inner box. We randomly generate a planet inside each inner box, that ensures planets are far enough from each other to maintain optimal performance.



Planets To Show

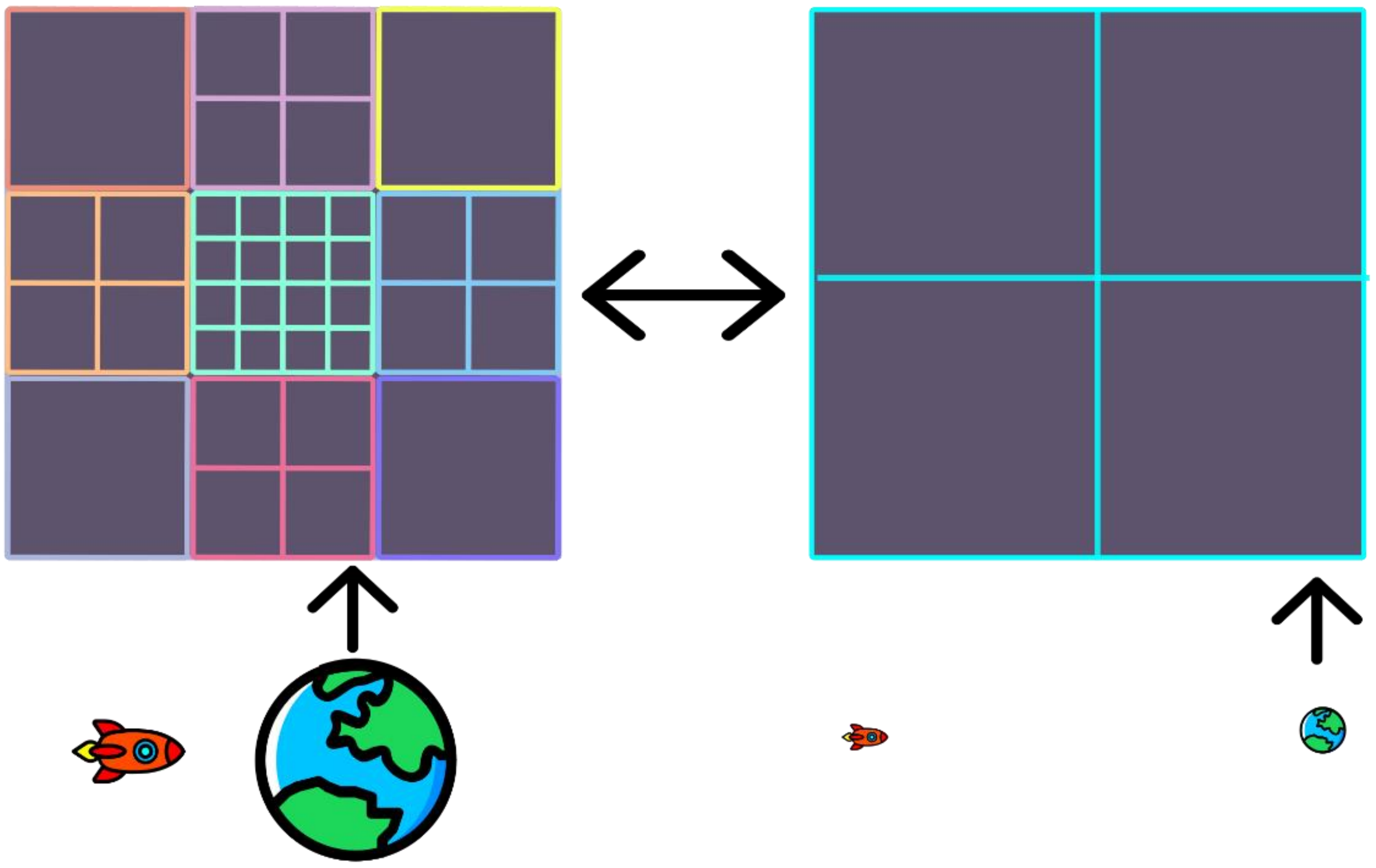
As I mentioned earlier, it's not feasible to show all the planets in all the boxes. To address this, we declare two variables: "max L1 distance" and "max L-infinity distance".
We then show the planets of the boxes that are within L1 and L-infinity distances from the current box (the box that the spaceship is currently inside).
In the example below, "max L1 distance" is set to 3 and "max L-infinity distance" is set to 2. Please note that this example is presented in 2D for better illustration, but in the actual project, we work with 3D.



Multiple Mesh Matrices

As we have seen in the previous example, we may end up with a large number of planets rendered on the screen.
Currently, we have a matrix of chunks for each face, where the "number of chunks" is set to 20. This means that for each face, there are 400 meshes.
Even if far planets have the lowest possible resolution, they will still require a significant number of vertices and triangles that are not necessary, which worsens performance and puts unnecessary strain on the GPU.

To address this issue, I added a new feature to the LOD system in which each face can have multiple matrices. When we get too far from the planet, we can disable all the meshes of the first matrix, which has 20 x 20 = 400 meshes, and enable the second matrix, which has a smaller number of meshes, for example, 2 x 2 = 4. This way, far planets don't take up much computational resources. We can go between the matrices according to how far we are from the planet."



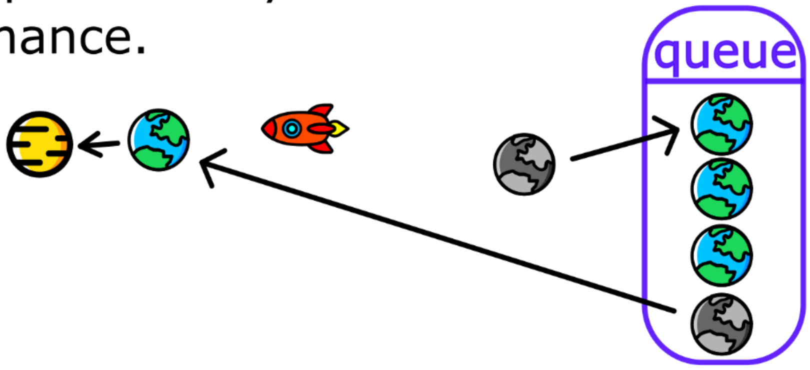
New Planets

In the previous section, we discussed how planets are shown based on which box the spaceship is currently in. However, what happens when the spaceship moves to a neighboring box to reach another planet? In this case, new planets must be shown while others need to be removed. The naive solution might be to instantiate new planets and destroy the ones that are no longer needed. However, this approach is computationally heavy and can break the game.

Instead, we can use a more efficient method called object pooling, which we will discuss in the next chapter.

Object Pooling

Object pooling is a programming technique that involves reusing and managing a pool of pre-allocated objects instead of constantly creating and destroying them during runtime.
This can help improve performance and reduce memory usage in applications that frequently create and destroy objects.
in our project we need to use object pooling for creating and destroying planets.
Classic object pooling is not a viable solution for our project, as the planets are essentially the same object with minor parameter differences, but with over 400 unique meshes that differ across all planets.
When we retrieve a planet from the pool, we would need to modify and remesh its meshes according to the new parameters, which is computationally intensive and would negatively impact performance.



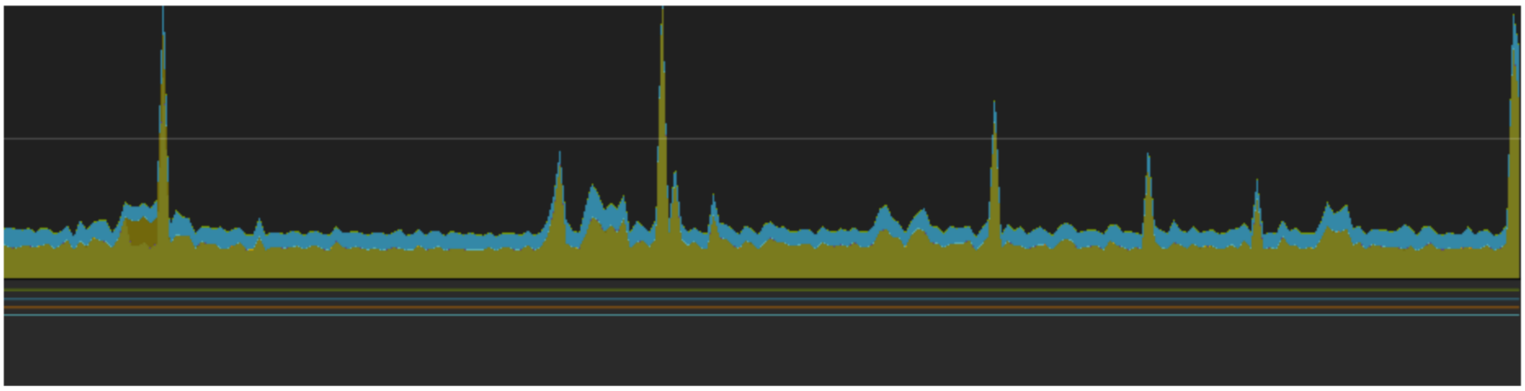
Not modifying the meshes can lead to a bug where the planet looks different when transitioning from one matrix to another, but this can be resolved when the meshes change LOD.
Since we have 2 matrices, the first with 20 x 20 meshes and the second with 4 x 4 meshes, we face a problem with the last LOD of both matrices.
However, the second matrix is only used when the planet is very far away, so the camera does not notice the difference in the planet's appearance.
The real challenge is transitioning from (matrix = 1, LOD = 0) to (matrix = 2, LOD = last LOD), where we need to remesh all the planet's meshes intelligently.
We will explore this further in the next chapter.

Updating Matrices Smoothly

How to efficiently remesh all the 20 x 20 meshes and when?
We should begin remeshing the "min LOD" becomes 0, we should end remeshing when we reach the distance where we need to switch matrices.
To ensure a smooth transition, we need to
divide the meshes linearly over the timeline and remesh them in small amounts each frame.
This was a significant challenge in the project, and I had to experiment with various dividing functions.
Initially, I tried remapping the distances from the chunks to the timeline, but that didn't work well due to the speed of the spaceship, and the distances were not changing linearly.
Ultimately, I found a solution to remap the meshes based on a percentage calculation of the rows and columns to distribute the meshes linearly over the timeline:
(row + column * 20)/400.
we can optimize by remeshing only the meshes not culled due to back face culling caused bugs, so I left that aside and focused on updating the matrix meshes smoothly.
During this process, I learned to work with the Unity Profiler to address the breaks and lags I was encountering.

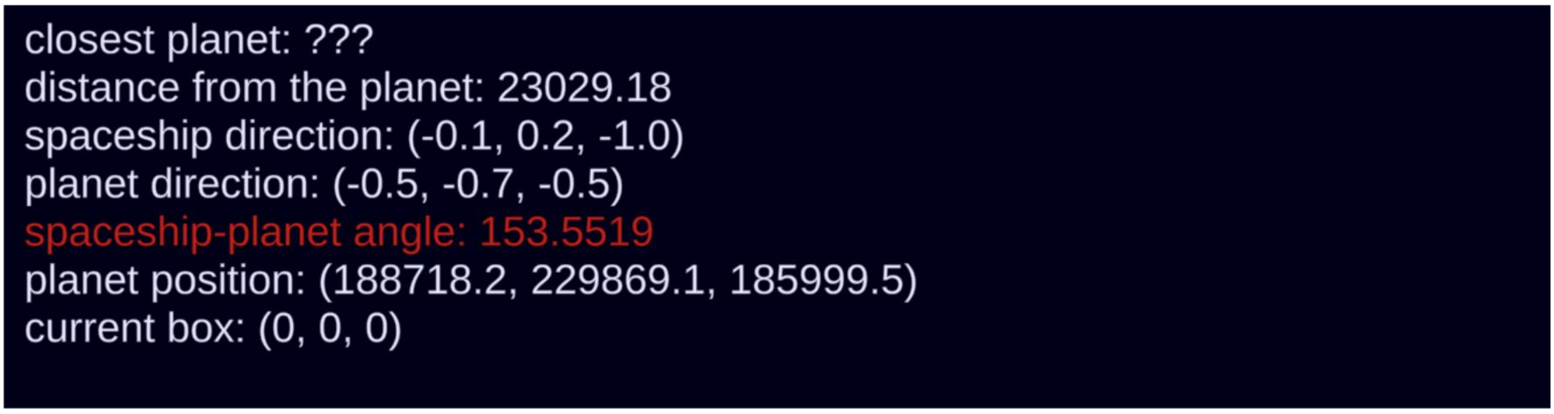
Working With The Profiler

to know where the frame drops and the lags are coming from
I had to learn to work with the profiler API , write profiler code
so I can group parts of the code and make their perfomance
show in the profiler windows , I grouped 3 main parts.
the smooting code , the defualt remeshing code, and the
default enabling mesh renderers code, then I started to try
understand where are the problems and what is happening
I understanded part of the problems , and others still don't
understand untill now but I think big part of them is because
of my computer is a bad one and doesn't have a real GPU
I will took about that in the later sections



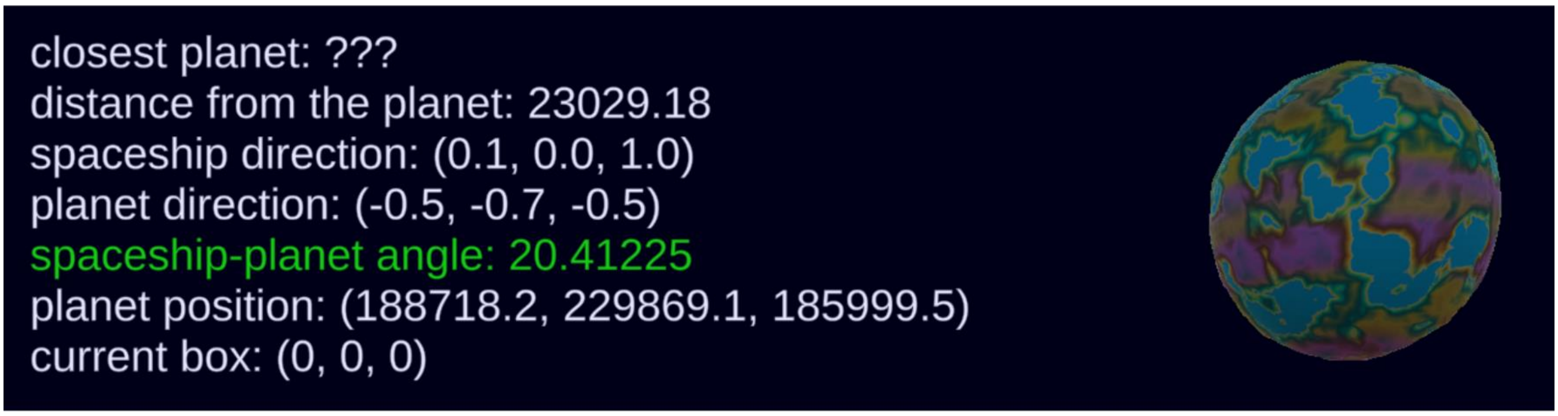
Information Screen

The project has advanced significantly, and now the spaceship can travel between well-crafted planets in the vast space. However, a problem arises: there is no navigation or naming system, which makes it difficult for the captain to distinguish between two planets with similar shapes and colors.
To address this issue, I created an information screen that displays relevant information about the planet being observed, as shown in the screenshot.



```
closest planet: ???
distance from the planet: 23029.18
spaceship direction: (-0.1, 0.2, -1.0)
planet direction: (-0.5, -0.7, -0.5)
spaceship-planet angle: 153.5519
planet position: (188718.2, 229869.1, 185999.5)
current box: (0, 0, 0)
```

If the spaceship is not currently pointed at the nearest planet, it can use the spaceship-planet angle to try and minimize it.

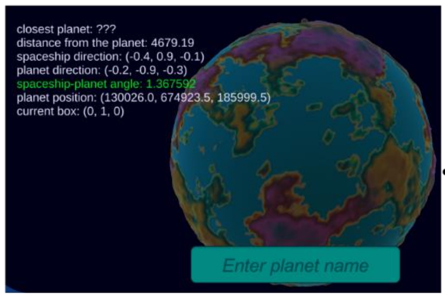


```
closest planet: ???
distance from the planet: 23029.18
spaceship direction: (0.1, 0.0, 1.0)
planet direction: (-0.5, -0.7, -0.5)
spaceship-planet angle: 20.41225
planet position: (188718.2, 229869.1, 185999.5)
current box: (0, 0, 0)
```

In the screenshot, the planet at the top of the screen is marked as "???" because it is unknown. Our next objective is to develop a system that enables the planets to be named.

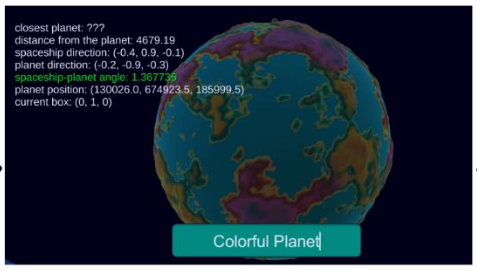
Naming Planets

At first, I considered using a planet-names database from the internet to randomly assign names to each planet. However, I discovered that the names in such databases are often code-like due to the sheer number of planets discovered, which is not particularly engaging for the player and doesn't assist with navigation.
Instead, I came up with a more enjoyable solution which involves allowing the player to name the planets themselves. This not only adds an element of fun, but also helps the player to better remember and distinguish between the planets.



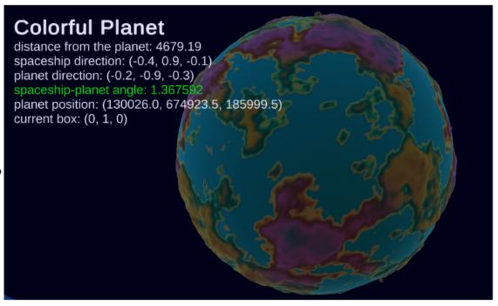
```
closest planet: ???
distance from the planet: 4679.19
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.2, -0.9, -0.3)
spaceship-planet angle: 1.367592
planet position: (130026.0, 674923.5, 185999.5)
current box: (0, 1, 0)
```

$\rightarrow$



```
closest planet: ???
distance from the planet: 4679.19
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.2, -0.9, -0.3)
spaceship-planet angle: 1.367735
planet position: (130026.0, 674923.5, 185999.5)
current box: (0, 1, 0)
```

$\rightarrow$

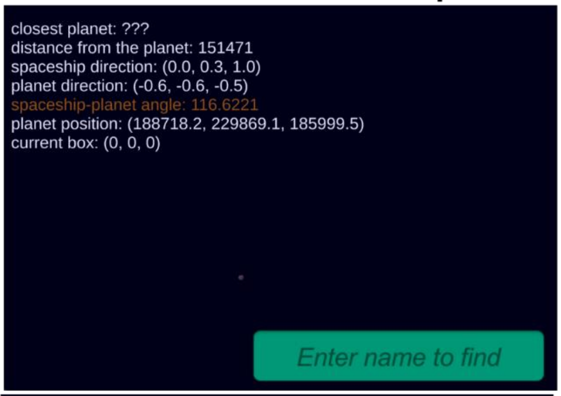


```
Colorful Planet
distance from the planet: 4679.19
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.2, -0.9, -0.3)
spaceship-planet angle: 1.367592
planet position: (130026.0, 674923.5, 185999.5)
current box: (0, 1, 0)
```

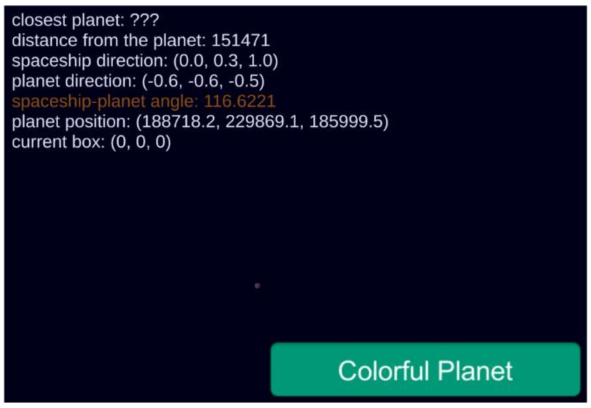
The naming system is designed to be intelligent and incorporates various rules to ensure proper naming conventions.
For example, it prevents spaces from being used at the beginning or end of a name, disallows two spaces in a row, prohibits numbers from being used in the middle of a word, and requires each word to start with a capital letter.
Additionally, the system dynamically modifies the name as the player adds or removes letters, providing immediate feedback to the player.

Finding Planets

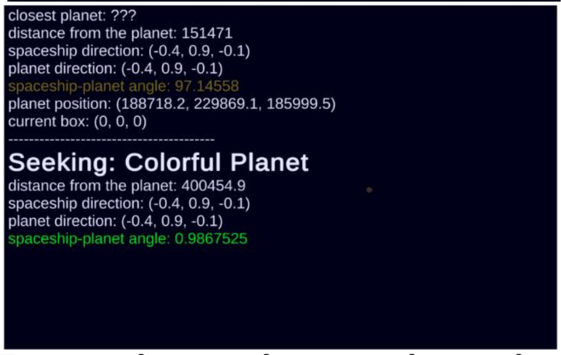
Having implemented the planet-naming feature, players can now recognize the planets they've previously discovered using the information screen.
However, another problem arises: what if a player wants to revisit a specific planet, but is too far away and doesn't know which direction to go in?
To address this, I added an input text field that allows the player to type in the name of the planet they're searching for. Once the name is entered, the spaceship is automatically rotated towards the direction the player needs to travel to reach the desired planet.



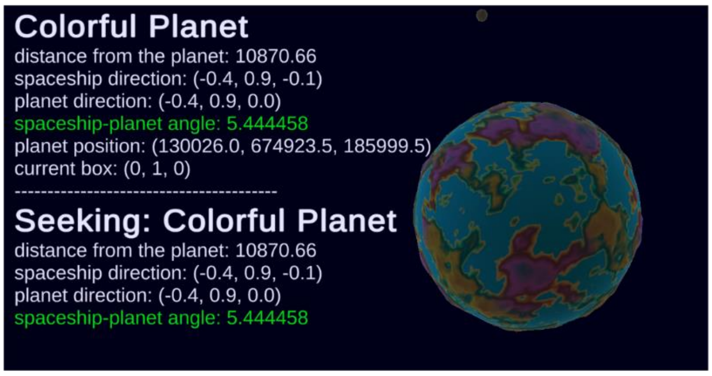
```
closest planet: ???
distance from the planet: 151471
spaceship direction: (0.0, 0.3, 1.0)
planet direction: (-0.6, -0.6, -0.5)
spaceship-planet angle: 116.6221
planet position: (188718.2, 229869.1, 185999.5)
current box: (0, 0, 0)
```



```
closest planet: ???
distance from the planet: 151471
spaceship direction: (0.0, 0.3, 1.0)
planet direction: (-0.6, -0.6, -0.5)
spaceship-planet angle: 116.6221
planet position: (188718.2, 229869.1, 185999.5)
current box: (0, 0, 0)
```



```
closest planet: ???
distance from the planet: 151471
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.4, 0.9, -0.1)
spaceship-planet angle: 97.14558
planet position: (188718.2, 229869.1, 185999.5)
current box: (0, 0, 0)
-------------------------------------
Seeking: Colorful Planet
distance from the planet: 400454.9
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.4, 0.9, -0.1)
spaceship-planet angle: 0.9867525
```



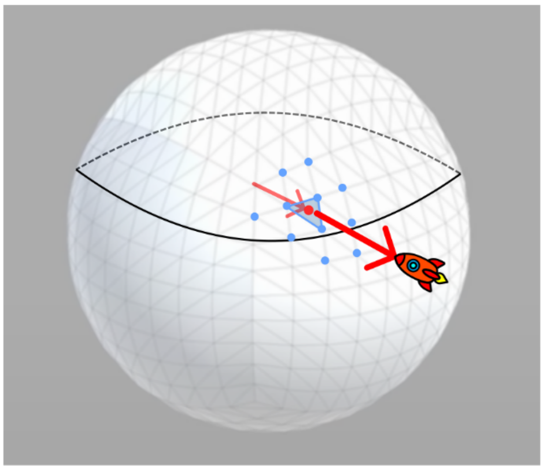
```
Colorful Planet
distance from the planet: 10870.66
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.4, 0.9, 0.0)
spaceship-planet angle: 5.444458
planet position: (130026.0, 674923.5, 185999.5)
current box: (0, 1, 0)
-------------------------------------
Seeking: Colorful Planet
distance from the planet: 10870.66
spaceship direction: (-0.4, 0.9, -0.1)
planet direction: (-0.4, 0.9, 0.0)
spaceship-planet angle: 5.444458
```

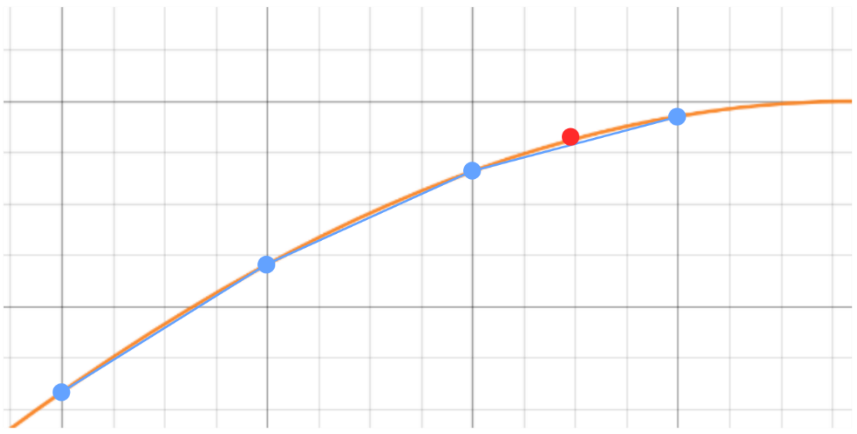
But what about the planets that are not currently in the scene due to object pooling?
How can I know their position to calculate the direction and make the spaceship look at them?
The solution to this problem is that I have created a dictionary, where each planet's name is used as a key and a corresponding box is used as the value. Since I use the box as a seed when calculating the position of the planet, as well as all other attributes such as noise and colors, I can restore the position of the planet without it being present in the scene.

Collision

There's one remaining interaction that we need to address: what if the player intends to collide with a planet? As we currently lack a collider for the planets, the spaceship would pass through them, which is problematic. We need to devise a way to prevent this from happening.
Adding colliders for the meshes seems like a simple solution, but even if we only add colliders for meshes with LOD = 0, the collider would still cause lag and poor performance. Furthermore, Unity cannot add a collider for meshes with a certain number of triangles.
Unity's code is optimized, so attempting to create colliders from scratch would be a waste of time without yielding any results.
This problem put a great deal of pressure on me because without a way to determine whether there is a collision or not, the only solution would be to calculate whether the spaceship is outside a sphere that encloses the planet, which would remove the fun of exploring the planet's terrain, and If the supervisors reject this solution, I might have to abandon the project.
The only path forward was to consider the limitations of my project that didn't require a collider and attempt to find a solution based on them.
We know that the vertices of the planet meshes are based on simplex noise that changes smoothly, and their positions are calculated based on the noise and their position on the sphere that we got from the cube with 6 faces we started with.
After a lengthy search, I succeeded to find a solution that checks for collisions in O(1) time.

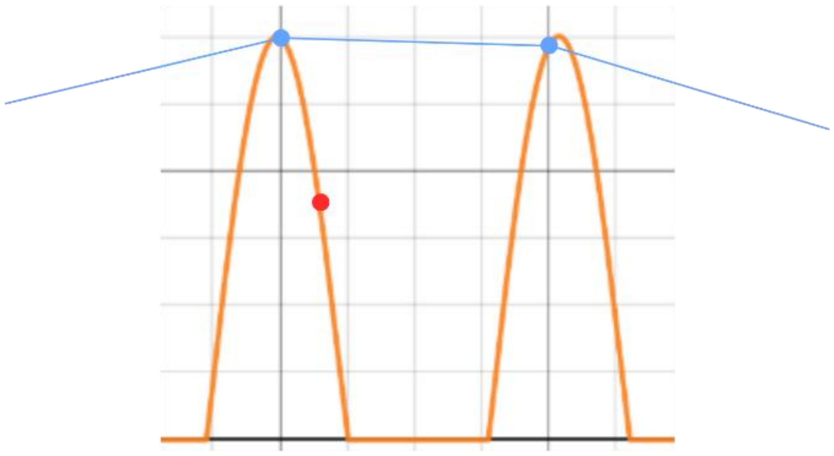
I obtained the normalized vector from the center of the planet to the spaceship, which is now on the small sphere mentioned earlier.
Then, I pretended that this vector was one of the planet's vertices, even though the probability of that is low.
Using the same method as we do for the real vertices of the mesh, I calculated the height of that vector.
I then checked the distance between the spaceship and the planet.
If it was lower than the height, then a collision had occurred; otherwise, it had not. However, this method is not entirely accurate, although it is adequate for our needs. To explain why, let me provide a 2D example that you can extrapolate to the project's 3D setting.



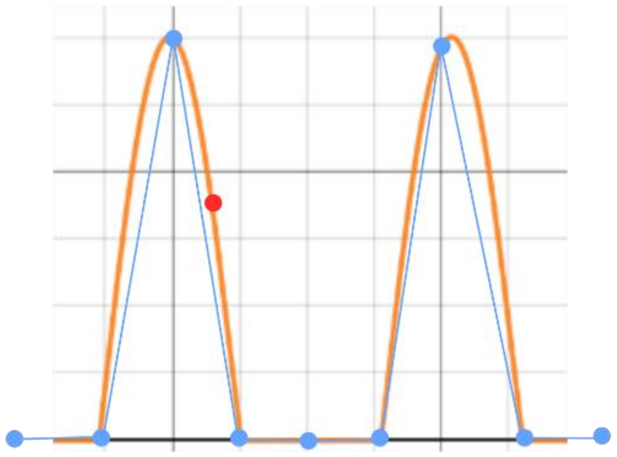


As shown in the diagram, the height of vector x can be anywhere, but we know that there are smooth transitions, so the vector times height is close to the line between the two adjacent points. Several factors affect the height's accuracy, such as choosing a small frequency and amplitude and a large resolution.
In my project, I opted for these settings to achieve greater precision. However, there is still some inaccuracy, so I check whether the distance is less than the height plus a "collision distance" to prevent the player from entering the planet.

Here is an example of how this solution may become useless when using a small resolution with high amplitude and frequency.



By increasing the resolution, we can observe a significant difference in the results.
In contrast to the previous example, the solution works effectively without any issues.

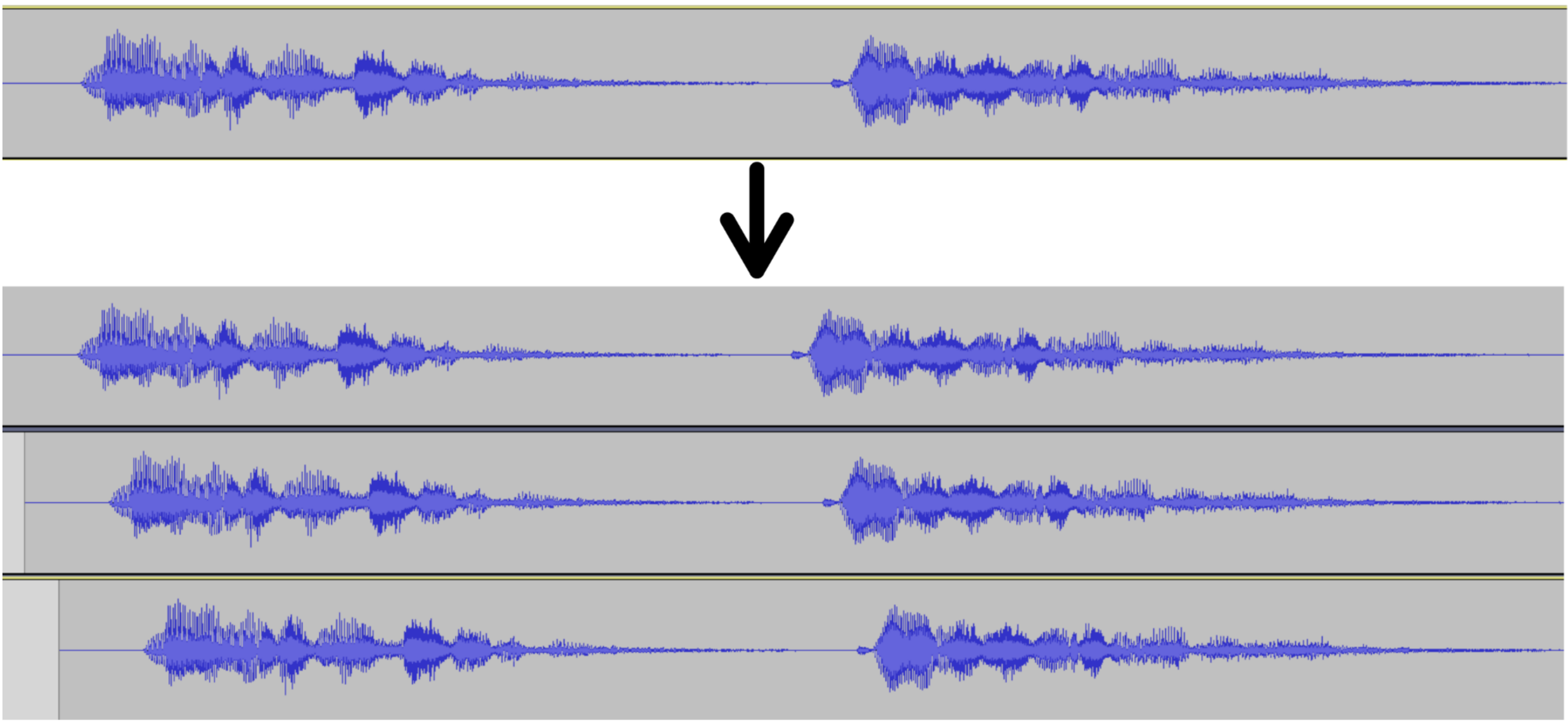


UX and game design

To enhance the player's experience, I implemented several features to assist them.
Firstly, an AI system was developed to provide the player with information that they could easily understand and use to progress further.
Additionally, I color-coded the text on the information screen to indicate important details, such as changing the distance from the closest planet to red when the player is too close, indicating potential danger.
Furthermore, I remapped the angle between the spaceship and planet from 0-180 to green-red and color-coded the text to help the player understand that smaller angles are better for finding the planet.
To further aid the player, I manipulated the sound volume and pitch.
For instance, when the player is close to a planet, a warning sound is played, and the pitch of the sound increases as they get closer to indicate greater danger.
Lastly, since the velocity change is automatic and takes time to move away from the planet, I increased the sound of the engine to help the player feel the velocity change.

AI speech making

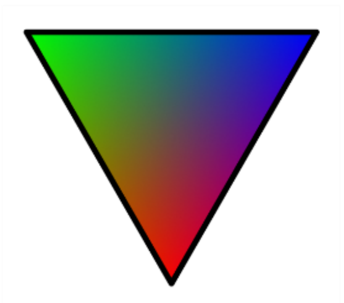
To create speech for the AI, I followed a few steps.
Firstly, I brainstormed interesting interactions and events that would require the AI to speak, and wrote appropriate phrases for those situations. Then, I asked Chat GPT to rephrase and enhance the sentences in an eloquent manner.
Next, I passed the output of GPT to ElevenLabs to generate the speech.
Once I received the audio files from ElevenLabs, I manually edited them in Audacity to make the sounds more robotic.
I used a technique where I duplicated the audio files several times and shifted them slightly more each time to create a realistic AI sound.

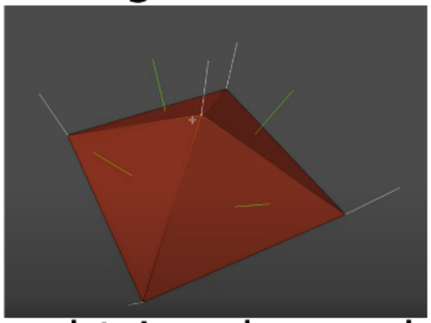


Finally, I added reverb effects to the sound and imported it into Unity.

Normals

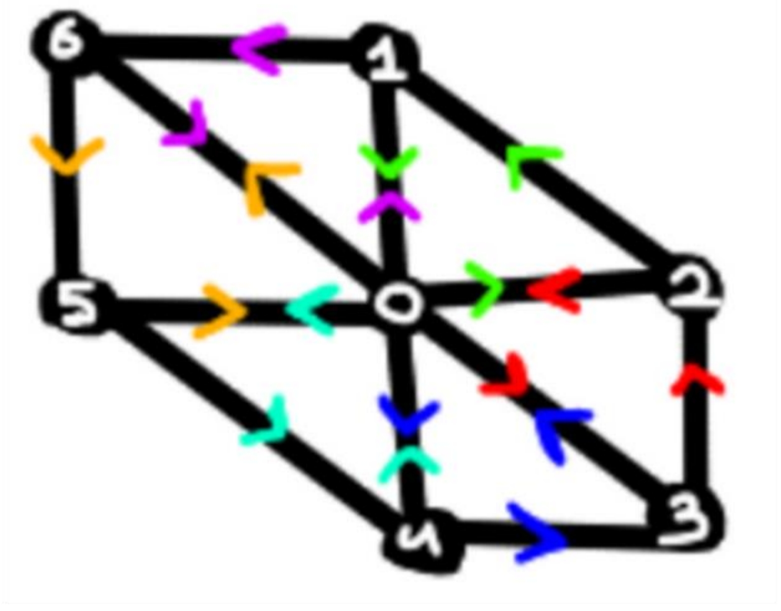
I will now discuss some challenges that I attempted to mitigate in my project. Let's begin with the issue of normals, which arises when two neighboring meshes exist.
In a scene with lighting, all triangles that the light hits should be affected, and the direction of the normals of these triangles determines the amount of shading perceived.
In Unity, triangles are colored based on the colors of their vertices and then blended together.
The same approach is used for normals, where the vertices have normals assigned to them.
For every point on a triangle, we give it a normal which is the weighted average of the normals of the vertices forming that triangle.
The weights are based on the distance from each vertex.
This technique is called "normal interpolating" and it ensures that there are smooth transitions between two triangle normals, resulting in a smooth shading transition.

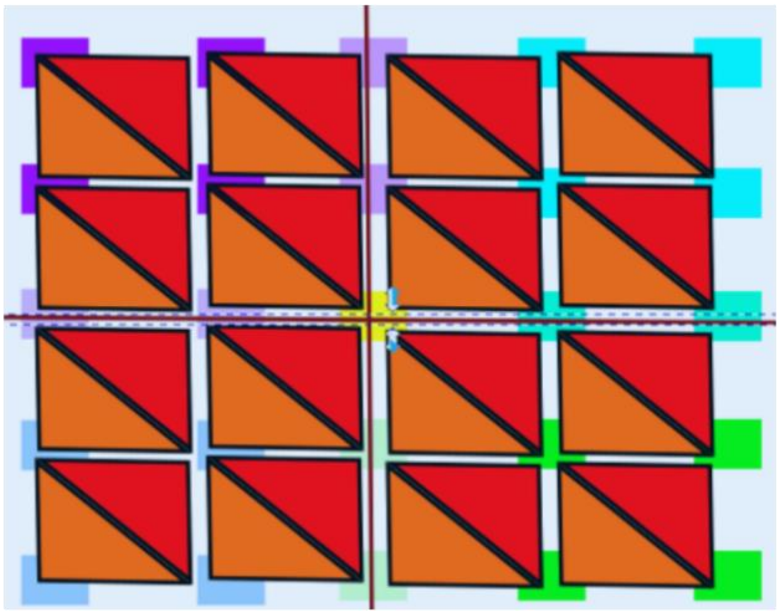




After creating arrays of vertices and triangles and assigning them to the mesh filter, we can use the built-in "recalculate normals" method to automatically recalculate the vertices' normals by averaging the normals of the triangles that use them.

However, this method only considers triangles that are part of the exact mesh that the vertex is on.
This causes issues when there are neighboring meshes, as the vertices on the edges have incorrect normals since the triangles from the neighboring meshes that use them are not included in the average.
This leads to visible lines between neighboring meshes.
To fix this issue, I attempted to recalculate the normals manually by multiplying vectors and taking averages, but I was unsuccessful.

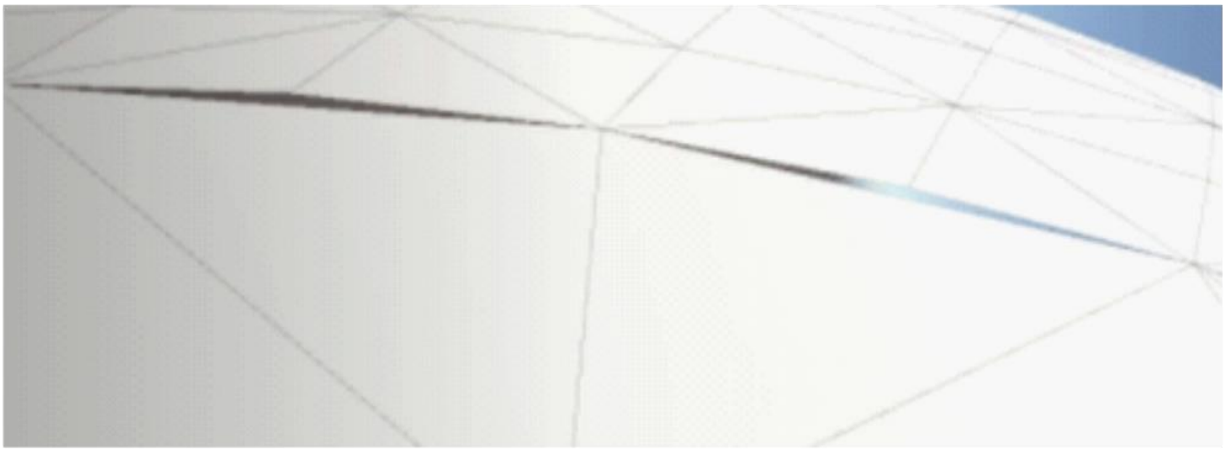




Eventually, I resorted to disabling the lighting, making normals irrelevant and eliminating the lines.
Although this solution worked, it resulted in a loss of visual appeal in the scene.

Gaps

This issue also arises between neighboring meshes, but this time only for meshes that have different resoluitons.

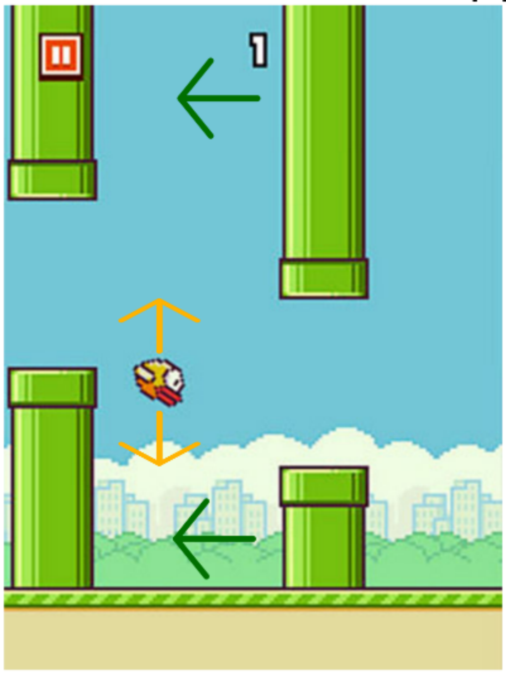


Finding a solution for this problem proved to be a challenging task, as most of the proposed solutions were complex to implement and led to other issues or poor performance. Fortunately, I was able to minimize the impact of this problem by adjusting some parameters.
For instance, I found that increasing the resolution of the meshes helped reduce the size of the gaps between them, as the height changes more smoothly.
Additionally, by increasing the distances in the lookup table, more meshes were given higher resolutions, which slightly lowered performance but also made the gaps between neighboring meshes less noticeable and sometimes even disappear altogether.

Floating Point Inaccuracy

The problem I'm facing in my project would take me at least a week or two to fix, if I even succeed in fixing it.
It arises from the fact that I use small numbers like amplitudes and frequencies for generating small terrain features, such as mountains reaching 10^-3, as well as large numbers like distances between planets that can reach magnitudes of 10^6.
Unity's engine sometimes shows an error message when numbers exceed 10^5, prompting me to use smaller numbers.
As a result, when the spaceship moves in a particular direction and encounters several planets, issues arise with inaccurate positioning of meshes and vertices, leading to noticeable gaps and strange behavior.
In extreme cases, the app may even crash and display an error message.
One possible solution is to adjust the range of all variables in the project from (10^(-3) to 10^6) to (10^(-4) to 10^5), which would result in problems arising when reaching the 40th planet.
Alternatively, adjusting the range to (10^(-5) to 10^4) would delay these issues until reaching the 400th planet, which is much more desirable than the problems that arise at the 4th planet.
However, implementing this solution would be time-consuming since it requires adjusting all numbers in the project, including the lookup table, velocities, distances, and noise parameters.
Furthermore, reducing the range of numbers may introduce floating-point errors due to the reduced precision of the values.

After some consideration, I came up with a new solution that completely resolves the problem, although it requires a lot of work.
Surprisingly, I got the idea from a simple game called "Flappy Bird".
In Flappy Bird, the player controls a bird that must avoid obstacles to progress.
The player feels as though the bird is moving forward, but in reality, the bird stays in one place while the obstacles move towards it and disappear when they cross the bird.



Similarly, in my project, I would make the planets move towards the spaceship at (0, 0, 0), which would eliminate floating point problems.
However, implementing this solution is not straightforward. Moving 100 planets, each with 1000 children and over 400 meshes, in every frame would cause the application to crash.
To address this issue, I could move the planet with only its 4x4 matrix when it's far away, and then start smoothly bringing in the meshes of the 20x20 matrix as it gets closer, as described in the "Changing Matrix Smoothly" chapter.
This approach would require a significant amount of work, and I anticipate encountering many bugs that would need fixing.

Further Development

In this chapter, I will discuss the aspects of the project that require improvement and the things I would like to add to it. First and foremost, I need to fix the three issues that I mentioned in the previous chapters. Currently, I have around 15 handmade color gradients that I use to color the planets. It would be great to procedurally generate colors and develop a formula that produces logical colors for the planets.
Additionally, I regret not writing cleaner code.
In later stages of the project, I encountered a significant number of bugs that consumed a lot of time.
I would like to learn how to write cleaner code for larger projects.

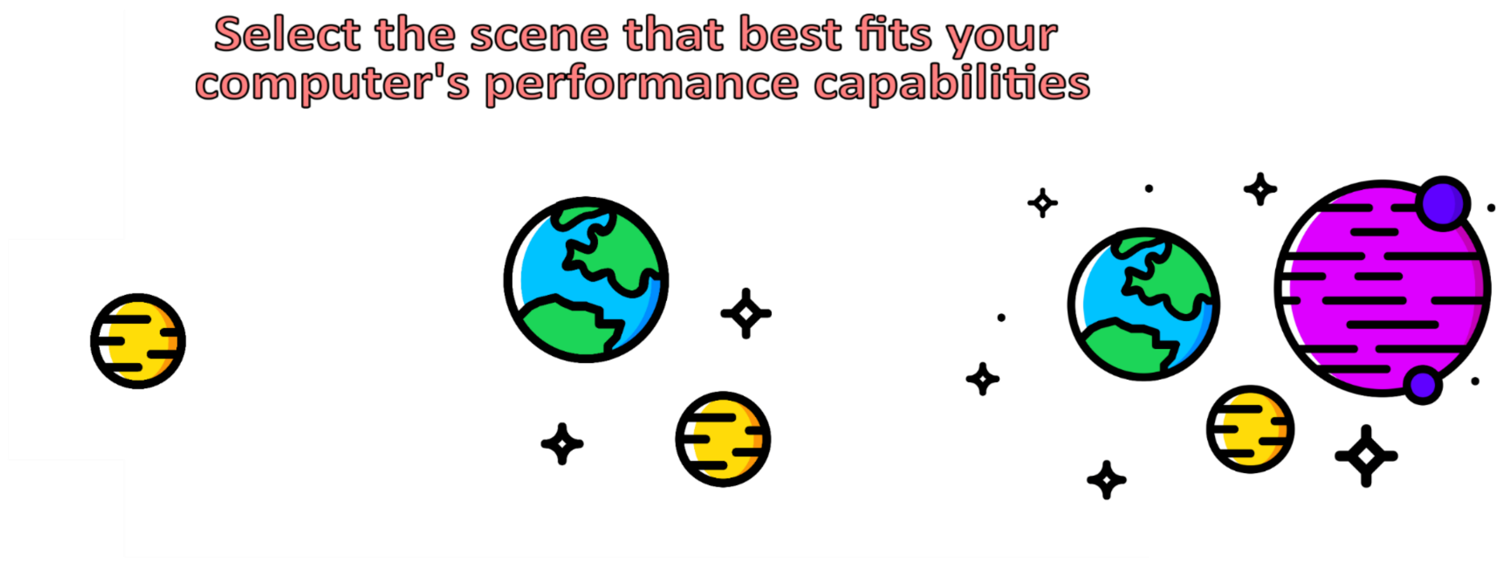
I also want to enhance the shapes of the planets and include plants and animals on them, but this would pose a significant challenge due to the limitations of computer performance. During the project's development, I gained knowledge of the basics of compute shaders, but I did not use them. I would like to use a compute shader to calculate the "calculate height" method and optimize it as much as possible since it is used millions of times per frame. It would be interesting to observe the performance differences when running the method using a compute shader on the GPU instead of the CPU.

Challenges

The project posed a significant challenge that often left me feeling exhausted and frustrated.
While learning new techniques and materials was time-consuming, the primary obstacle was working within the constraints of the computer's performance. Adding new features required optimizing all previous components, which compounded the difficulty.
I encountered several problems that were extremely challenging, such as managing gaps and collisions, and I feared that if I could not find a solution, I might have to abandon the project entirely.
Additionally, I faced a further challenge when I encountered password issues, preventing me from using the lab's high-speed computers.
This forced me to optimize my the project on my personal computer, which consumed additional time but ultimately proved worthwhile when I observed how smoothly the project ran on the lab's machines without any glitches or lag.

Main Menu

So you're interested in trying out the simulation and running the app?
The simulation is designed to run on powerful computers, but I've included a main menu that allows you to select a scene based on your computer's performance capabilities.
The scene on the left has the fewest planets and the lowest level of detail (LOD) resolutions, while the scene on the right has the most planets and the highest LOD resolutions.


Select the scene that best fits your computer's performance capabilities

However, I recommend using the most right scene that your computer can handle, because as I mentioned in the "Gaps" chapter, when using lower resolutions, gaps tend to be more visible.

References

During the development process, I extensively watched videos and read materials from various sources.
Although I came across many ideas, most of them were not suitable for my project.
However, I found Sebastian Lague's YouTube channel to be the most important source of information for my project.
In particular, I learned a lot about turning a cube into a sphere, coloring vertices based on their heights and noise layers, gaps, normals, flat shading, and optimizations from his two series of videos:

https://www.youtube.com/playlist?list=PLFt_AvWsXl0cONs3T0By4puYy6GM22ko8

https://www.youtube.com/playlist?list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3

Another source that I found useful was a person who tried to expand upon Sebastian's series.
However, his implementations, such as quad trees and edge fans, were too complex for my project.
I ended up taking only his back face culling idea and expanded it in the "Advanced Culling" chapter.
You can find his series here:
https://www.youtube.com/playlist?list=PLwRBcuYHwOZ9QVCaZWChCugGIsWuUaTZA

This is the noise script that I used in the project:
https://github.com/SebLague/Procedural-Planets/blob/master/Procedural%20Planet%20Noise/Noise.cs

Last Words

This marks the conclusion of my report.
This project holds a special place in my heart as it was a dream of mine to create something like this even before starting university.
I have always been inspired by the magical things that Sebastian Lague does and I wanted to create something similar.

I dedicated a lot of time and effort into this project, going above and beyond what was required.
The report itself is over 40 pages and looks like a project book.
I hope to revisit it in the future and enjoy reading it or use it as a reference if I decide to add more features to the project.

I want to express my gratitude to God for giving me patience and strength throughout the development process.
I also want to thank my parents and family for their unwavering support, Sebastian for his incredible tutorials that I could draw inspiration from, and my supervisors Yaron and Boaz for their guidance.
Musa also deserves a special mention for the significant work he did in other courses that allowed me to complete this project.
I would also like to thank Lior for his help with fixing problems related to the lab password and downloading applications on the lab's computers.
Finally, I want to thank everyone who has shown interest in my work or has taken the time to read through this report.
If you have any questions, feedback, or ideas, please feel free to reach out to me via this email address:
saadisaadi1@campus.technion.ac.il.