University of Portsmouth

School of Film, Media and Creative Technologies

Final year Final year Project undertaken in partial fulfilment of the requirements for the BSc (Honours) in Computer Game Technologies

Physical Sailing Simulator for Game Purposes

By
Lewis Maskell
UP2206042

Supervisor: Nipan Maniar
Module: M31481
April 2026

Project Type: Artefact

I consent to my report in this attributed format (not anonymized), subject to final approval by the Board of Examiners, being made available in the Library Dissertation Repository and/or the school digital repositories for a maximum of 10 years.

YES

This project investigates the development of a real-time sailing simulation in Unity, alongside the development processes, analysis and user evaluation of doing so. The artefact combines several systems, from a Gerstner style vertex displacement shader for an Ocean simulation with complimenting water surface shaders, a multi-point sampling buoyancy system and a sailing movement behaviour that is constructed from real polar performance data. Development process followed an iterative methodology, supported by broader Kanban task management. Results are justified with in-depth analysis of performance statistics and qualitative user feedback. A literature review is also conducted to evaluate how existing options for core features are implemented and vary, with chosen methods for building features explained in detail. A self-reflection is conducted and changes for future development are suggested, in order to target an industry-standard deployment.

Contents

Primary Aims

The aim of this project is to develop a sailing simulator, which aims to reflect how a modern, single masted laser-style vessel would move with respect to wind and waves. The movement of the vessel should be as realistic as possible, without having to compute every aspect of a 100% accurate physical model. The decision for this approach is made to ensure the final solution remains performative on a variety of machines.

Specific goals

The user should be in control of the vessel through steering a rudder and controlling the length of a virtual “Mainsheet” that will determine the lateral reach of the sail.

The boat should travel with a realistic motion with respect to the following.

- The strength of the wind
- The angle of the wind to the sail
- The force of buoyancy caused by waves
- Drag forces acting upon the boat by the water
- Rotational forces acting upon the boat by the wind
- Rotational forces acting upon the boat by a keel

At minimum, the artefact should be able to run at 30 frames per second (maximum 33.33ms total compute time) for over ten minutes on the development machine to ensure the artefact is performative and there are no significant memory leaks.

Development machine hardware specification:

- Windows 11, 64-bit.
- 12th Gen Intel (R) Core (TM) i7-12700H (20 CPUs), ~ 2.3 GHz
- 40960MB RAM
- NVIDIA GeForce RTX 3060 Laptop GPU
- 5994MB VRAM
- DirectX 12

Secondary aims

As will be required for the buoyancy component of the simulation, the project will include a Gerstner style wave shader. I intend to build upon a previous implementation of a Gerstner style vertex displacement shader where I will be solving issues I had faced with inaccurate normals calculation and incorrect world space displacement.

I also intend to improve upon the visual appeal of the wave shader by adding surface texture to represent ripples and implementing sea foam that outlines objects intersecting with the surface of the water. I also intend to make the water surface slightly translucent, which will be the final change I make to build a shader that aims to be representative of an open body of ocean water.

I also aim to gather user feedback on completion of the artefact to prove that relevant requirements as outlined previously have been met.

Risk to development

There are several easily mitigated risks that could cause development of the artefact to be incomplete.

Risk	Likelihood	Mitigation
Loss of data or corruption of files	Very low	Use of source control through GitHub ensures a cloud backup that can be cloned if required. Updates to the source on GitHub can easily be made.
Development hardware failure	Extremely low	Again, using GitHub, a copy of the project on the cloud means a clone can be made on a different machine.
Over-scoping of features causing the project to be rushed and/or unfinished	High	Setting a clear plan of features, organised through Trello and scaled on a timeline will allow for easy evaluation of if or when features need to be dropped
Features taking longer to complete than expected causing other features to remain rushed and/or unfinished	High	Regularly re-evaluating the remaining features with respect to the original timeline plan, then re-scoping if required

Literature review

Before beginning work on this project, it is essential to explore and review existing, relevant solutions.

Sea of Thieves (Rare, 2018) is one of the most successful open ocean adventure games since its launch in 2018. *Sea of Thieves* then reached over 66,000 players on its *Steam* release in June of 2020 (SteamDB, n.d.), with users often hailing the game's visual appeal as "stunning [...], especially the sea." (luna tuna, 2020) as well as an IGN review from 2018 describing how "Getting to your destination [sailing] is, arguably, the greatest part of Sea of Thieves." (Tyrrel, 2018).

In a SIGGRAPH talk from 2018, Kozin discusses the technical art of *Sea of Thieves*, where it is mentioned that their ocean system implements the Fast Fourier Transform (FFT) for vertex displacement and Jacobian compression to determine sea foam, as a benefit of operating with oceanographic spectra (Kozin et al., 2018). It is also then later mentioned that their implementation requires only 0.638ms of compute time to update water FFTs, a more complex system than what I plan to implement, well within the defined compute budget.

Kozin also references a 2001 paper by Tessendorf titled "Simulating Ocean Water" as the fundamentals for building the *Sea of Thieves'* FFT based water system. However, Tessendorf's focus is primarily on the construction of visually realistic ocean waves and does little to explore how one may implement the movement of physical objects within his solution.

Tessendorf does however briefly discusses the use of Gerstner waves, where within the context of computer graphics, they are described as a "fairly realistic representation of typical waves on the ocean when the weather is not too stormy." and "relatively light [mathematically] compared to FFTs" (Tessendorf, 2001).

Fournier and Reeves explore how the abstract trochoidal movement observed in the particle motion of waves can be used alongside existing principles of hydrodynamics to construct a "realistic motion" for a "range of waves from very low ripples to high storm waves" (Fournier & Reeves, 1986). This abstraction of real motion is appropriate for this project as it will target the ideal balance between performance and realism. While, again, this source fails to explore how an individual physical object could interact with it, the abstraction of required computing and simplified implementation lends itself to an easier integration of buoyant behaviours.

To create a buoyant force due to the motion of simulated waves, it is first essential to consider Archimedes principle, wherein the upward buoyant force exerted on an object is equal to the weight of the fluid that the body displaces. For a simple object floating on a water surface, the sum of forces can be abstracted to the force of gravity, acting on the object's centre of mass and the force of buoyancy, acting on the centre of mass of the submerged volume (Gábor, 2025).

For a wave moving across the volume of an object, the centre of buoyancy will vary. This variation in submerged volume needs to be recalculated every time the wave simulation updates its position. Gabor explores a precision approach to this problem through determining the fully submersed faces of an object mesh and the proportion of partially submersed faces to construct a highly accurate submersed volume (Gábor, 2025). However, this method is highly computationally expensive, reflected in Gabor's own results wherein a mesh of ~5000 triangles resulted in compute times of 0.75ms.

Despite the computational expensiveness of Gabor's approach, the fundamentals of applying the calculated buoyant force remain the same. As with the approaches discussed for water simulation, an abstracted mathematical method will again be necessary.

When considering how a boat should move with respect to the wind in the context of this solution, it is necessary to step away from the user-friendly, simplified solution offered by *Sea of Thieves*. The nature of a "laser-style vessel" means exploring how a boat sailing close-haul to the wind can travel faster than the wind itself.

The sum of forces acting on a moving sailboat are not simply limited to the force of the wind and the force of drag acting against the motion. This can include the lift forces due to Bernoulli's principle, as well as the normal reaction forces resulting from the Coanda effect (Landell-Mills, 2020). While Landell-Mills' exploration into the combined result of the Coanda effect, Bernoulli's principle and reactionary forces are insightful for generating accurate models of sail forces, the prediction model is not appropriate for a game development context as it takes away from a developers control over the behaviour of boat movement.

As argued by Swink, the way a player perceives and interacts with a game takes priority over physical fidelity (Swink, 2009). Therefore, it is important to again consider alternative abstractions of how the sail forces can be emulated.

Justification of Specification

Unity, C# and .NET

The development environment I have chosen to use for this project is Unity version 6.0. This choice was made with respect to the following reasons.

First, Unity as an Integrated Development Environment (IDE) provides extensive libraries and framework to support the basic requirements to begin development. These range from providing mathematical functions and constants in the *Mathf* (Unity, 2026a) library, to User Interface (UI) management and rendering with the Universal Render Pipeline (URP) (Unity, 2026f) These libraries alongside a 3D-ready scene means that significant time can be saved by not having to set up frameworks before beginning development, allowing more of that time to be spent focusing on building towards the goals that I have previously outlined.

Secondly, the Gerstner vertex displacement shader that will be used as a starting point for development was built with Unity's *ShaderGraph* API (Unity, 2026c) using the URP, a rendering pipeline within the Unity IDE that provides all of the tools and features needed to improve the vertex displacement shader into an ocean water representation.

Furthermore, Unity has existed as an IDE since 2005, meaning there is extensive documentation for all the libraries that I intend to use. The secondary benefit of this is the vast community of developers posting questions, answers and tutorials through Unity developer forums and independently, which will serve as methods of research when encountering a more dynamic issue that is not covered in official Unity documentation.

Alongside this, the Unity IDE is frequently updated and maintained. In the unlikely event that I encounter any engine problems; I can switch versions or seek professional assistance.

Unity also provides their own garbage collection system (Unity, 2026e), which will automatically prevent any easily encountered memory leaks. However, this system does require processing time as well as system memory, so relying on the garbage collector is not desired. I will continue to build this project with performance in mind.

Finally, the Unity game engine implements the *.NET* framework. This framework provides many advantages, including some already mentioned, however the main benefits of the framework include a consistent object-oriented programming environment (Microsoft, 2026), which will compliment how I intend to implement some features.

Gerstner Waves

I have chosen to build upon my existing Gerstner wave shader for several reasons.

The primary benefit of using an existing solution as a starting point, allows for a greater proportion of available time to be spent developing features that would have otherwise been out of scope. Furthermore, I will have the added benefit of already being familiar with the solution and its requirements. This base level of understanding will again save time when expanding upon the solution to develop an ocean shader that satisfies the target goals.

There are several existing techniques that would be suitable to simulate an ocean surface, though all options vary significantly when considering visual appeal, performance, and consistency factors. Fast Fourier Transform (FFT) style waves are often considered the industry standard approach for ocean surface simulation, with this technique also being implemented in games such as *Assassin's Creed IV: Black Flag* (Ubisoft, 2013). This is often due to its striking visual similarity to real bodies of open ocean water, while still maintaining an appropriate compute time for gaming applications.

In an ideal scenario, implementing an FFT-based ocean simulation for this project would produce a visually realistic ocean surface, while maintaining performance goals and still allowing for a similar implementation of buoyancy calculations. However, after evaluating the available time within the scope of the project alongside my own skill as a programmer, it is highly unlikely that I would be able to complete the artefact to a degree that satisfies all success criteria.

Conversely, a scrolling Perlin-based heightmap for vertex displacement would be undesirable for this project. Despite being the simplest option compared to a Gerstner or FFT based solution, a scrolling Perlin texture would not allow for the desired level of visual complexity for a realistic ocean surface. If I were to build a more stylised solution that may be targeting devices that lack graphical processing power, this solution would receive more consideration.

Finally, the nature of the Gerstner solution as a discrete, parameter-based formula lends itself to how I intend to implement buoyancy to the project. The buoyancy solution I will implement needs to be calculated on the CPU. To remain performative, the wave shader must leverage the benefits of GPU processing. Due to the nature of Unity, it is not efficient to exchange data between the CPU and GPU. However, maintaining a single reference for input parameters

shared by both the shader and buoyancy simulation means the calculation for vertex displacement can be done separately with the same result.

Buoyancy

As with water simulation, there exist multiple approaches for simulating the buoyant force acting upon a virtual object. Each approach again varies between their performance cost and physical accuracy. The most physically accurate solutions involve the Archimedes principle, wherein determining the submerged volume of an object it is possible to calculate the magnitude and direction of the buoyant force acting upon that object (Mohazzab, 2017). However, implementing this solution by fluid simulation is only suitable for engineering solutions, as the computational requirement is significantly higher than other available options, while the difference in visual appeal to an average user is negligible.

The other alternative to a submerged-volume style approach would be to determine an approximate submerged volume by measuring the submerged mesh faces, then calculating the centre of buoyancy and buoyant force from the submerged set of faces. While this approach is less computationally expensive than a fluid simulation and would work with the chosen implementation of a water surface shader, the CPU time required and the lack of scalability with large, complex meshes makes this approach undesirable.

At its simplest, a height trace of the water surface at a given point can be used to approximate a simple object's given position, yet this alone will not provide any realistic rotational behaviour. However, by combining multiple simple trace points, it is possible to build a system that operates on a similar principle to the submerged set of faces approach, wherein forces are applied at given points relative to the target centre of mass, providing the otherwise missing rotational behaviour. The key difference between this approach and the submerged faces approach is that the developer has full control over the number and position of trace points, making the multiple point trace approach much more scalable and efficient. While the behaviour may not be as physically accurate, it will be suitable enough to provide a visually appealing solution.

Methodology

The chosen development cycle for this project is a combination of Kanban and Iterative cycles. Due to the nature of the artefact, distinct features that have already been outlined and specified lend themselves to be easily segmented into individual Kanban tasks. Due to the nature of certain features being dependent on other systems, such as buoyancy being dependent on a coherent and specific wave shader formula, the Kanban tasks can also be plotted onto a timeline, as a useful reference for when features may need to be rescoped.

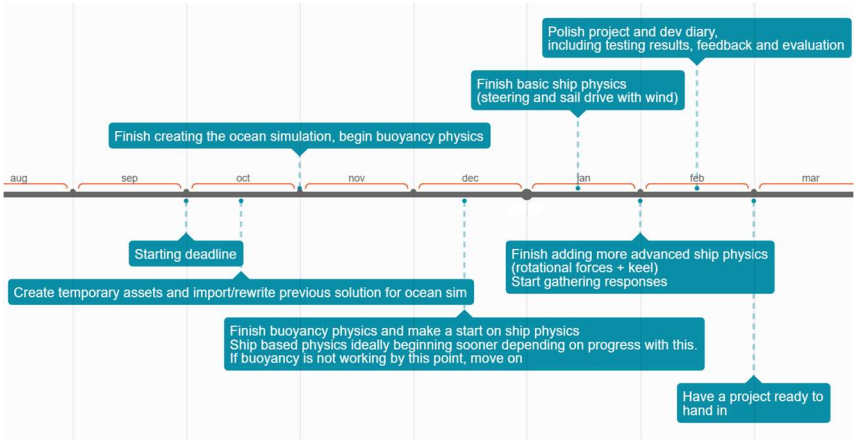
While various complex industry standard tools exist for kanban management, for example, *Jira* (Atlassian, 2025a), which offers many features such as task assignments, calendars, dynamic links to external documents and tools. I have instead opted into using *Trello* (Atlassian, 2025b)

for its simplified and easy-to-use style. The scope of this project and the irrelevance of team management do not necessitate the use of a more feature-complex tool.

The need for an iterative development cycle is due to the uncertain nature of the integration of individual features within the wider project. While it may be possible to plan in detail, with source backed evidence of how a feature should function, the context that it will be implemented in is completely unique and requires iterative adjustments to ensure integration is seamless. This methodology is further supplemented by the decision to use *GitHub* (GitHub, n.d.) for source control, as the ability to develop features within branches of the main source significantly reduce integration risks as different parts of the project can be worked on in parallel if needed.

Planning, Design and Implementation

Attached below is the original project timeline, with week 1-4 dedicated to solving the initial ocean simulation required to begin implementation of buoyancy.



While the original approach was to reconstruct the existing *ShaderGraph* solution using HLSL, it was very soon apparent that this method was taking far too long and needed to be approached in a different way to be completed before the first milestone. This pivot was conducted successfully and development continued to schedule until the end of December.

The next major rescope of features occurred in late January, when the ship movement was taking much longer to cleanly implement than expected. This was primarily due to an oversight with the original implementation of buoyancy that required an extension to the system, which meant that other less essential features needed to be considered for removal. By moving the “rotational forces and keel” to the backlog, enough time was made to complete the essential movement by the project ready deadline.

As mentioned previously, the finer task planning and management was handled using *Trello*. A kanban-style board including sections for “Obsolete, Backlog, In-Progress, Testing & Polish, Complete”. By moving tasks between these sections, it gave a useful visual approximation of the work required to complete various features. This was then used to justify the re-scoping of previously mentioned features.

An additional value to using *Trello* was the ability to add checklists to more complex tasks. This further improved the granularity of the task planning, which would evolve alongside the iterative development methodology.

While a project overview is good for managing scope and timelines, it is still necessary to plan and design the individual functions making up a feature to ensure understanding, good practice and smooth integration. To supplement the functionality design, a combination of a virtual whiteboard using *Miro* (Miro, 2026) and a *Draw.io* (draw.io, 2025) flowchart creator was used. Primarily, the time saved by having a universal area to save reference images, drawings and links was invaluable to maintaining a smooth and efficient workflow. Furthermore, the ability to quickly draft sketches was incredibly useful to act as a visual reference for implementing systems involving movement, apparent wind and rotation as the correct frames of reference would have otherwise been incredibly difficult to implement. The use of *Draw.io* flowcharts was equally important, as there were several key systems in the project that required a very specific order of operations, which a flowchart was well suited to help design. These flowcharts could then also be saved on the *Miro* whiteboard, for easy reference when programming.

Use of AI tools for development

While the use of AI (artificial intelligence) can often be a heavily debated subject, appropriate and careful use can rapidly accelerate development processes. For the nature of this project, there are far too many interconnected systems for a generative AI model to directly produce entire features. Furthermore, blindly trusting the code generated by an AI agent without adequate understanding or evaluation often leads to causing more issues than it would otherwise solve. However, there are two main instances which were leveraged during development, where the benefits of an AI model can be fully realised.

First, was as a documentation reference. The documentation for well established game engines such as *Unity* is often highly extensive and frequently updated. Often, without knowing prior key words or phrases, searching engine documentation manually can be an extensive and tedious process. The nature of a LLM (large language model) such as *ChatGPT* (OpenAI, 2026) to identify patterns and relationships rather than following a strict programmed structure (Garnelo & Shanahan, 2019) means that as a lookup tool, where prompts can be vague or non-technical, the process of finding needed pages of online documentation can be greatly accelerated. Furthermore, when using *ChatGPT* it is possible to request and inspect the links to sources used to generate a response, which will also frequently include relevant documentation.

The second major benefit of using AI tools to assist in the development of this project was to analyse and debug logic errors. Most IDEs can provide syntax highlighting to ensure code is clean for compiling, yet it is often entirely the responsibility of the developer to spot and debug logic errors. While logic errors can be avoided using flowcharts and good planning, it is still highly likely that they will be encountered at some point in the process. This then typically requires extensive iterative testing to determine the root cause of the issue before a solution can be developed. By describing an expected behaviour, the output behaviour and the section of malfunctioning code, an AI assistant has enough context to suggest several reasons why a logic error may be occurring. This can then be used to rapidly accelerate the analysis process before working on a solution. While the AI assistant may not be able to produce a 100%

accurate solution every time, the total time saved can be spent towards working on other features.

Attributes of the artefact

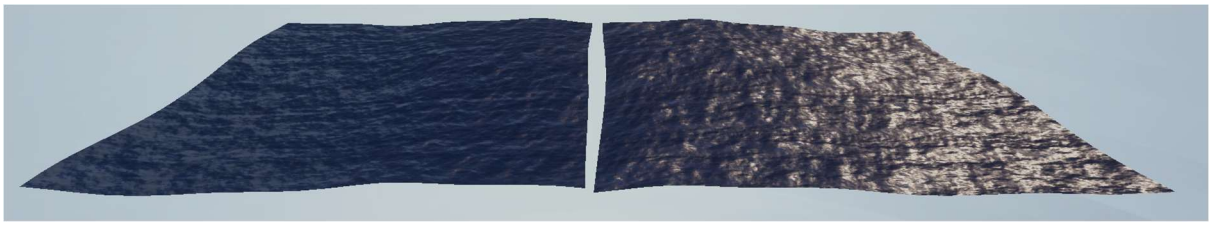
Water Effects

The first feature that needed to be completed was the construction of a convincing ocean water simulation. For smooth integration, the water simulation needed to be developed with two major components in mind.

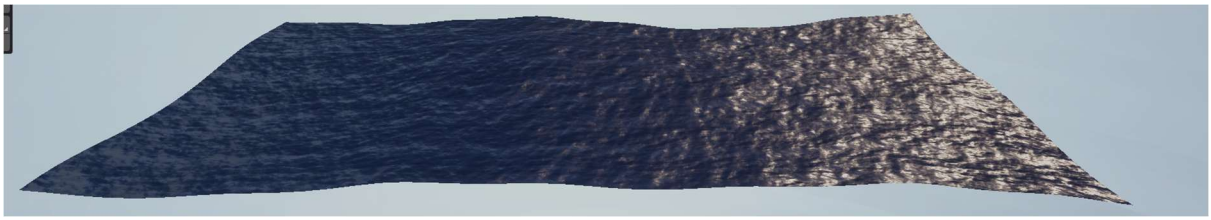
First, was the discrete vertex displacement formula, that needed to leverage GPU processing to remain performative. There were three relevant options to choose from when implementing this, from a FFT implementation, which would take significant time to implement but would yield highly realistic and appealing results. Second, would have been a scrolling texture heightmap implementation, that could be constructed using *Perlin noise* (Perlin, 1985). This implementation would have been very performative and easy to implement but would lack the visual fidelity available in alternative solutions. Third, as the chosen solution, would be a Gerstner-based solution, which targets the middle ground of visual fidelity and complexity as offered by the other two solutions. Furthermore, it satisfies the requirements for the later implemented buoyancy system.

The second reason why the Gerstner style implementation was chosen is because I could begin working on this solution by importing the feature from an existing project that I had developed previously. Furthermore, the existing solution had been developed using Unity's *ShaderGraph* (Unity, 2026c) API allowing for a smooth transfer but meant that I was unable to use HLSL, as originally intended.

Despite this, there were still several issues with the solution that required fixing to make the water appropriate for this project. First, was the fact that the vertex displacement was operating within object-space, making any mesh tessellation impossible and buoyancy significantly more difficult to implement.



Planes with vertex displacement applied in object space



Same two planes with displacement applied relative to world space position

Secondly, the normals being calculated in the original solution were completely incorrect. While it may seem obsolete to be manually calculating face normals when vertex displacement is already functional, the Unity rendering engine treats the displaced vertex normals as being the same as those on the undisturbed mesh, with no automatic recalculation.

While recalculating normals could be seen as a highly computationally expensive process, through iterative approximation, there was a more performative solution available due to the nature of the Gerstner displacement as a mathematically discrete formula. By simply differentiating the formula used to displace vertices at any given point, you will end up with a formula that instead returns the gradient (normals) at any given point. Fortunately, after evaluating the sources used previously to construct this wave shader, in chapter 1 of GPU Gems (Finch & Worlds, 2004), Finch & Worlds provide the differentiated formula, alongside the required method to interpret the binormals and tangent values (Finch & Worlds, 2004). This was then implemented in the *ShaderGraph* solution.

Without correct normals, scene lighting would not have correctly interacted with the water surface and implementation of further methods for water surface visual effects would not have been possible.

The second set of features added to the ocean shader are a series of surface visual effects. These effects can be categorised into the following.

- Transparency
- Depth colour gradient
- Surface ripples by normal textures
- Sub-surface distortion due to ripples
- Sea foam

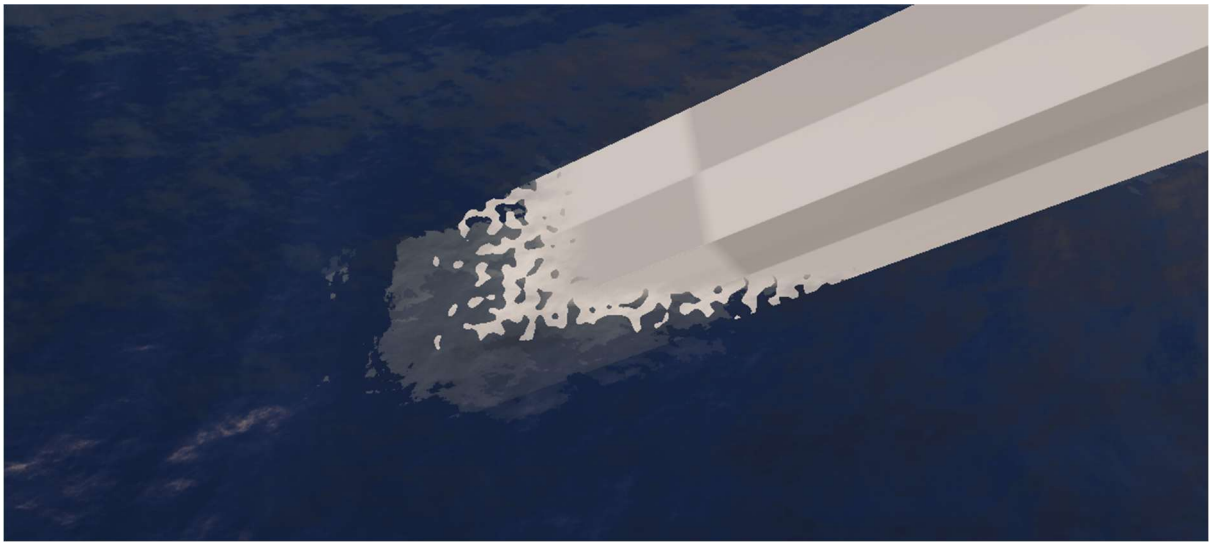
The transparency in this shader is not applied by setting the alpha value of the shader fragment. In order to support sub-surface distortion and refraction, a screen sample is taken, with the objects behind the water surface distorted, then re-applied directly to the base colour node of the shader fragment (Cloward, 2025).

The depth colour gradient and sea foam are applied using very similar techniques to one another. This technique uses a linear interpolation (Lerp), controlled by the measured distance between the water surface and sampled objects beyond the surface, with respect to the screen position (Cloward, 2025). This means it is possible to add gradient-based effects that are only visible based on how close an object is to the surface. This causes objects to “fade into the depths” with the depth colour gradient and allows for foam to appear around objects floating near the surface. The sea foam would also be visible in shallow water, for example, on the shore of an island.

The final feature added to the water surface effects, were surface ripples. On a real ocean, these ripples are often caused by wind, currents and turbulence, leading to a shimmering effect when the light is cast at the right angle. If these ripples were to be implemented through vertex displacement, the wave shader would require significantly more parameters and an incredibly high triangle count to attain the same resolution. The obvious solution to this is to instead apply this effect using surface normal maps. Normal maps are often very computationally efficient, with the major benefit being that they are designed to influence how light reflects onto an object, instantly capturing the desired “shimmer” effect.

Water surface normals

To create the pseudo-randomness needed to convincingly sell the ripples effect, two normal maps, generated from real water surfaces, are imposed on top of one another, and are then set to scroll in opposite directions. At moderate to long distances, this effect is suitable but as reported in user feedback, the effect is very noticeable when viewed closely. This can then become highly distracting. However, the performance benefits and ease of implementation cannot be overlooked, as the time saved was put towards the development of more important features.



Combination of all water surface effects - Foam, refraction, surface noise and depth gradients.

Buoyancy

With the successful implementation of a discrete vertex displacement shader, the next essential feature to build was a system for Buoyancy. The nature of a Gerstner wave shader as a discrete mathematical formula, means that by writing the same displacement formula and passing in the same values, the same result will be returned.

The first step to building this buoyancy system was to set up parameter synchronisation between the wave shader and buoyancy. Since the wave shader operates on the GPU and the buoyancy system is required to operate on the CPU, functions cannot directly be referenced. Fortunately, the Vector4 containing parameter inputs on the wave shader can be read by the CPU, solving the first issue. The second parameter that would need to be synchronised is time.

Since both implementations of time in *ShaderGraph* and *Time.time* are represented as an incrementing float value since the start of the scene runtime, these values should already be synchronised. While this appears to be true, after the simulation has run for an extended time, it is apparent that the time values used by the CPU and GPU are not accurately synced, with the difference between the two slowly increasing over time. The way I attempted to solve this, was by ensuring that the time parameters in both instances are controlled by the same value. This variable was set in the scene manager and updated at the start of every frame. However, this did not solve the problem. While I initially suspected that this issue could be caused by a floating point precision error, Unity documentation confirms that “PC GPUs always use full 32-bit precision” (Unity, 2025b).

Nevertheless, it is clear that the synchronization issues are a result of increasing runtime. The next test to attempt to solve this issue would be to determine a value for time that would allow the formula (that operates around sine and cosine) to seamlessly repeat. However, solving this issue would eventually end up exceeding time constraints.

The initial approach to game object displacement was to apply the full displacement, including horizontal movement to the object. Then, by also aligning the objects “up” direction with the calculated normals, a visually appealing buoyancy system could be created. However, this approach would not be suitable for integration with a movement system, as the horizontal displacement would be applied every frame as the object moved, causing major inconsistencies in speed. Even though a different approach is required, this method would still be relevant for small, stationary objects and would serve as a much computationally cheaper solution.

To solve the lateral movement issue, determining the height due to displacement at a given point is necessary. However, reading the “Y” value of the vertex displacement alone is not appropriate, as that value would correspond to a point that has already been displaced horizontally, and would only be accurate at the highest and lowest points of the wave, where the horizontal displacement (controlled by cosine) is zero.

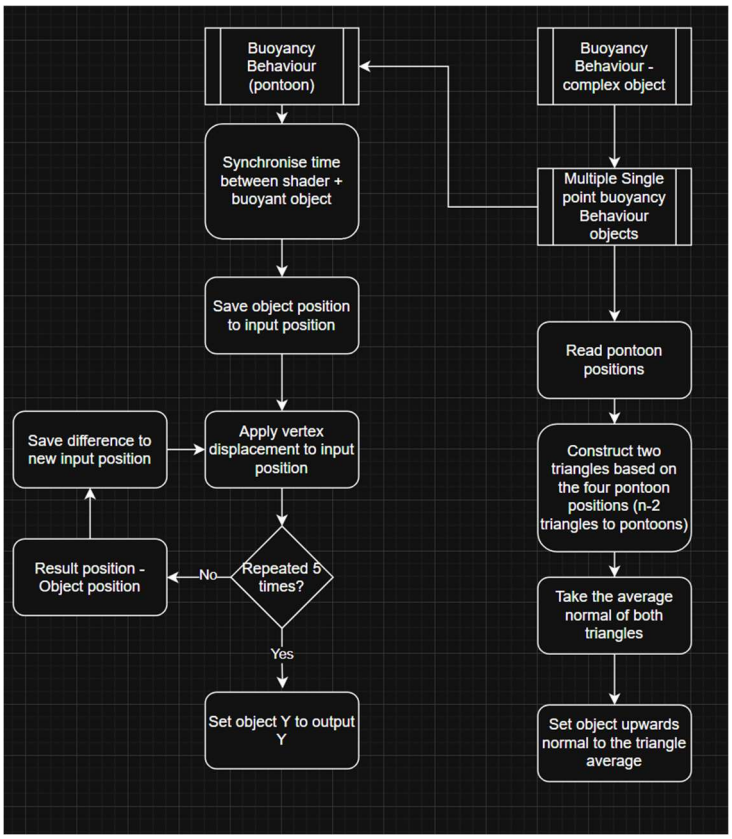
Solving this required an iterative approximation. By determining the original displacement, subtracting the input horizontal position from the output horizontal position, then repeating for at least 4-5 times, the trial-and-error approach would result in a close-enough approximation of the Y position at a given input point.

While again it would be possible to calculate the normals of the given position to determine the rotational behaviour, objects with greater volume would not behave in this manner. Prior research has shown that an object receives a buoyant force, applied at the centre of the submerged volume, proportional to the volume of water displaced. This then causes the object to rotate around its centre of mass. The force of rotation then exerted upon the object is also proportional to the distance between the centre of mass and centre of buoyancy.

To recreate this behaviour entirely would require submerged volume calculations. However, due to the inflexibility of the objects this would be applied to, a height trace can be used to generate a simplified solution.

By taking a minimum of three height traces at fixed positions around an object, a triangle can be constructed between the three points, which can then be used to determine a normal direction, which would then be applied to the rotation of the complex object.

By ensuring that these height traces or “pontoons” are placed near the outer edges of the object, where the strongest rotational forces due to buoyancy would be created, a fair approximation of how a large object would rotate can be made. This precision can be improved by increasing the number of pontoons, then taking the average of all normals generated by the triangles constructed. But the increase in precision does experience diminishing returns for a linear increase in compute times. Because of this, the number of pontoons per complex object in this scenario is set to four, used to construct two triangles.



Flowchart describing the full complex buoyancy system

Movement

The final essential feature of this project is the sailing system.

Without wind, there would be no sailing. While an engineering solution may simulate wind across a virtual sail using complex laws of aerodynamics, such a system would be incredibly computationally expensive.

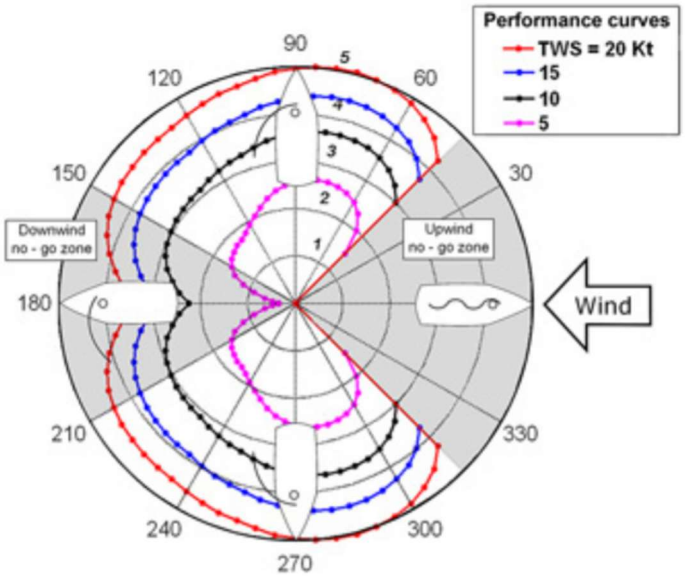
Implementing wind in this project is as simple as setting a wind direction and strength, which can then be referenced by appropriate systems. An equally simple sailing system would then determine the forward force applied to the vessel, scaling with the angle between the incident wind and the sail.

The original plan for sail force would operate on a similar principle, where the combination of all forces acting upon the boat would determine resultant movement. For the style of sail that was implemented, unique behaviours would be necessary for when the user would be sailing into the wind itself as various aerodynamic behaviours such as Bernoulli's principle and the Coanda effect, leveraged alongside apparent wind could cause the boat to travel faster than the wind itself. It was quickly evident that implementing every force that acts on the boat itself would severely overcomplicate the solution. Furthermore, ignoring some of these forces meant that movement would lack defining behaviours.

The solution to this was through the use of Unity's animation curve API (Unity, 2025a) and polar performance curves. A polar performance plot in the context of sailing, is a radial graph that typically maps boat speed to wind speed, depending on the angle of attack. By plotting an animation curve that represents the data applied to the polar plot, it is possible to simply evaluate the curve value at a given AOA to determine the sail force acting upon the boat.

Polar plots are typically constructed by measuring real world data or are generated through highly complex simulations. For the context of this project, not only does it massively simplify the computing required to determine sail force, but it is also a good approximation of the culmination of all forces required to produce real behaviour. Furthermore, animation curves are very easily editable, allowing changes for the sake of user experience to be made quickly.

One improvement to this system that was not implemented in the current artefact would be the interpolation between distinct curves depending on current wind speed and movement speed, as polar plots do vary depending on these parameters.



Polar plot of a similar style of sailing boat (Plumet et al., 2015), used as a reference for existing implementation

For some titles, such as *Sea of Thieves*, the user is given rigid and precise control over the angle of their sails, through either keyboard controls or pulley interaction points. While these solutions benefit an intuitive user experience, this behaviour, especially for a Laser-style dinghy is unrealistic.

Normally, the sail reach is controlled by a “mainsheet”, which only limits how far the sail can swing out. The driving force on the sail itself, ignoring inertial and gravitational effects, is the wind. By determining the relative angle of the wind to the boat, the sail can then rotate such that it will end up pointing into the wind, generating no force. The user controls the mainsheet through a slider, which can then force the sail back into an AOA where lift is generated.

While it may be unintuitive to those unfamiliar with sailing, this implementation enables a more experienced user to perform real manoeuvrers such as a *tack* and a *jibe* wherein changing direction and turning through the no-go zones, the sail will swap sides and begin generating force again.

Testing and evaluation

Feature test checklist, all behaviours listed here are evidenced in the video attached to the report.

Feature	Expected Result	In-engine Behaviour	Pass/Fail
Scene Wind	Wind direction and strength exist and can be determined by the user.	Wind direction can be interpreted through the weathervane UI, and wind strength determines boat speed by AOA.	Pass
Sail Control/Movement	Sail is controlled by a mainsheet, that determines the rotational reach. Given mainsheet “slack”, sail points into wind.	Mainsheet is controlled by a UI slider; sail will rotate into the wind given slack.	Pass
Boat steering	The boat can be steered by controlling a rudder.	A and D keys cause the boat to steer, relative to velocity.	Pass
Boat movement	The boat will move at varying speeds dependent on wind speed and AOA.	The boat will accelerate/decelerate to target speeds depending on AOA and wind speed.	Pass
Ocean simulation	An ocean shader representative of open ocean water, including large waves and small surface turbulence/ripples.	A Gerstner based vertex displacement shader to control larger wave motion alongside a scrolling heightmap to emulate surface ripples.	Pass
Buoyancy	Boat rotates with respect to the ocean waves.	Gerstner wave formula is used to determine buoyant movement. Buoyancy drifts out of sync after extended use time.	Adequate
Lateral Boat Rotation	Boat rotates around the forward axis due to wind, righted by a simulated keel force. Boat also rotates around the upward axis to gently turn into the wind.	Not implemented.	Fail
Performance	Game maintains a minimum framerate of 30 fps on the test machine.	Game framerate does not drop below 60 fps, even after extended runtimes.	Pass

Quantitative data

Using the performance profiler provided by *Unity*, it is necessary to stress test the individual features of the project to evaluate if performance is adequate and scalable.

The built project that was shared for user testing, as well as the project demonstrated in the attached video follow this specification;

One boat (includes one complex buoyant object with 4 simple pontoons), containing ~ 8000 triangles. A 200 by 200-unit plane with full wave shader applied, subdivided to ~22 000 triangles.

On the test machine, this build of the project maintained an FPS (frames per second) count of ~110, or a total frame compute time of ~9.1ms. This value is measured using *Time.unscaledDeltaTime*, which captures the unscaled time between frames operating on the main thread (Unity, 2026d) Multithreading has not been implemented in this project, so this value will capture compute times of all relevant scripts. Unity's profiler documentation also mentions how certain markers on the CPU profile include time spent waiting for GPU processing to finish (Unity, 2026b).This means that the frame compute time measured by *unscaledDeltaTime* also includes any GPU stalling/bottlenecks.

It is important to note that the GPU processing is occurring simultaneously with the CPU, meaning that an increase in CPU or GPU compute time does not necessarily mean a decrease in FPS.

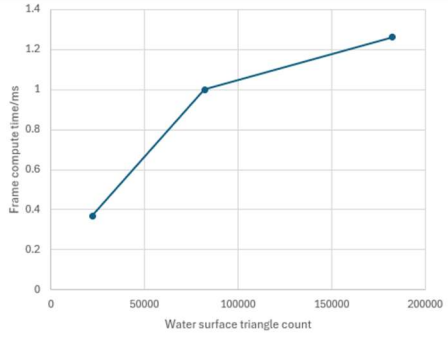
Furthermore, the later mentioned "script compute time" is the total time spent processing scripts on the CPU main thread and is not influenced by any stalling caused by the GPU.

The first values measured were the GPU processing times. Since the wave shader requires a relatively high subdivision count to maintain visual fidelity, this was the chosen variable for measuring GPU performance as it is often the costliest feature of similar titles (as previously discussed when reviewing the *Sea of Thieves* SIGGRAPH talk).

Gerstner shader stress test (water shader only)

Entire water surface is in view – eliminating any occlusion culling effects

Water surface triangle count	Median GPU frame time/ms	Total used memory/GB
0 (control)	2.10	1.77
~22 000	2.47 (+0.37)	2.48 (+0.71)
~82 000	3.10 (+1.00)	2.53 (+0.76)
~182 000	3.36 (+1.26)	2.78 (+1.01)

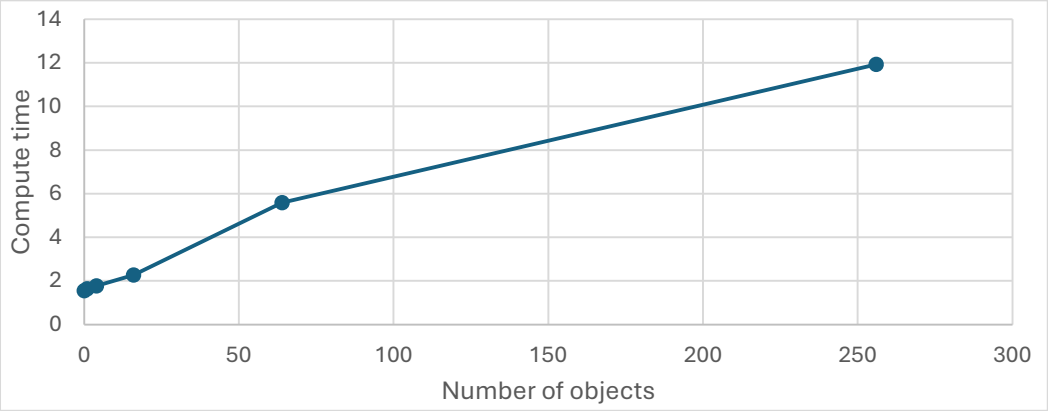


As expected, there is a moderate linear correlation between frame time and the number of triangles in scene. For a full deployment, the ocean would likely require a much larger scale and many more triangles to maintain visual fidelity. This would mean implementing level of detail (LoD) systems wherein performance is maintained by decimating the number of triangles depending on the distance between the observer and mesh, this would maintain visual fidelity as the reduced triangle counts would not be noticeable at extended distances.

The objects used to manage boat buoyancy can universally be applied to any other object that requires floating motion. In a full deployment, these buoyancy controllers could be applied to scene decoration such as buoys, markers or even birds sat atop the water. Therefore, insight can be gained by testing the computational load of increasing numbers of buoyancy controllers.

Buoyancy physics stress test

Number of Buoyant Objects (4 pontoons per object)	Median script compute time/ms
0 (control)	1.54
1	1.63 (+0.09)
4	1.76 (+0.22)
16	2.26 (+0.72)
64	5.58 (+4.04)
256	11.93 (+10.39)



As expected, the strong linear correlation between an increasing number of buoyant objects and compute time would leave a reasonable budget for buoyant objects at around 200 before it would exceed default GPU processing times and begin acting as a bottleneck. (note that an increased number of objects and transforms would also increase GPU processing times as GPU time is required to turn world space transforms into screen space transforms).

One of two easily implementable ways to reduce this bottleneck would be to first disable buoyancy updates for objects sufficiently far in the distance, as the displacement and rotation may be imperceptible. Furthermore, small objects or objects with a perfectly uniform centre of mass may not require a full complex buoyancy implementation and a much more lightweight height trace may be applicable.

Secondly, as no current systems or scripts require buoyancy calculations to be complete prior to execution, this system could leverage benefits of multithreading such that buoyancy calculations are completed in parallel, avoiding stalling of the main thread.

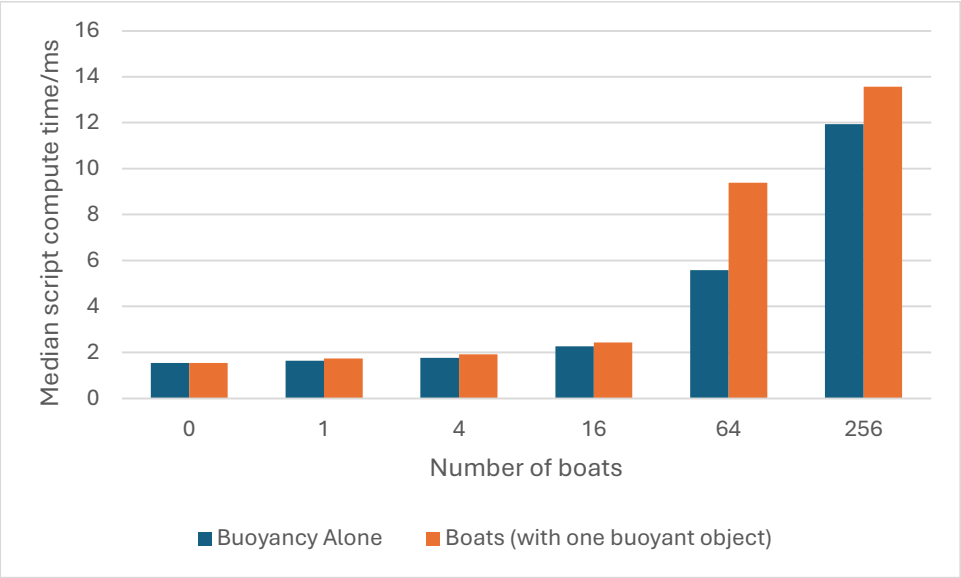
While it may seem obsolete to test the performance of increasing numbers of boat controllers for a game where you simply control only one boat, this data is highly relevant to server-authoritative movement behaviour for a multiplayer implementation. Since it is not recommended to trust positional data sent directly from the client, the server-authoritative model verifies movement by simulating inputs on the server then reconciliating with the client.

This following data shows how a server load may increase depending on an increasing number of players.

Sail physics stress test

Number of Boats (~8000 triangles per boat, 1 buoyant object per boat)	Median script compute time/ms	Difference in compute time, ignoring compute of buoyancy controllers/ms
0 (control)	1.54	0
1	1.74 (+0.20)	(+0.11)
4	1.92 (+0.38)	(+0.16)
16	2.43 (+0.89)	(+0.17)
64	9.38 (+7.84)	(+3.80)
256	13.56 (+12.02)	(+1.63)

It is likely that the performance values measured for 64 boats are outliers.



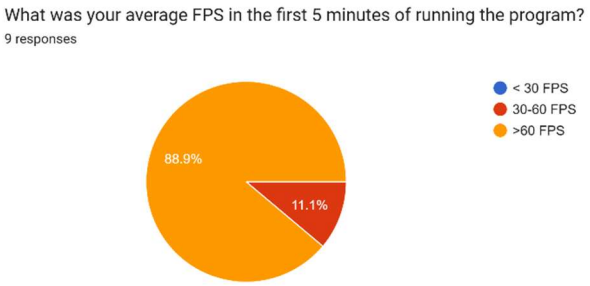
When comparing the results of the boat performance evaluation to that of the buoyancy performance, it is evident that the buoyancy calculation represents a significant proportion of the compute time required. Therefore, when considering optimisations to be made in the context of server performance, the primary target should be reducing or eliminating the buoyancy compute times.

User Performance

For the above data, all values were measured on the same machine. It is important to consider how end-users experience running the artefact, as unique issues can emerge depending varying hardware specifications.

During the feedback survey, data was collected regarding hardware specification, as well as framerates from the initial and late runtimes to determine if any memory leaks existed.

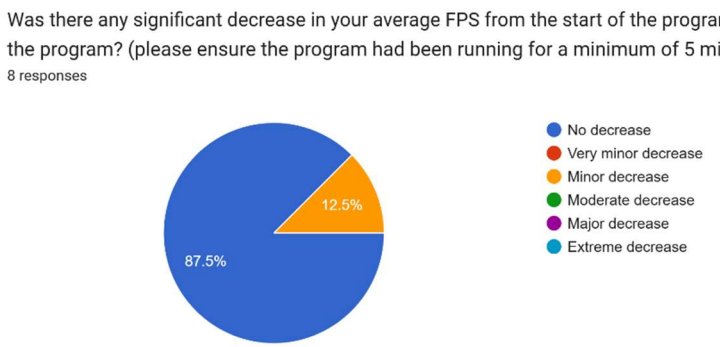
Users were instructed to run the program for a minimum of 5 minutes, and all builds were shipped compiled for *Windows* operating systems.



Between all users, only one user reported a framerate of below 60 FPS. By evaluating the individual response, the user reported running the program with an *11th Gen Intel(R) Core(TM) i5* and integrated graphics. Integrated Graphics CPUs often have significantly less resources available for graphics processing than dedicated GPUs, which may explain the reduction in framerate.

If this project were to be deployed for systems that lack graphics processing power, for example as a Mobile game, an alternative approach to water rendering may be required (such as scrolling noise textures) as this system requires the most GPU resources.

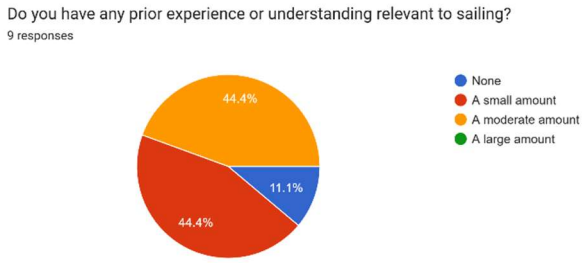
When users were asked if they experienced any decrease in FPS over the duration of the program, there was no observed trend of any significant reduction in FPS, with only one response reporting a minor decrease. This, however, can still be due to a variety of reasons and it is difficult to determine one specific cause.



Qualitative feedback

To evaluate the design and behaviour of features implemented in this artefact, it is important to review qualitative user feedback to justify decisions made.

Prior sailing knowledge may cause users expectations and experiences to vary, so a poll beginning this feedback section was used to gauge how confident users were with their prior knowledge.



In an ideal scenario, varying focus skill groups, such as people from sailing/yacht clubs would have been targeted. Despite this, there is still a moderate distribution in prior skill and knowledge across the set.

Primarily, user responses were positive regarding boat movement matching their expectations, however there was a trend among the “moderate experience” group that picked up on acceleration rates being inaccurate and the small amount of motion while turned head to wind.

While acceleration rates can easily be adjusted to match more realistic standards, the latter is necessary to ensure that less experienced users do not end up stuck. However, it is possible that this extra motion could be an optional setting for more experienced users.

Users did also report that steering behaviour was as expected, with the exception of some of the first testers as in the initial build shipped out, steering was not scaling correctly with *deltaTime* causing excessively fast turning at high framerates. This bug was patched in the subsequent builds that were sent out.

The final feature specifically evaluated in the feedback form was the ocean water shader. Users consistently reported that they felt the water shader behaved realistically, however there were several points made regarding the visual effects. First, the surface noise was frequently reported to be distracting when observed close to the camera but was convincing when observed at a distance. There were also several responses mentioning that the sea foam surrounding the boat was unrealistic and needed a rework to include a wake behind the boat.

While the general user response was positive in reinforcing the original target goals, it is clear there are improvements to be made regarding visual effect aspects of the scene. Furthermore, it was consistently reported that after extended playtimes, the buoyancy simulation drifts out of sync with the wave shader. Unfortunately, I was unable to determine the root cause of this issue.

Self-evaluation and conclusion

The original goals of this project were to develop a player controlled, game-oriented and performative sailing simulation, with a dynamic ocean shader, buoyancy and wind physics.

- Extensive performance profiling and analysis, with a variety of user feedback strongly enforces that the project is suitable for high performance game purposes but may require certain optimisations if the scale of the scene were to be significantly increased.
- User feedback suggests that the ocean simulation, while simple, is convincing but may require changes to surface effects, textures and particles before being considered industry standard.
- While relatively computationally expensive, the implementation of a functioning buoyancy system without the use of physical simulation or collisions is highly effective but will require multithreading and the solving of the time synchronization issues before being industry ready.
- User feedback was also highly positive in regard to the implementation of sail motion, proving that referencing real-world data can save on performance cost and increase consistency, but requires fine design control to find the balance between realism and player experience.
- Unfortunately, the boat heeling feature needed to be removed due to time constraints, but the existing system would allow for it to be implemented without an entire redesign.

The two major limitations of this project were the time available as well as my own skill as a programmer. These limitations completely dictated the scope of all features within the project, leaving various features unsolved. Any subsequent development would ideally allow enough time to eliminate these limitations entirely.

Despite having to drop certain features to complete this project within the given time frame, I believe that the outcome is satisfactory when compared to success criteria and user feedback.

The development process of this project has also been a valuable learning experience, from exploring different methods of generating ocean water and solving for buoyancy, to implementing new tools to assist with project management and design such as *Miro*. Being more effective and comfortable with the use of AI tools is also becoming increasingly more relevant in industry, thus experimenting with them in this project has been a positive use of time.

If this project were to receive further development, the next steps to implement would be as follows;

- Solve all existing issues with the current implementation, being acceleration tuning, buoyancy synchronisation and apparent wind representation.
- Implement audio effects for sails, wind and water to enhance user experience and intuition.
- Implement optimisation techniques for water plane tessellation and LoDs, as well as a LoD toggle system for buoyancy control. These features could also be controlled

through an in-game settings menu, to allow users with a variety of machine specifications to run the project at appealing framerates.

- Improve existing water visual fidelity with better surface textures, foam, wake and spray to make the scene feel more immersive.
- Implement boat heeling and rotation due to wind to improve the representation of wind and enhance realism.
- Allow wind to be variable and implement more visual indicators other than simply a weathervane, to make user interaction more intuitive.
- Expand the scene to be more visually interesting, with floating islands and static buoyant objects.
- Expand the project to be more technically interesting with gameplay features such as time trials or races.
- Explore the implementation of an FFT-based ocean to further improve the visual fidelity of the water.
- Explore the implementation of multiplayer features to allow for a competitive or collaborative experience between players.

Bibliography

Atlassian. (2025a). *Jira* (Version 11.x) [Computer software]. Atlassian. https://www.atlassian.com/software/jira

Atlassian. (2025b). *Trello* (Version 2025.x) [Computer software]. Atlassian. https://trello.com

Cloward, B. (2025). *How to: Create stunning water visuals in Unity*. https://youtu.be/IQ3Zgv0PPNU

draw.io. (2025). *Draw.io* [Computer software]. www.drawio.com

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

Fournier, A., & Reeves, W. (1986). *A Simple Model of Ocean Waves*. Assoc. for Computing Machinery.

Gábor, F. (2025). *Approximate and exact buoyancy calculation for real-time floating simulation of meshes*. EUROGRAPHICS. https://diglib.eg.org/server/api/core/bitstreams/b25beed6-6035-4574-921d-f7c40b88c168/content

Garnelo, M., & Shanahan, M. (2019). Reconciling deep learning with symbolic artificial intelligence: Representing objects and relations. *Current Opinion in Behavioral Sciences*, *29*, 17–23. https://doi.org/10.1016/j.cobeha.2018.12.010

GitHub. (n.d.). *GitHub* [Computer software]. GitHub.

Kozin, V., Cifariello Ciardi, F., Catling, A., & Ang, N. (2018). *The Technical Art of Sea of Thieves*. https://history.siggraph.org/learning/the-technical-art-of-sea-of-thieves-by-ang-catling-ciardi-and-kozin/

Landell-Mills, N. (2020). Sailing into Wind Is Explained by Newtonian Mechanics Based on The Mass-Flow Rate. *European Journal of Applied Physics*, *2*(4). https://doi.org/10.24018/ejphysics.2020.2.4.18

luna tuna. (2020, November 26). *User Review of Sea of Thieves* [Online post]. https://store.steampowered.com/app/1172620/Sea_of_Thieves_2026_Edition/

Microsoft. (2026). *.NET* [Computer software]. Microsoft. https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview

Miro. (2026). *Miro* [Computer software]. https://miro.com/about

Mohazzab, P. (2017). Archimedes' Principle Revisited. *Journal of Applied Mathematics and Physics*, *05*(04), 836–843. https://doi.org/10.4236/jamp.2017.54073

OpenAI. (2026). *ChatGPT* [Computer software]. OpenAI. https://chatgpt.com/overview

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?

Plumet, F., Petres, C., Romero-Ramirez, M.-A., Gas, B., & Ieng, S.-H. (2015). Toward an Autonomous Sailing Boat. *IEEE Journal of Oceanic Engineering*, *40*(2), 397–407. https://doi.org/10.1109/JOE.2014.2321714

Rare. (2018). *Sea of Thieves* [PC]. Microsoft. https://www.seaofthieves.com

SteamDB. (n.d.). *Sea of Thieves Steam charts* [Dataset]. Retrieved https://steamdb.info/app/1172620/charts/#max

Swink, S. (2009). *Game feel: A game designer's guide to virtual sensation*. Morgan Kaufmann Publishers/Elsevier.

Tessendorf, J. (2001). *Simulating Ocean Water*. https://evasion.inrialpes.fr/Membres/Fabrice.Neyret/NaturalScenes/fluids/water/waves/fluids-nuages/waves/Jonathan/articlesCG/simulating-ocean-water-01.pdf

Tyrrel, B. (2018). Sea of Thieves Review. *IGN*. https://www.ign.com/articles/2018/03/28/sea-of-thieves-review

Ubisoft. (2013). *Assassin's Creed IV: Black Flag* [Computer software].

Unity. (2025a). *AnimationCurve*. Unity Technologies. https://docs.unity3d.com/ScriptReference/AnimationCurve.html

Unity. (2025b). *Shader data types and precision*. Unity Technologies. docs.unity.cn/6000.0/Documentation/Manual/SL-DataTypesAndPrecision.html

Unity. (2026a). *Mathf*. Unity Technologies. https://docs.unity3d.com/6000.4/Documentation/ScriptReference/Mathf.html

Unity. (2026b). *Profile markers reference*. Unity Technologies. https://docs.unity3d.com/6000.4/Documentation/Manual/profiler-markers.html

Unity. (2026c). *Shader Graph* [Computer software]. Unity Technologies. https://docs.unity3d.com/Packages/com.unity.shadergraph@17.0/manual/index.html

Unity. (2026d). *Time.unscaledDeltaTime*. Unity Technologies. https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Time-unscaledDeltaTime.html

Unity. (2026e). *Unity Garbage Collector*. Unity Technologies. https://docs.unity3d.com/6000.3/Documentation/Manual/performance-garbage-collector.html

Unity. (2026f). *Universal Render Pipeline*. Unity Technologies. https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@17.6/api/index.html