PROGRAMMING API

Trochoidal Wave Shader using Unity Shader Graph API

Lewis Maskell
UP2206042

Contents

Project choices and preliminary research

Project options

For this project there were various options that we discussed openly and listed within our project proposal document.

Some of the results of our idea generation were as follows:

- A single-player chat room that uses the ChatGPT open API tools, where we would develop the means to enable back-and-forth communication between the player and AI, simulating a chat experience.
- A Spotify playlist generator, which would use Spotify's web API to create playlists based on custom parameters and preferences.
- A Terrain generator, which would use a Perlin Noise API within Unity to deform a mesh to generate 3d maps that could be used for gameplay environments.

Single-Player Chat Room

For this project, we considered the use of OpenAI's *ChatGPT* API to generate text responses to inputs in our program. After researching OpenAI documentation a *JavaScript*, *Python* or *Curl* based program (OpenAI, 2024), combined with *Visual Studio* and it's "Windows Desktop Application" API would allow for inputs and responses to be presented through a Graphical User Interface (GUI), similar to chatrooms that gained popularity in the 1990s and 2000s.

According to OpenAI, while the GPT-4o model is the most sophisticated, there are several options that would suit our project (OpenAI, 2024). Many models advertised on their website boast text-to-speech, speech-to-speech and "Vision capabilities." These features would be unnecessary for our project, especially considering the increased cost for an access key.

The GPT-4o-mini model would be most ideal, as it has the lowest cost per input/output token due to the lack of irrelevant features.

This project is the only one requiring payment to use, however the OpenAI APIs provide almost all features required to complete.

Spotify Playlist Generator

For a playlist generator, our project would be able to take several input parameters, such as genre, artist or album alongside potential slider inputs for: "acousticness", "danceability", "energy", "instrumentalness", "liveness", "loudness", "speechiness" or "tempo", according to Spotify's web API documentation (Spotify, 2024). Using these parameters, it would be possible to analyse potential tracks for a playlist, then returning the songs with the best match to these parameters. The web API also has the capability to create new playlists for a specific user and fill them with the returned song IDs.

Spotify's Web API has extensive documentation, providing valuable information on how to integrate the API with our project. Alongside this, it is also free to use with several forums for developer discussions. The web API provides all features required to build our project.

Terrain Generator (final choice)

After consideration, we decided our project should be a terrain generator, as it would be the most relevant option for a game development portfolio, would likely have the most available resources to assist development and was the option that had the largest potential for scalability and distribution of work.

We had decided to use Perlin noise due to its properties as a procedural noise texture, constructed by using gradients (Perlin, 1985), lending itself to the basic features required for a procedural hilly terrain style.

After deciding on this project, we conducted some preliminary research into how Perlin noise is used in terrain generation. As part of this research, we found a report by Zoscak that describes how "textures can be subsequently aggregated in a way that produces a more visually appealing and custom terrain result." (Zoscak, 2023).

Taking this into consideration, we felt that developing a function to generate a planar mesh, followed by applying multiple layers of Perlin noise to deform, would be an adequate solution to our terrain generator.

For our project, we decided to use the Unity editor as it is pre-packaged with all the APIs we would need to build our project. This included a 3D renderer for our scene views, alongside a Mathf.Perlin() function to apply the Perlin noise to our mesh data. Furthermore, due to the APIs being pre-packaged within Unity, there is significant documentation (Unity Technologies, 2024) to assist in our development process.

Another benefit of using the Unity editor, is the built in *Unity Version Control* which enables basic source control functionality, allowing my team members and I to commit our changes remotely, alongside being able to store and switch between saves of previous versions of our project. However, due to some complications during setup we instead decided to use *GitHub* as our primary source control.

Water waves shader

After the first week of development, we found that the implementation of this terrain style was much easier than expected and did not require the working hours we had planned to distribute between our group to implement this solution. As a result of this, we decided to increase our project scope by assigning each member of our team to different aspects of map generation including clouds, water waves and caves. I chose to take on the task of creating a procedural water wave generator and began doing research into various aspects of this.

While I had initially considered a simple C# script to work directly with our existing mesh generator, several articles including a book chapter by NVIDIA on "Effective Water Simulation

from Physical Models" describe how simple wave functions would run more efficiently on a GPU pipeline over the CPU (Finch & Worlds, 2004).

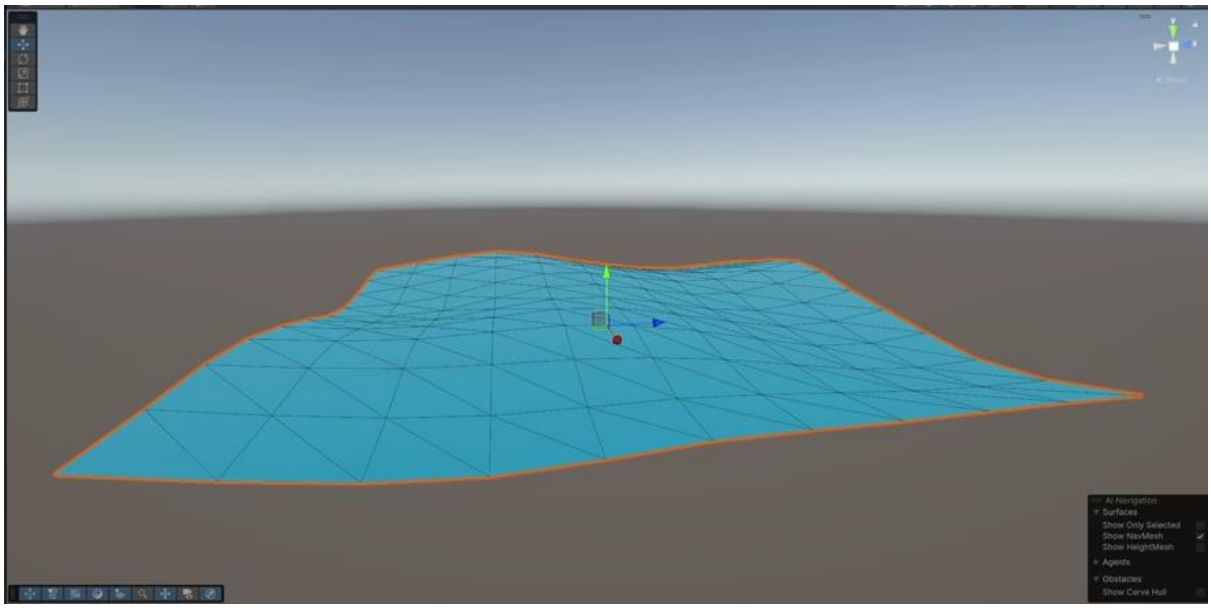
Fortunately, the Unity Shader Graph API, a node-based API typically used for creating advanced textures and lighting, is designed to run on the GPU, using the *Universal Render Pipeline* (URP)

This pipeline also comes with the functionality to deform mesh vertex positions, making it ideal for my task of creating a water waves shader and allowing it to function with our pre-existing mesh generator.

Problem solving

My first attempt at implementing a wave shader was to use a Perlin noise texture, applying the alpha value to the Y value of each vertex in the mesh, and scrolling the texture using an offset and time node.

This approach was quite simple to implement and gave a semi-realistic looking result.



A single frame image of the Perlin noise applied to a plane object created within the editor.

This approach would be well suited to a project that had low performance requirements but lacks fine tuning and control over how the waves could be stylised.

It was after this, that I ran into my first major issue.

When applying the texture to our custom-generated mesh, all vertices would be combined into a single location. After a while spent debugging to no avail, I instead decided to try making the meshes tesselate, as an alternative to our custom mesh generator. This, unfortunately, provided no results.

I finally decided on a different approach by researching different types of wave generation. After a while spent researching, I decided to implement a Trochoidal or Gerstner wave solution. This solution is ideal as it gives me fine control over how the wave can be constructed, as it works by the summation of multiple waves with separate parameters for direction and amplitude (Finch & Worlds, 2004)

This approach is also more realistic, as when observing a wave, the water does not only move up and down but has a horizontal component to its motion. (Flick, 2018)

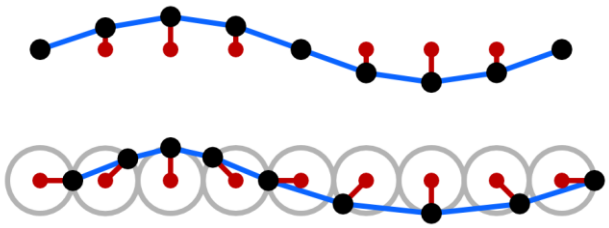
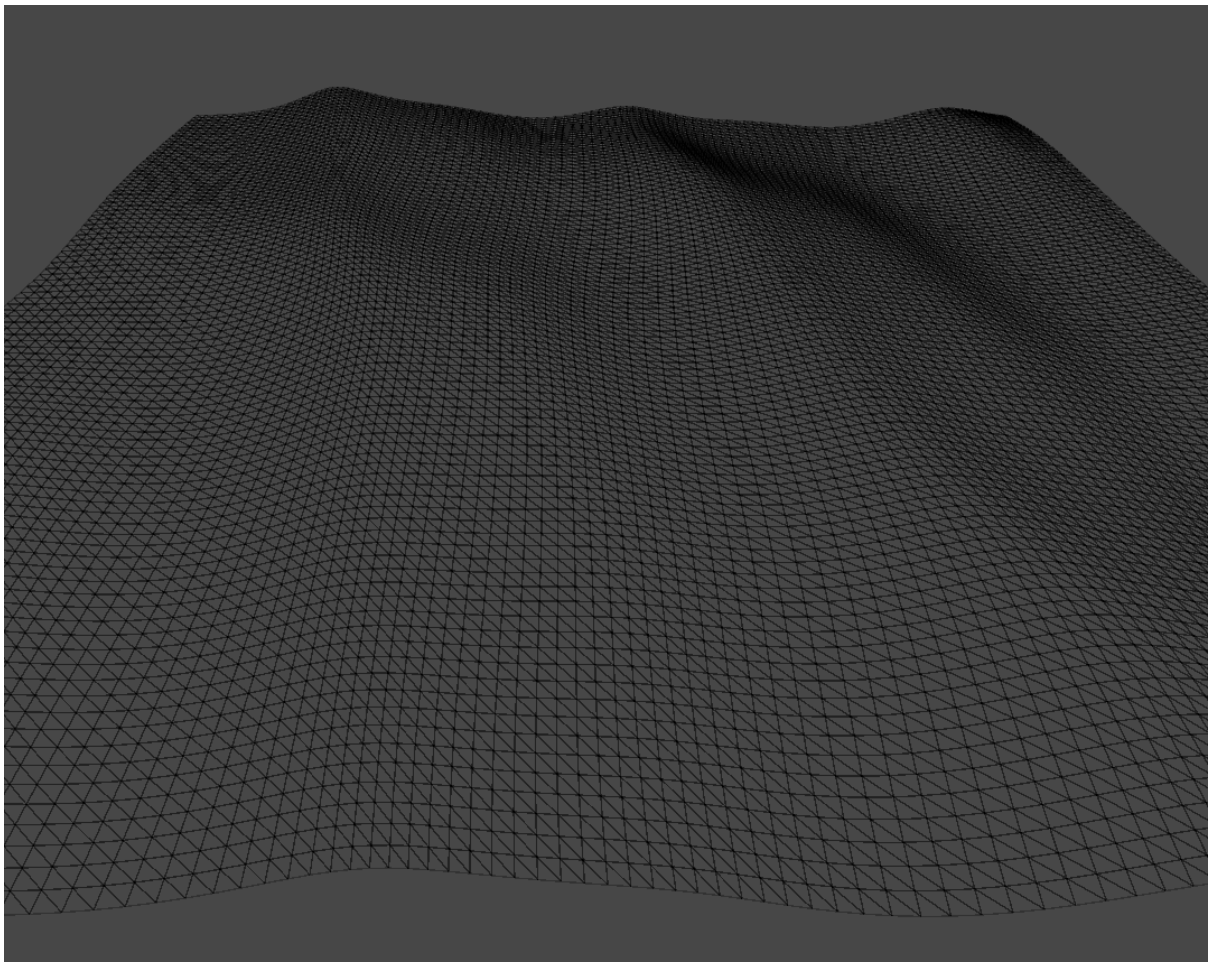


Fig.1 from Catlike Coding, of a sine wave compared to a Trochoidal wave (Flick, 2018)

Through the use of sub-graphs, I could construct a function for the vertex displacement of each individual wave, then by returning the offset, I could sum it together with the output of every other function taking different parameters, to create a highly realistic looking wave.



A wireframe view of 4 separate Trochoidal waves applied to a single mesh.

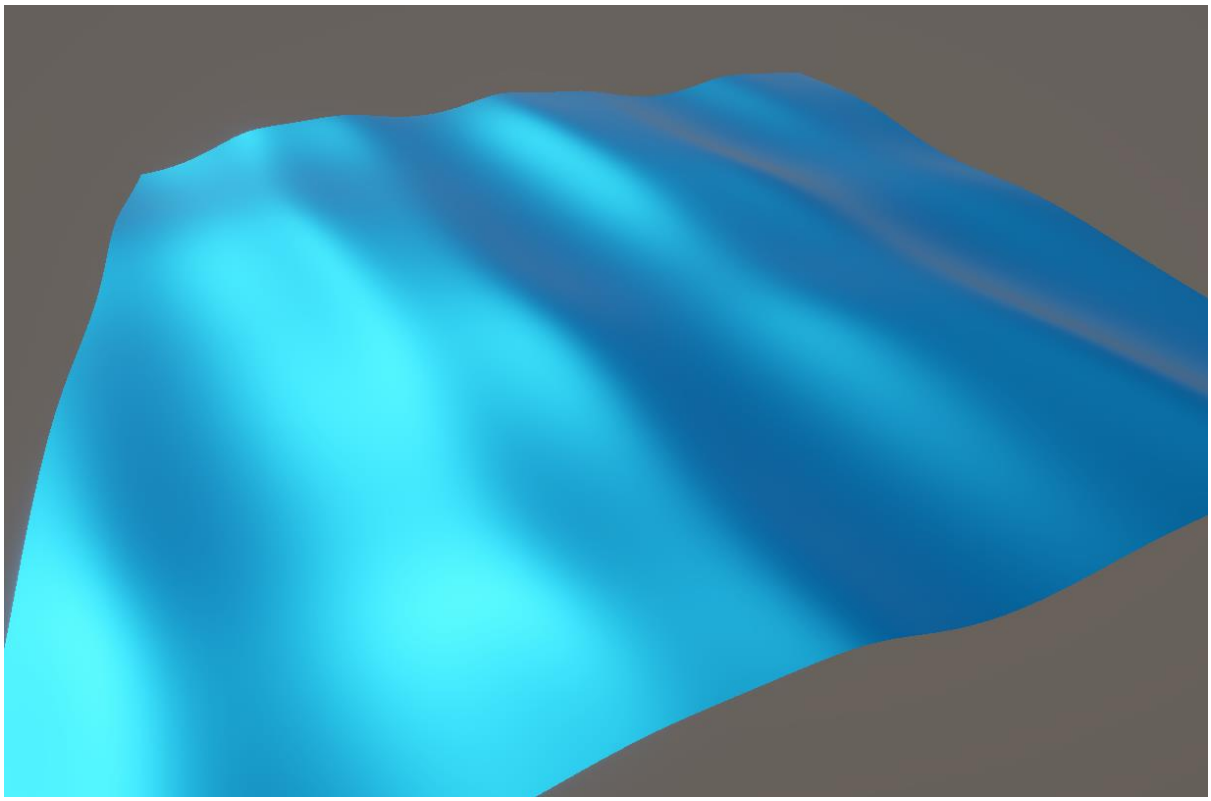
As shown in the image above, this approach worked perfectly with the custom mesh generator function produced by my teammates.

The second major obstacle that I had to overcome was the recalculation of the mesh normals.

As an unlit texture, there were no shadows or reflections being cast on the plane surface. From my research, including the paper by Flick, suggested a possible solution to calculating the normals of my mesh, would be to repeat the wave function twice with positional offsets, in order to calculate a gradient from the returned vertex locations. This, I struggled to implement both conceptually and practically, as the resulting normals would never look quite right.

Through further research, I found that it was possible to return the vertex normals for each individual wave function, through differentiation, as the function itself is explicit. (Finch & Worlds, 2004).

Finally, by taking the sum of the normals, and the sum of the tangents for each wave function, I could cross product and normalise the results to return exact UVs. This approach worked perfectly with the rest of the solution.



A shaded view of the Gerstner wave function with correct normals.

Appraisal

API Appraisal

Through this experience, the primary API I used was Unity's node-based Shader Graph.

While this API is primarily designed for the purpose of creating complex procedural textures for game assets, the functionality it provides by allowing the user to mathematically deform the vertex positions of meshes, was ideal for the application of a Gerstner wave function.

The node-based system used in Shader Graph was incredibly intuitive and easy to learn. The ability to search nodes and functions, then connecting them via the input/output pins meant I did not need to learn any syntax to begin coding effectively. The only downside to this, is that as the complexity of the code increases, the resulting graph becomes less legible and harder to follow. This can be averted by using sub-graphs as functions wherever possible.

The second major benefit of Shader Graph, as stated previously, was the fact that it was built to run through the GPU pipeline and suited the techniques I had planned to implement previously. This meant that the implementation of my solution was more optimal than if I had used a script that runs through the CPU pipeline.

The fact that Shader Graph can easily be installed as a package within the Unity Editor, meant that I could implement a range of parameters that I can edit in the inspector. This greatly improved my ability to debug my code as I could see the results of my changes as the code ran in real time.

Using Shader Graph also improved the compatibility of my solution with the rest of my teammates code. Since the solution can be applied to any mesh as a texture, it means that my team can update any part of the project, without breaking any functionality of the wave shader. This also opens the possibility of implementing a procedural Level of Detail (LOD) management system, to increase performance in larger scenes.

On the other hand, the node-based design of Shader Graph did limit the compatibility with other code classes. If I were to expand my Gerstner wave solution to a Fast-Fourier Transform (FFT) solution, I would likely have to use a different code-based solution, as it requires a much greater number of input parameters and calculation.

Another challenge I had to overcome when completing this project, was the lack of documentation and resources for Shader Graph, compared to those available for a code-based solution. However, this was offset by the intuitiveness that the API provides.

Solution Appraisal

The original approach I had to this project using scrolling Perlin noise, was rather easy to implement, and could remain an effective solution for projects with lower budgets, performance requirements or development time.

My final solution was an effective approach for the project as the resulting wave shader was realistic, due to the fine control of multiple parameters allowing for precise and varied adjustments, alongside its similarity to how waves behave in deep water. The GPU pipeline allowing for parallel processing of vertices, means my solution was also well optimised.

If I were to continue working on this project, I would want to begin working towards a FFT solution, or at least a Gerstner solution that includes a depth property. This would allow it to be fully integrated with procedural environments, where a single texture can be applied to a plane, and allow for both shallow water shore waves and deep-water ocean waves without any external calculations.

I would also try to implement a functioning LOD system, which would reduce the triangles in the meshes at further distances from the camera, saving a lot of GPU processing time. I would likely attempt this in a code-based solution again using the URP.

Collaborative techniques

Throughout this project, there were several techniques my teammates and I used to collaborate with each other. Without effective communication of ideas, I am certain my solution would have taken much longer to implement.

Discord

After meeting my teammates, it was essential that we set up a mutual correspondence. Since we all had set up accounts using *Discord*, it was the clear choice for us to use. This is also because a private group chat provided us with several useful features. The first of these, image and video sharing, enabled us to show each other our code remotely or share videos and visual resources that may be found useful. This feature, combined with the fact that Discord will store message history, videos and images sent in chats near indefinitely, meant that the resources shared could be found and re-used again later.

Discord also provides a voice call and share screen functionality, which further expanded the options my team had to communicate and collaborate remotely.

In future, I will likely use *Discord* again, however the software is somewhat unprofessional, as it is intended to be used as a chat app. Although, the benefit of meeting in-person cannot be overlooked.

Git Hub

The next essential technique we used for collaboration was source control. This allows us to remotely work on our project and commit changes to an online repository, so that everyone may always have access to the most up to date version of the project.

As mentioned previously, our first consideration for a source control software, was to use *Unity Version Control*. This was because a pre-packaged source control in our editor, would allow us to greatly improve our workflow, as *Unity Version Control* enables the user to switch between versions of the project within just a few clicks. This functionality would be primarily useful for debugging. Due to some issues during setup though, we instead decided to use *Git Hub* as it provides mostly the same functionality. Another benefit of *Git Hub* is that the project repository is saved online using their website, meaning if the source control software were to fail, the project can still be downloaded in its entirety.

The only downside to these source control softwares, is that two members of the team cannot work on the same code at the same time, as it would lead to merge issues when pushing changes. Our solution to work around this, was to first ensure that any time we interact with each other's code, we would ensure that there were no changes pending beforehand. Secondly, we would work on aspects of the project in separate scenes, so that scene elements which may be essential to parts of the project, could not be accidentally changed by other group members.

Stand-up meetings

Each week, our group would attend in-person meetings with the module leader and other groups. During these sessions, we would participate in stand-up meetings. These stand-ups allowed each member of our group to share and fully explain the parts of the project they had been working on up until that point. This meant each member would be given accountability for doing work each week, which I believe greatly accelerated our development process. Another benefit of these stand-ups was allowing us to receive feedback from other group members and course leaders. It was this feedback that originally helped us to decide on what we should expand our project to after implementing the first terrain solution.

The main limitation of this technique is that it requires an organised time to meet with everyone, and you cannot guarantee that all group members may be present to receive the information shared. However, the benefits of this cannot be overlooked and I would like to continue doing this again in the future.

Bibliography

Finch, M., & Worlds, C. (2004). *GPU Gems*. Addison-Wesley Professional. https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models

Flick, J. (2018). *Waves Moving Vertices*. Catlike Coding. https://catlikecoding.com/unity/tutorials/flow/waves/Waves.pdf

OpenAI. (2024). *OpenAI Platform*. https://platform.openai.com/docs/guides/text-generation

Perlin, K. (1985). *An Image Synthesizer*. Courant Institute of Mathematical Sciences, New York University. https://www.cs.drexel.edu/~deb39/Classes/Papers/p287-perlin.pdf?

Spotify. (2024). *Spotify Web API*.

Unity Technologies. (2024). *Unity Manual*. https://docs.unity3d.com/6000.0/Documentation/Manual/UnityManual.html

Zoscak, J. (2023). *Generative Technology: Multi-Layering Perlin Noise Textures in Terrain Generation* [CS4991 Capstone Report]. University of Virginia.