Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruna Vieira Cruz

Simulating Buoyancy

July 2023

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruna Vieira Cruz

Simulating Buoyancy

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
António José Borba Ramires Fernandes

July 2023

ACKNOWLEDGEMENTS

To begin with, I would like to thank my supervisor António José Borba Ramires Fernandes for proposing the theme of this dissertation and guiding me throughout it.

I would like to thank the University of Minho, more specifically the Department of Informatics for providing the conditions and the professors that allowed me to take this course and the Department of Mathematics for providing me with the necessary conditions to write the dissertation.

Finally, I would like to thank my boyfriend and my friends for all the emotional support they gave me throughout this dissertation and for always being there when I needed it most.

STATEMENT OF INTEGRITY

ABSTRACT

The simulation of object buoyancy is a very interesting topic that involves a lot of research in the area of fluid dynamics.

To better understand how the simulation of floating objects is done, it is necessary to distill the articles found to make a simple simulation that still covers the main characteristics of the simulation.

In this sense, the research required to develop the algorithm is focused on the main dynamic characteristics of the object's movement in the water, that is, the main forces that are applied.

Although this type of simulation is widely used, as in video games, articles that address this topic in a simple way that can be easily adapted in a simulation are quite scarce in terms of the physics involved or focus on how to start and don't develop much further.

In this sense, there are few techniques or methods of implementation, most of which are focused on minimizing the costs of these same techniques and still providing a simulation as realistic as possible.

Faced with this problem, the objective is to have a robust algorithm and be as realistic as possible, being able to demonstrate a good representation of the buoyancy of objects according to their mass and the forces that are exerted. To accomplish this objective, an algorithm was adapted from Kerner (2015) and Kerner (2016).

Each added force contributes to a better representation of reality making the algorithm more realistic. With this accomplished, it is implemented in shaders and tested with multiple objects simultaneously for performance analysis, being possible to observe the capacity of the algorithm.

The simulation made in this dissertation was tested in terms of performance and time spent on GPU, by the number of boats and by the number of triangles in the mesh of the boat.

KEYWORDS Buoyancy, Performance, Scalability, Torque

RESUMO

A simulação de flutuação de objetos é um tema bastante interessante que envolve imensa pesquisa na área de dinâmica de fluidos.

De forma a compreender melhor como é feita a simulação de objetos a flutuar é necessário destilar os artigos encontrados para fazer uma simulação simples que ainda assim abranja as caracteristicas principais da simulação.

Neste sentido, a pesquisa necessária para a elaboração do algoritmo será focada nas principais características dinâmicas do movimento do objeto na água, ou seja, nas principais forças que são aplicadas.

Apesar deste tipo de simulação ser muito utilizado como, por exemplo, em videojogos, os artigos que abordam este tema de forma simples e que podem ser facilmente adaptados numa simulação são bastante escassos em termos de física envolvida ou simplesmente se focam em como começar e não são muito desenvolvidos.

Neste sentido, existem poucas técnicas ou métodos de implementação, sendo a maior parte focada em minimizar os custos dessas mesmas técnicas e mesmo assim proporcionar uma simulação o mais realista possível.

Considerando este problema, o objetivo será ter um algoritmo focado na simplicidade, mas que seja o mais robusto e realista possível, podendo demonstrar uma boa representação da flutuabilidade de objetos consoante a sua massa e as forças que são exercidas. Para cumprir tal objetivo foi adaptado um algoritmo a partir de Kerner (2015) e Kerner (2016).

Cada força adicionada contribui para uma melhor representação da realidade tornando o algoritmo mais realista. Com isto realizado, este será implementando em shaders e testado com vários objetos em simultâneo para uma análise de desempenho, sendo assim possível observar a capacidade do algoritmo implementado.

A simulação feita nesta dissertação foi testada em termos de desempenho e tempo gasto em GPU, tendo em atenção o número de barcos e o número de triângulos na *mesh* do barco.

PALAVRAS-CHAVE Flutuabilidade, Performance, Escalabilidade, Torque

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ACRONYMS

CPU Central Processing Unit. 2, 28, 29

fps Frames per Second. 49, 59

GHz Gigahertz. 49
GLSL OpenGL Shading Language. 49
GPU Graphics Processing Unit. iii, iv, 2, 26, 28, 29, 43, 49, 51, 52, 59

Hz Hertz. 49

ITTC International Towing Tank Conference. 19

kg Kilograms. 49

m Meters. 49

OpenGL Open Graphics Library. 36, 37, 48, 49

SSBO Shader Storage Buffer Object. 28, 37, 43, 44

Part I

INTRODUCTORY MATERIAL

1 INTRODUCTION

The simulation of floating objects is mainly based on the interaction of the boat with the water and vice versa.

A boat is a watercraft of a large range of types and sizes. The type and size vary with their intended purpose. But the question is how they float.

The ability of an object to float depends on the gravitational force and the buoyancy force. If the gravitational force is less than the buoyancy force then the object floats, otherwise sinks. A boat floats because the displaced water weighs more than the boat's weight.

Buoyancy is an upward force exerted by a fluid that opposes the weight of a submerged object. This force, as the Archimedes principle (Ray and Rodriguez, 2018) explains, is equivalent to the weight of the displaced fluid.

In simulations, to calculate all the forces that act in the boat we have to determine the submerged surface and divide all the forces into two groups, the hydrostatic forces, and the hydrodynamic forces.

Hydrostatic forces are those exerted by the water on the object, whereas hydrodynamic forces are the dynamic forces caused by the movement of the object in relation to water.

The main motivation was the development of a model which captures the major dynamic traits of boats in the water, but avoids the complexity and expensive fluid dynamics computation, focusing on obtaining a better performance per boat and constituting a model robust enough to adapt to a large number of boats and different types of water.

1.1 CONTEXT

The simulation of floating things is widely used in video games. Video games have unique constraints when it comes to simulating vehicles, making it necessary and interesting to write about the subject. Often, the dynamics used to simulate boats are not fully understood. The models or theories are too complex or require very expensive simulation methods that are very difficult to control for their application in video games and have a complex adaptation to the needs of both programmers and players.

The solution used for the simulation of object buoyancy, based on Kerner (2015) and Kerner (2016), is focused on performing calculations in a heuristic way with several simplifications along the algorithm in order to avoid complex and expensive calculations in terms of the fluid dynamics involved. In this sense, the main forces that act on the object are calculated in order to provide a simplified model but to still capture the main dynamic characteristics of the object in the water.

When an object is submerged in a fluid, the fluid exerts a force on the surface of the object due to its pressure. In this sense, the force can increase with increasing pressure and/or with increasing depth. The increase in force with depth is very relevant for buoyancy because the buoyancy force is related to an imbalance in the vertical component of hydrostatic forces on the surface of the object.

1.2 OBJECTIVES

The main aim of this dissertation is to implement an algorithm in order to simplify the simulation of a maximum number of boats floating at the same time in the most realistic way possible, depending on the available resources. There is also the aim of monitoring the performance of the algorithm and understanding what can influence the intended realism and what makes the algorithm maintain or not the performance when simulating several boats at the same time.

In order to achieve the aim mentioned, the article cited on Lesek (2016) in which 1000 boats are put to float was consulted. This article concluded it is tough to keep up the performance of an algorithm that puts so many boats to float because the shader receives more data and needs more CPU and GPU resources. Despite that, the algorithm must be robust in dealing with adverse atmospheric conditions, as waves with different amplitudes and frequencies, and maintain performance.

The goal is to implement the simulation in the GPU level in Shaders. This solution was analyzed in terms of performance and time spent on the GPU.

1.3 METHODOLOGY

Since articles referring to the realistic simulation of floating objects are scarce, this dissertation's contribution focuses on distilling the articles found (Ray and Rodriguez, 2018; Bourg and Bywalec, 2013; Batchelor, 2000; Todd, 1957; Williams, 2006; Kerner, 2015, 2016) and showing only what is necessary to get objects to float in the simplest and most realistic way possible. Furthermore, this dissertation presents a GPU implementation of the simulation.

In order to implement the algorithm and to write this dissertation, the articles present in the bibliography were analyzed, being made a general observation of the existing projects related to the simulation of objects' buoyancy and the related fluid dynamics. In this way, it was possible to make the state of the art of this theme, also aiming to find realistic models of boats and to study in more detail the surrounding physics with the interaction of the boat with the water.

The main algorithm can be divided into three initial components:

- Creation of a cutting algorithm to distinguish the triangles that are submerged and those that are not with the objective of determining the forces that act in them. This algorithm has to cut the triangles so that those that are only partially submerged give way to triangles totally submerged or not submerged.

- Calculation of the hydrostatic forces in the fully submerged triangles and their application to the object. These forces are the forces exerted by the water's drive on the object.
- Calculation of the hydrodynamic forces in the fully submerged triangles and their application in the object. These forces are the forces exerted by the movement of the object over water.

With the implementation of the initial algorithm carried out, it was tested with several models to verify if the behavior of the objects fits with reality so that the algorithm can be further improved.

Having already improved the algorithm, the implementation has been tested to the limits of the computer used with several objects simultaneously for performance analysis, this allowed observing the robustness and capacity of the implemented algorithm.

1.4 OVERVIEW

This dissertation is divided into 5 chapters, which are summarized below, excluding the introduction.

- State of the art - this chapter clusters all items found around the floating object simulation. Topics such as discovering the submerged surface of the boat, hydrostatic forces, and hydrodynamic forces are treated.
- Implementation - this chapter details the implementation made based on the information present in the articles found.
- Tests and Results - this chapter shows the tests and results made during implementation.
- Conclusions and Future Work - this chapter presents the conclusions drawn from the completion of the dissertation and the work that can be done in the future to improve the implementation.

2 STATE OF THE ART

There are very few resources that discuss boat physics simulation. Maybe one of the reasons for this scarcity is the excess of simplifications and shortcuts made to run these simulations in a realistic way. This state of the art is mostly based on Kerner (2015) and Kerner (2016) because these articles use a clear and simplified approach that can be adapted into an algorithm that does this type of simulation.

With the objective to make objects float in a mesh of water, the structure of the algorithm can be outlined as shown in Kerner (2015).

The main algorithm can be broadly divided into three parts:

- Determine the submerged surface;
- Calculate and apply the hydrostatic forces;
- Calculate and apply the hydrodynamic forces.

Since we opted for the surface method discussed in Section 2.1 to determine the submerged surface and considering the boat is represented by a triangular mesh, the first part of the algorithm is based on a solution found in Halbert (2016) that takes into account all cases of intersection between the water surface and a triangle and is explained in Section 2.2.

The second part of the algorithm calculates two forces, the buoyancy force which is a linear force and consists of the sum of all buoyancy forces acting on each fully submerged triangle and the torque which is the rotational equivalent of linear force (Section 2.3).

The last part of the algorithm calculates three forces, the viscous water resistance which occurs when water flows across a surface; the pressure drag force is perpendicular to the surface, which is different from the viscous water resistance that instead is tangent to the surface and the slamming force that mainly captures the impact on fluids from sudden accelerations or penetrations, which are almost like rigid collisions (Section 2.4).

This chapter starts by introducing the concept of buoyancy in Section 2.1, determines the submerged surface in Section 2.2, deals with hydrostatic forces in Section 2.3, thus establishing a basis for calculating all other forces involved in the model, deals with the dynamic forces caused by the movement of the boat in the water in Section 2.4 and finally, in Section 2.5, talks about the challenges present in the implementation.

2.1 BUOYANCY CONCEPT

Before discussing the main algorithm, the question of buoyancy is introduced. To compute this force, it is necessary to calculate the magnitude and its point of application in a partially submerged body.

When an object is submerged in a fluid, a force is exerted by the fluid on the surface of the object due to the pressure from the fluid. The greater the pressure, the greater the force exerted. This force exists regardless of having water particles moving in the fluid or if the boat stays still, so it is called a hydrostatic force. Water pressure increases with depth, and a greater depth implies a greater amount of water pressing down with its weight. The pressure at a certain point in the water only depends on its depth as shown in Figure 1.

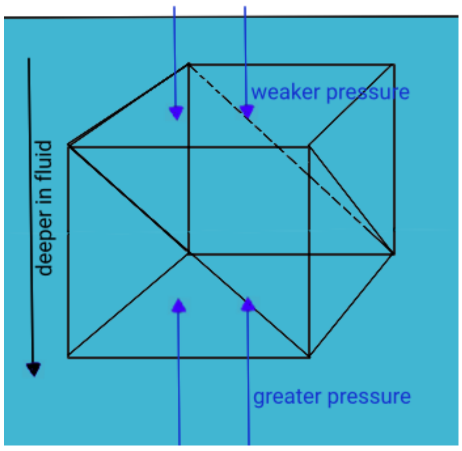


Figure 1: Diagram that describes the pressure applied at points in different depths.

The increasing force with depth is very important for buoyancy, which is calculated from the vertical component of the hydrostatic forces of the body's surface. If we consider the forces as vectors, we have 3 components, which can be divided into 2 horizontal components and 1 vertical component. According to Archimedes' principle (Ray and Rodriguez (2018)), the horizontal components of the water pressures on unit areas of the boat's sides, act in opposite horizontal directions at the same depth. Since the magnitudes of the hydrostatic forces are the same but have opposite directions, the horizontal component of these forces cancels each other out. However, for a generic object, the vertical component of hydrostatic forces does not cancel out, because surfaces where the normal is pointing downwards are at greater depths than surfaces in which the normal is pointing upwards.

The resulting force is pointing in the opposite direction of gravity and its magnitude is equal to the weight of the volume of the submerged part of the body as if it were filled with water. In addition, it is also necessary to find the point of application of the buoyancy force, which is the center of mass of the submerged volume.

There are two methods for calculating the buoyancy forces, the volumetric method and the surface method. The first is to evaluate the submerged volume and determine the centroid, and the second determines the submerged surface and computes the force to be applied on the submerged surface. Both methods should produce the same result when calculating the force applied to the object.

In the volumetric method, it is necessary to close the submerged hull surface to form a closed volume, because by definition volume is the space occupied within the boundaries of an object in three-dimensional space.

Since the shape of the boat can be complex, the volume of the boat can be difficult to calculate. In the volumetric method, as shown in Kerner (2015), it is suggested to use simple volume primitives to approximate the volume of the boat. Using these primitives, it is possible, at least in theory, to make approximations to the submerged volume as precise as floating point operations allow.

In Kerner (2015) is shown an example that uses spheres to calculate the volume of the object, but significant gaps between the spheres can occur as seen in Figure 2. The presence of gaps leads to noticeable deformities in buoyancy.

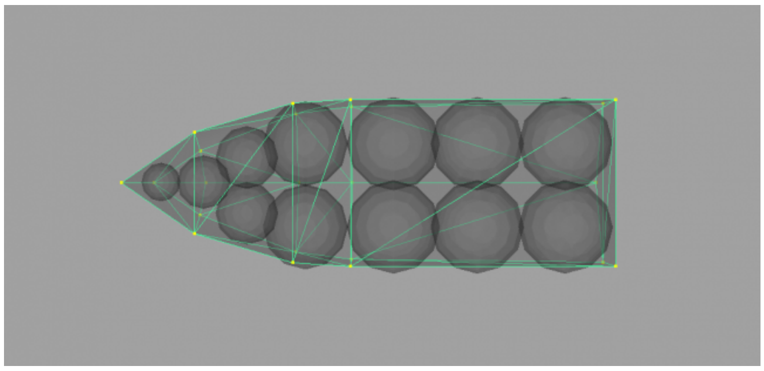


Figure 2: Approximating the volume of a boat with spheres. Source: Kerner (2015).

The volume of the body could also be voxelized. The voxels intercepted by the water could be further voxelized in order to achieve arbitrary precision.

Considering the surface method, assuming the object is modeled by triangles, the hydrostatic pressure forces in each triangle of the submerged surface are calculated and their impulses are added around the body's center of gravity.

Both methods have their pros and cons. The surface method was selected for this work since it has better support in the literature, like Kerner (2015).

2.2 DETERMINING THE SUBMERGED SURFACE

Since the main objective of this section is to calculate the submerged surface, it is assumed that the boat is an approximation based on a triangular mesh of the hull. Within this set of triangles, there are triangles that are fully submerged, triangles that are completely out of the water, and others that intersect the waterline as shown in Figure 3. For the last ones, it is necessary to determine the submerged part of them. For this, we must start by calculating the height at which the vertices of the triangle are located to determine the points of intersection with the water. It is assumed that there is a function $f(x,z)$, where x and z are the horizontal coordinates of the vertex in question and this function returns the height of the water. On a traditional height map, the height map consists of a rectangular area subdivided into bands that form lines and columns, intersecting in square cells as shown in Figure 4.

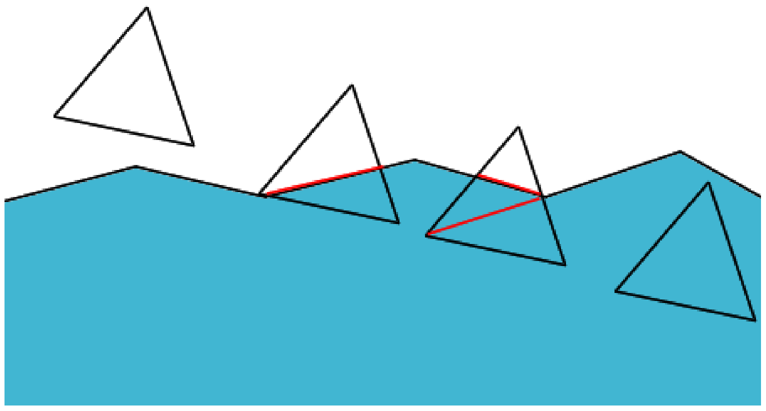


Figure 3: In the figure are shown 4 simplified cases of triangle intersections with the water patch.

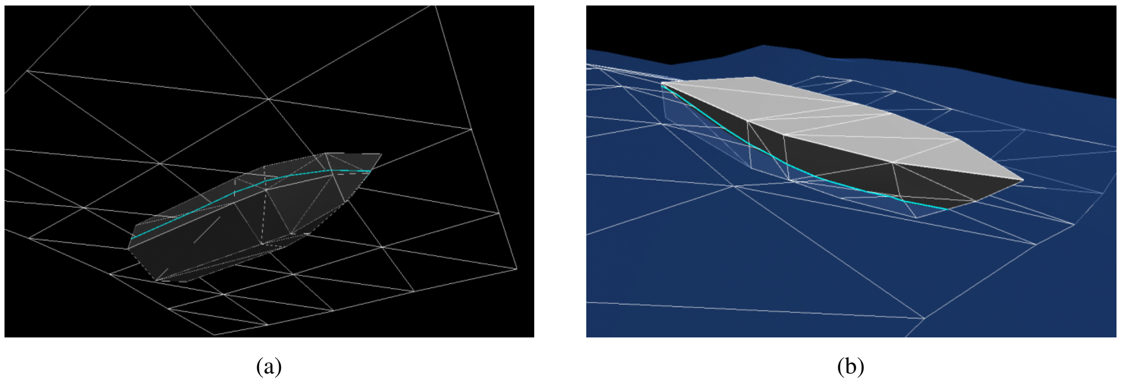


Figure 4: A water patch and the water line (cyan) seen from different perspectives. Source: Kerner (2015).

There are several ways for a surface to divide a triangle. For example, it can cut the triangle in several places; the surface can pass through the center of the triangle without crossing any edges or submerging any vertices. These cases can even be found in relatively calm waters, requiring proper handling in a way that does not cause unrealistic discontinuities in the amount of surface considered submerged. Triangles, where all vertices are above the water, are considered to be entirely out of the water.

If only one or two vertices are submerged, the triangle is cut in a submerged region and in a non-submerged region as shown in Figure 3. If the submerged region is not a triangle, it is necessary to continue triangulating it. It is assumed that the water surface cuts the edge only once between a submerged vertex and a vertex out of the water. With this assumption, in Figure 5 we have some examples of cases not accurately intersected and which are not considered in the algorithm.

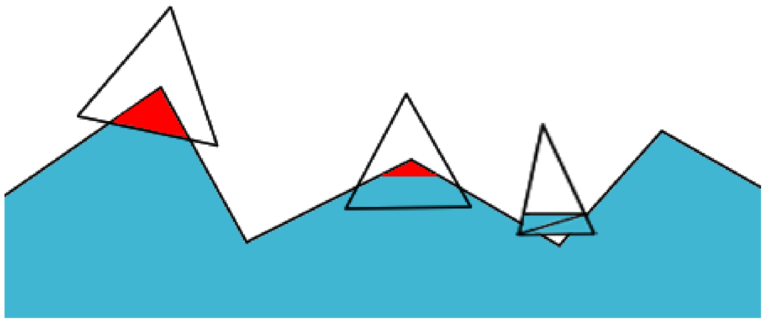
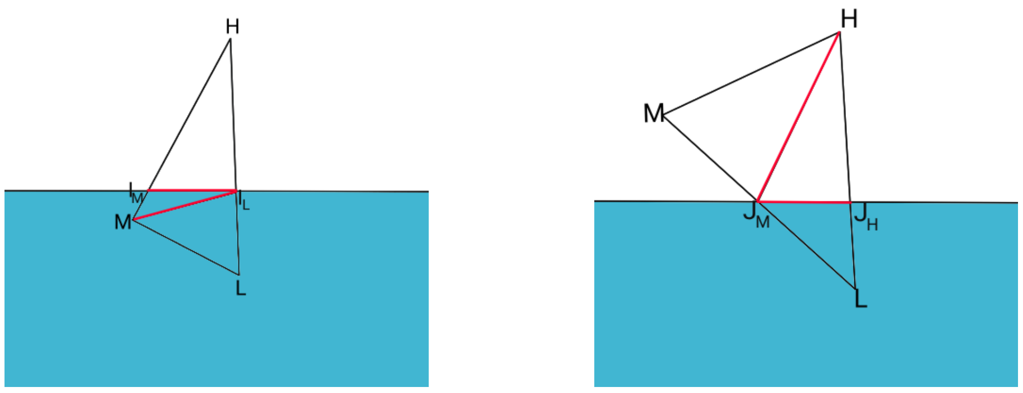


Figure 5: In the figure are shown three examples of cases mishandled by the optimized algorithm. The red areas denote triangles that should have been considered underwater, yet were missed.

To simulate the object buoyancy, it is necessary to determine which triangles are present in the mesh that are submerged and which are not (assuming that the mesh is composed only of triangles) in order to apply correctly the forces exerted.



(a) (b)

Figure 6: Simplified triangle cutting in the possible cases. In Figure 6a is shown a simplified triangle cutting when only one vertex is out of the water and in Figure 6b is shown another triangle cutting when two vertices are out of water.

This algorithm starts by determining the vertices of each triangle that are below the water plane, then rearranges the vertices of the triangle according to the height in relation to the water surface (called H, M and L the highest, mid and lower points). Then it checks how many vertices of a given triangle are submerged, and depending on the case, a cutting strategy for that same triangle is performed. We refer to the heights of these points as h_H, h_M and h_L. We first consider the case where H is above the water but M and L are under (Figure 6a). So $h_L \leq 0$ and $h_M \leq 0$ but $h_H \geq 0$.

We assume that the water cuts the edge HM at a point I_M and the edge HL at a point I_L and we assume that the edge ML is fully submerged. We can parameterize the location of the intersection points by Equation 1 and Equation 2.

$$\overrightarrow{MI_M} = t_M \overrightarrow{MH} \tag{1}$$

$$\overrightarrow{LI_L} = t_L \overrightarrow{LH} \tag{2}$$

We use the heights above water to find a suitable value of t_M and t_L in Equation 3 and Equation 4.

$$t_M = \frac{-h_M}{(h_H - h_M)} \tag{3}$$

$$t_L = \frac{-h_L}{(h_H - h_L)} \tag{4}$$

In the second case, we consider the case where H and M are above the water but L is under (Figure 6b). So $h_H \geq 0$ and $h_M \geq 0$ but $h_L \leq 0$.

We assume that the water cuts the edge HL at a point J_H and the edge ML at a point J_M. We can parameterize the location of the intersection points by Equation 5 and Equation 6.

$$\overrightarrow{LJ_M} = t_M \overrightarrow{LM} \tag{5}$$

$$\overrightarrow{LJ_H} = t_H \overrightarrow{LH} \tag{6}$$

We use the heights above water to find a suitable value of t_M and t_H in Equation 7 and Equation 8.

$$t_M = \frac{-h_L}{(h_M - h_L)} \tag{7}$$

$$t_H = \frac{-h_L}{(h_H - h_L)} \tag{8}$$

2.3 HYDROSTATIC FORCES

From the list of fully submerged triangles, it is possible to calculate the hydrostatic forces. These forces are necessary for the buoyancy force acting on the object. The buoyancy force is defined by the sum of all hydrostatic forces acting on each submerged triangle.

This section is divided into two parts, in Section 2.3.1 the buoyancy force is calculated and applied and in Section 2.3.2 the same process happens to the torque.

2.3.1 *Buoyancy*

The first force to be calculated is the buoyancy force, this force can be calculated by the sum of the vertical component of the hydrostatic force, since the other components cancel each other.

To calculate this linear force applied to a triangle, we need the following values for each triangle:

- Centroid of each triangle;
- Distance between the centroid and the water plane (depth of the triangle's center underwater);
- Normal of each triangle;
- Area of each triangle.

We can calculate the hydrostatic force of each triangle with Equation 9. The buoyancy force is given by the vertical component of the hydrostatic force and is represented by Figure 7.

$$\vec{F_i} = -\rho g h_{center} \vec{n_i} A_{\triangle_i} \tag{9}$$

Where ρ in Equation 9 is the density of the water, g is the gravitational acceleration, h_{center} is the depth of the center of the triangle under the water, $\vec{n_i}$ is the normal of the triangle pointing outwards and $A_{\triangle_i}$ is the triangle's area.

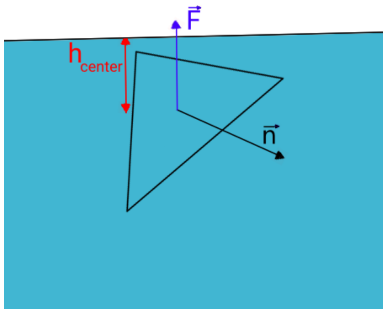


Figure 7: Representative diagram of Equation 9

To calculate the total buoyancy force exerted on the submerged triangles, we compute the resulting force based on Equation 10.

$$\vec{F} = \sum \vec{F_i} \tag{10}$$

Once the buoyancy force has been calculated, it can be applied to the application point i.e., the boat's center of mass, which is assumed to be its geometric center. For the application, the time interval between each frame was calculated and the buoyancy force was added to the weight of the object. Assuming that we have the position and velocity at time t_i, we want to calculate the position and velocity at time t_{i+1}. This can be achieved with Equation 11, Equation 12 and Equation 13.

$$\vec{a} = \frac{1}{m} \times \vec{F} \tag{11}$$

$$t = t_{i+1} - t_i \tag{12}$$

$$\vec{v}_{i+1} = \vec{a}t + \vec{v}_i \tag{13}$$

This velocity is used in the application of Equation 14, which assumes that the initial position is equal to the center of the water plane, and the initial velocity is equal to zero, the next position of the object is calculated.

$$p_{i+1} = p_i + \frac{1}{2}\vec{v}_{i+1}t \tag{14}$$

Finally, a translation is applied to the boat according to vector $\overrightarrow{p_i p_{i+1}}$.

2.3.2 *Torque*

In this section, we present two methods to calculate the point of application for computing the angular force, or torque, and then we proceed to apply it to the boat.

To calculate the torque we can perform Equation 15. For this, it is necessary to calculate the cross product between the vector r_i and the vector $\vec{F}_i$ for each triangle as shown in Figure 8.

$$\tau_i = r_i \times \vec{F}_i \tag{15}$$

Since the vector r_i is the vector between the center of mass of the boat and the point of application of the force, this vector depends on the chosen point of application.

We can apply the torque on the center of the triangle calculated in Section 2.3.2.1 or we can apply the torque on the center of application calculated in Section 2.3.2.2.

2.3.2.1 *Computing the triangle center*

Assuming that the application point is the center of the triangle, we calculate the cross product between the vector r_i (vector between the center of mass of the boat and the center of the triangle) and the vector $\vec{F}_i$ for each triangle as shown in Figure 8.

The vector $\vec{F}_i$ was calculated from the sum of the buoyancy force obtained in Equation 9 based in Kaylegian-Starkey (2022), the weight of the boat as seen in Catto (2020) and the hydrodynamic forces mentioned in Section 2.4, that is, all forces applied on the boat as cited in Bourg and Bywalec (2013).

After calculating the torque of each triangle, it is necessary to add the torques of all the triangles to obtain the torque exerted on the boat, as shown in Equation 16.

$$\tau = \sum \tau_i \tag{16}$$

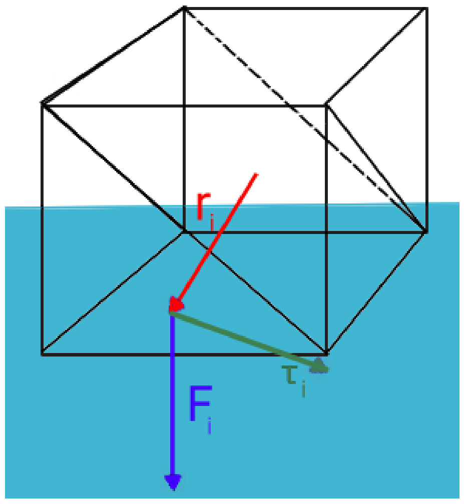


Figure 8: Representative diagram of Equation 15

2.3.2.2 *Computing the application point*

Unlike Section 2.3.2.1, this method does not calculate the center of the triangle, but calculates the application center itself. If the number of triangles is large enough both methods should give the same result, the approximation made above being acceptable.

The formula that finds the point of application of the torque divides each triangle into two triangles whose base is horizontal, with the apex pointing up or pointing down. Thus, two application centers are calculated and, finally, these are averaged. If $H = M$ or $M = L$, the triangle has no contribution to the final torque since the area of this triangle is equal to zero.

To calculate these centers, the vertices of the triangle were reordered according to the highest value on the y-axis (called H, M and L the highest, mid and lower points), then the point of intersection between the line that passes through the point M of the triangle and the midpoint of the vector between H and L is calculated. In Figure 9 we have an example with a horizontal line, but the line orientation can vary.

The intersection point is saved and the new axes for the division of the triangle are calculated. Thus, it is possible to calculate the center of the triangle.

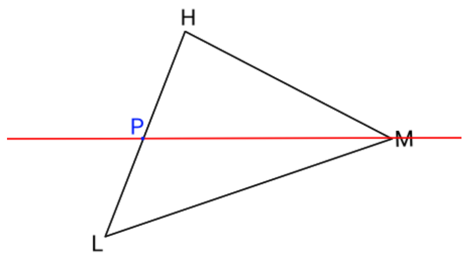


Figure 9: Representative diagram of the intersection between the line that passes through the point M of the triangle and the midpoint of the vector between H and L of the triangle.

One triangle can have the apex facing upwards, we called the center of this triangle C_1 and the center of another triangle that has the apex facing downwards C_2. The calculation of the center of the triangle can be done by Equation 17.

$$C = \begin{pmatrix} 0 \\ y_0 \\ 0 \end{pmatrix} + t_c \begin{pmatrix} (b-a)/2 \\ h \\ c \end{pmatrix} \tag{17}$$

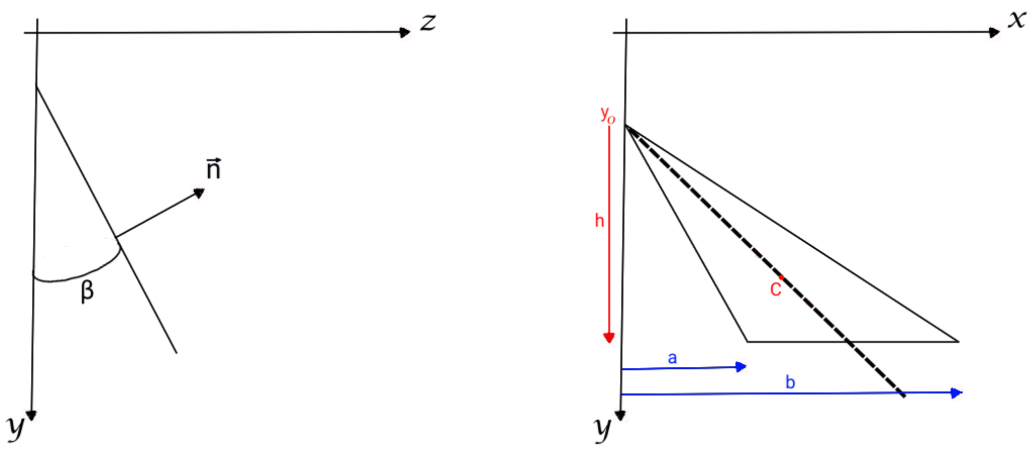


Figure 10: Triangle with a low horizontal edge. The x and z axes are chosen so the triangle normal is in the (z,y) plane. The x-axis is parallel to the base.

The z-axis unit vector is obtained from the cross-product between the new x-axis and y-axis unit vectors. The x-axis is the normalized vector $\overrightarrow{MH}$ if the triangle is pointing downwards or the normalized vector $\overrightarrow{LM}$ if the triangle is pointing upwards, and the y-axis is the vector with the same direction as the gravity vector.

To calculate the centers of the triangles, as shown in Figure 10, it is necessary to know, y_0 which is the depth underwater of the triangle's apex, $b-a$ is the length of the triangle's base, h which is the triangle's height, t_c which is a constant given by Equation 18 or Equation 19, where Equation 18 is for the triangle with the apex facing upwards and Equation 19 for the triangle with the apex facing downwards, and finally c is given by Equation 20 where $\hat{y}$ is the y-axis unit vector shown in Figure 10.

$$t_c = \frac{4y_0 + 3h}{6y_0 + 4h} \tag{18}$$

$$t_c = \frac{2y_0 + h}{6y_0 + 2h} \tag{19}$$

$$c = |\hat{y} \cdot M| \tag{20}$$

The deduction of the formulas in Equation 18 and Equation 19 can be found in the *Appendix A* of Kerner (2015).

Finally, the centers of each triangle are averaged according to the area of each triangle as shown in Equation 21.

$$\tau_i = \frac{area_{up}}{area_i}\tau_{up} + \frac{area_{down}}{area_i}\tau_{down} \tag{21}$$

The torque of each triangle is calculated as Equation 15 where r is the vector between the center of the triangle and the center of mass of the object.

Once the torque is calculated, it is possible to compute the angular acceleration using the inertia matrix. The inertia matrix summarizes all moments of inertia of an object. These moments are not equal unless the object is symmetric about all axes. We can obtain the inertia matrix by calculating manually or we can use programs that calculate this matrix by us.

Having the boat voxelized and based on Williams (2006), the moment of inertia can be calculated for a rigid body of N point masses m_k. These point masses in Equation 22 are calculated from $\frac{m_{total}}{N_{cubes}}$. In Equation 22, r_k (x_k, y_k, z_k) is the vector to the point mass m_k from the point about which the tensor is calculated and δ_{ij} is the Kronecker delta, which translates into the matrix shown in Equation 23.

$$I_{ij} = \sum_{k=1}^{N} m_k(||r_k||^2\delta_{ij} - x_i^{(k)}x_j^{(k)}) \tag{22}$$

$$\begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{N} m_k(y_k^2 + z_k^2) & -\sum_{k=1}^{N} m_k x_k y_k & -\sum_{k=1}^{N} m_k x_k z_k \\ -\sum_{k=1}^{N} m_k x_k y_k & \sum_{k=1}^{N} m_k(x_k^2 + z_k^2) & -\sum_{k=1}^{N} m_k y_k z_k \\ -\sum_{k=1}^{N} m_k x_k z_k & -\sum_{k=1}^{N} m_k y_k z_k & \sum_{k=1}^{N} m_k(x_k^2 + y_k^2) \end{bmatrix} \tag{23}$$

The Inertia Tensor is important to summarize the different moments of Inertia in all axes of the object and should be calculated in relation to the center of mass.

To apply torque to the object, it is necessary to use Equation 25 to calculate the angular velocity of the object in each frame, with the angular acceleration present in Equation 25 given by Equation 24.

$$\vec{\alpha} = I^{-1} \times \tau \tag{24}$$

Where I^{-1}, in Equation 24, is the inverse matrix of the Inertia Tensor and τ is the torque.

$$\vec{\omega} = \vec{\omega}_i + \vec{\alpha} t \tag{25}$$

The angular velocity is used in the application of Equation 26, assuming that the initial angular displacement $\vec{\theta}$ is equal to zero and the initial angular velocity $\vec{\omega}$ is equal to zero, the next angular displacement is calculated.

$$\vec{\theta} = \vec{\theta}_i + \frac{1}{2}\vec{\omega} t \tag{26}$$

Finally, a rotation, based on the matrix in Equation 66 where θ is the rotation angle, is applied to each vertex of each triangle of the object according to the vector $\overrightarrow{\theta_i \theta}$.

2.4 HYDRODYNAMIC FORCES

Complementing the application of hydrostatic forces to the object, it is necessary to calculate the hydrodynamic forces. These forces are calculated for each submerged triangle, added together, and applied to the object.

To help in the calculation of the following forces, the article Nordeus (2020) was used, which is an implementation of the article Kerner (2016) in *Unity*.

Hydrodynamic forces consist of the forces that are necessary to provide the damping of the system, as it happens in reality.

However, it is possible to circumvent these forces and just apply damping based on high velocity in all directions of movement, or in the vertical dimension.

Although not the most accurate, we can get a sense of what can be achieved with the hydrostatic forces as shown in Section 2.4.1.

It is known that if a boat falls from a height, it stops very quickly from swinging vertically. This phenomenon is called damping, the attenuation of movement by forces that oppose it.

In sequence, the maximum height reached by consecutive peaks in this movement becomes smaller and smaller. In the case of critical damping, the damping forces completely prevent oscillations.

In order to obtain greater precision, it is necessary to substitute later the damping forces (Section 2.4.1) for the hydrodynamic forces, that is, the viscous water resistance (Section 2.4.4), the pressure forces (Section 2.4.5), and the slamming forces (Section 2.4.6). Auxiliary calculations for the computation of hydrodynamic forces are made in Section 2.4.2 and Section 2.4.3.

2.4.1 *Damping Forces*

When hydrostatic forces are applied to the object, it is necessary to avoid having it bouncing uncontrollably. A possible solution is to add damping forces, both for linear and angular forces.

Based on the list of submerged triangles, determined in the cutting algorithm, we can obtain from the ratio between the submerged area and the total surface area of the boat, a simplistic measure of how much the boat is submerged, as shown in Equation 27.

$$r_s = \frac{S_{submerged}}{S_{total}} \tag{27}$$

Using r_s, it is possible to create a linear damping force given by Equation 28, and a quadratic damping force given by Equation 29 with a velocity $\vec{v}$ and the damping coefficient given by C_{damp}.

$$\vec{F}_{damp} = -C_{damp} r_s \vec{v} \tag{28}$$

$$\vec{F}_{damp} = -C_{damp} r_s v^2 \frac{\vec{v}}{||\vec{v}||} \tag{29}$$

The formula used was given by Equation 28. However, the formula that is going to be used is not of great importance, as the important thing is to damp a lot, with a constant C_{damp} sufficiently big.

These damping forces are replaced by 3 forces that constitute the hydrodynamic forces (each of them is described in Section 2.4):

- Viscous Water Resistance;
- Pressure Drag;
- Slamming Force.

Each of the hydrodynamic forces is added to the final model, as they are more realistic and precise than the damping forces.

2.4.2 *Drag Equation*

The term "Drag" quoted in detail in Batchelor (2000) means any force that resists movement, but can be caused by different physical phenomena. It can have several causes, such as:

- Friction on the surface of the body in motion;
- Pressure on the surface;
- Dissipating energy in the creation of waves.

The larger the surface in contact with the fluid, the greater the resistance. This is because there is more area for the fluid to interact with the submerged body. In parallel, the greater the density, the greater the mass that exists per unit volume of the fluid.

The drag equation, in fluid dynamics, as shown in Kerner (2016) is given by Equation 30.

$$R = \frac{1}{2}\rho C S V^2 \tag{30}$$

Where R in Equation 30 is the magnitude of the resistance force, ρ is the density of the fluid, C is the resistance coefficient in question, S is the boat area in contact with water and V is the speed of the body.

From the formula in Equation 30, we can conclude that if C were constant, the resistance would be proportional to V^2. This would happen if C was independent of velocity V, which is not, since C is dependent on velocity and other variables.

When talking about the wave resistance of certain boats at moderate speeds, C can change abruptly in relation to V. C becomes a function of V^4, making the power law of the drag equation not quadratic, but a function of V^6. The variation of C is important to understand the nature of a given hydrodynamic resistance.

In Figure 11, the first graph shows us chosen coefficients that may vary or not with the velocity. In the second graph, we observe that the difference between the chosen coefficients represented by green and red produces a very similar Reynolds number.

As suggested in Todd (1957) and shown in Equation 31, it is possible to express C as a function of flow velocity and “fluid path length”, when the Reynolds number R_n referenced in Equation 32 is introduced.

In Equation 31, $constant$ and X are constants dependent on the force to be calculated.

$$C(R_n) = \frac{constant}{(\log_{10} R_n - X)^2} \tag{31}$$

$$R_n = \frac{VL}{\nu} \tag{32}$$

Where V in Equation 32 is the velocity of the body, L is the length the fluid has to travel across the surface (the length of the largest side of the boat) and ν is the viscosity of the fluid (this value can vary with temperature, at 20ºC is 0.000001 and at 30ºC is 0.0000008). The difference between the value of ν to salt water and fresh water is minimal, so an approximation of the value is used. C can be expressed as a function of R_n.

It is noticed that the length L depends on the application and that C is also affected by the length that the fluid must travel, in addition to being affected by the velocity of the body in relation to the fluid.

2.4.3 *Triangle’s velocity for Hydrodynamic forces*

All hydrodynamic forces in the model are calculated from some values and vectors related to the submerged triangles of the boat’s hull.

The boat data structure consists of N_t triangles, a center of gravity G, a linear velocity $\overrightarrow{V_G}$ and an angular velocity $\overrightarrow{\Omega_G}$. Each triangle T_j^b of the boat, $j \in [0, N_t]$, can be submerged, creating one or two triangles T^S.

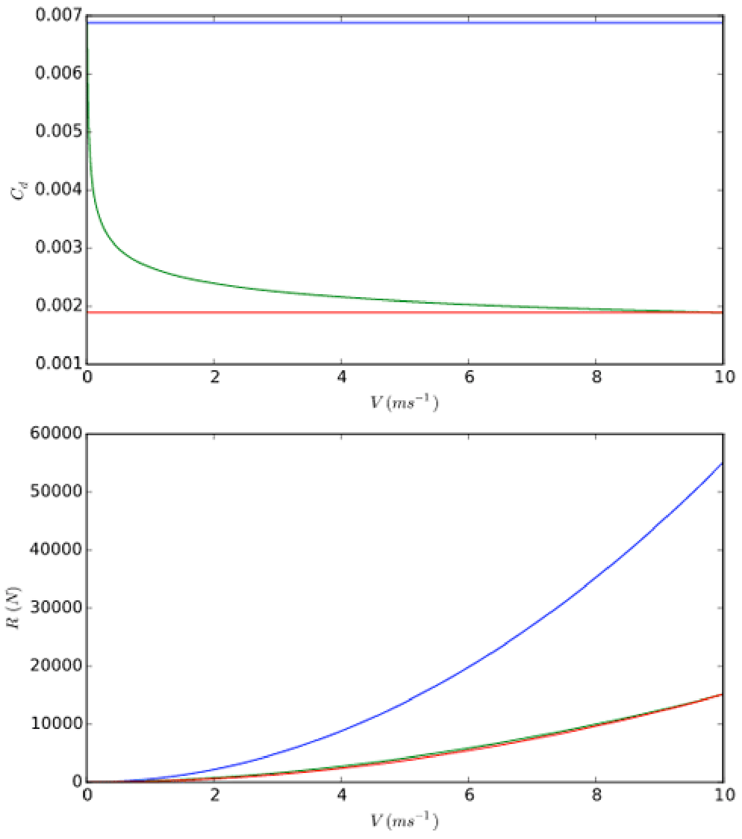


Figure 11: The upper graph represents the drag coefficient as a function of speed V, where $R(N)$ is the magnitude of the resistance force. In the bottom graph, the force is generated as a function of V. Two of the drag coefficients are constant, represented by the colors blue and red. The third coefficient represented by the green color varies with velocity and additionally, the drag force is a function of V^2. Source: Kerner (2016).

The submerged triangle T_i^S has a center of application in C_i and the normal $\vec{n}_i$ pointing outward, its velocity given by $\vec{v}_i$ in Equation 33.

In Equation 33, we can conclude that this velocity is different from the linear velocity of the object because this velocity depends on linear velocity and angular velocity as well as the center of the triangle itself.

$$\vec{v}_i = \overrightarrow{V_G} + \overrightarrow{\Omega_G} \times \overrightarrow{GC_i} \tag{33}$$

The velocity represented in Equation 33 is referred to as the point velocity of the triangle, measured at its center, in global coordinates. The direction of the point's velocity is given by $\vec{u}_i$ in Equation 34.

$$\vec{u}_i = \frac{\vec{v}_i}{||\vec{v}_i||} \tag{34}$$

From Equation 34, it is possible to introduce the scalar product given by Equation 35.

$$\cos\theta_i = \vec{u}_i \cdot \vec{n}_i \tag{35}$$

Since $\vec{n}_i$ points outward, $\cos\theta_i$ is positive if the triangle is pushed towards the water and negative otherwise. Some useful quantities on each submerged triangle are represented in Figure 12.

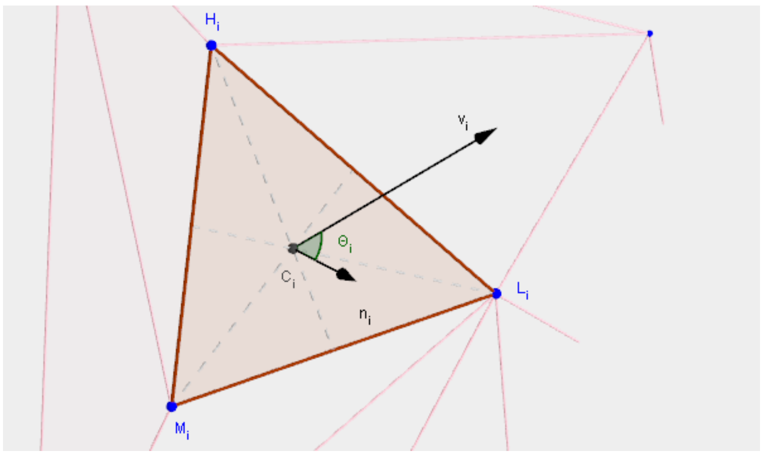


Figure 12: Representation of some quantities on each submerged triangle. Source Kerner (2016).

2.4.4 *Viscous Water Resistance*

Viscous resistance occurs when water flows over a surface. Because there is a strong interaction between the water that is close to the hull and the surface of the hull, there is a layer of water that attaches to the surface and moves with the boat. If the distance between the water and the hull increases, the water adheres less and less.

Each of these layers of water at a certain distance from the surface interacts with the ones immediately before and immediately after, this intensity of interaction is called viscosity. The more viscous a fluid, the more it will stick to the surface and be dragged along with it.

Over the years, several experiments have been carried out to measure resistance to viscous water. One example consisted of a smooth flat plate being dragged underwater. If this plate is thin and dragged along its length, the measured force is basically given by the resistance to viscous water. In 1957, the ITTC agreed that the formula given by Equation 36 was a good approximation of the data collected. This formula is called the ITTC 1957 model-ship correlation line and was not derived analytically from the first principles, but is a regression function that fits the data for practical purposes.

$$C_F(R_n) = \frac{0.075}{(\log_{10} R_n - 2)^2} \tag{36}$$

In the built model, a formula is needed that can be applied to each of the triangles to calculate the viscous friction. Being a complicated thing to do, often developers prefer to resort to measured resistance, rather than analytical formulas.

As the viscosity of the fluid is not the only parameter that influences the size of the force, we have to take into account parameters such as the roughness of the surface itself. If this surface is rougher, it will interact more visibly with a thicker layer of water than a very smooth surface.

It is very important to take into account the flow around the surface. This flow is considered to be laminar in the front of the boat, that is, ordered and the layers at increasing distances from the surface flow parallel to each other with different speeds, due to the influence that the velocity of one layer has on the other, all of this because there is viscosity.

As the time that the fluid flows through a surface increases, the flow is greater and more layers begin to mix. When the different layers mix without rest, the amount of fluid accelerated on the surface increases and gives rise to greater resistance. Therefore, we should expect more resistance to viscous friction at the back of the boat than at the front.

The last thing to take into account when calculating this force is the shape of the boat. Since the boat is not perfectly flat, the fluid must travel a greater distance and the relative velocity of the fluid is generally greater than the velocity of the boat. Pressure changes on the surface of the boat affect resistance due to the shape of the boat.

To circumvent the difficult assessment of the exact influence of all the factors mentioned above, some marine engineers assume that, at a constant low velocity, most of the forces acting on a boat are the result of viscous resistance.

Therefore, the viscous resistance of a reference plate dragged under the water at the same low velocity is measured. Assuming that the reference plate is a flat plate with an area equivalent to the surface of the submerged part of the boat's hull, the resistance of this same plate is multiplied by $1+k$ (a value greater than 1 resulting from all factors that increase the resistance).

Since C_{T0} is the total resistance coefficient measured for a small velocity, and C_{F0} is the viscous resistance of the reference plate in the same *Reynolds number*, k is determined in such a way that Equation 37 make sense.

$$C_{T0} = (1+k)C_{F0} \tag{37}$$

The values $1+k$ determined by the formula vary between 1.22 and 1.65. It is also assumed that the real coefficient of resistance to viscous friction C_{Fr} is multiplied by this same factor $1+k$ across the range of *Reynolds numbers*, from low to high velocity given by Equation 38.

$$C_{Fr}(R_n) = (1+k)C_F(R_n) \tag{38}$$

Where C_{Fr} is given by Equation 38.

In the model in question, we have to apply Equation 38 for the viscous resistance of a flat plate to each of the triangles, thus resulting in the viscous frictional force in the submerged triangle T_i^S given by Equation 39.

$$\vec{F_{vi}} = \frac{1}{2}\rho \frac{0.075}{(\log_{10} R_n - 2)^2} S_i v_{fi} \vec{v_{fi}} = \frac{1}{2}\rho C_F S_i v_{fi} \vec{v_{fi}} \tag{39}$$

Where $\vec{v_{fi}}$ in Equation 39 is the relative flow velocity at the center of the triangle. Since the flow velocity calculated from Equation 40 for a triangle almost facing forward would be very small and with the scalar product close to 1, Equation 40 is used to provide the tangential direction of flow $\vec{u_{fi}}$ (Equation 41) and multiplies by the velocity v_i in Equation 42.

$$\vec{v_{\|i}} = (\vec{v_i} - \vec{v_i} \cdot \vec{n_i}) \tag{40}$$

$$\vec{u_{fi}} = \frac{\vec{v_{\|i}}}{||\vec{v_{\|i}}||} \tag{41}$$

$$\vec{v_{fi}} = v_i \vec{u_{fi}} \tag{42}$$

To obtain a stronger viscous resistance at the rear of the boat, since the flow is greater at this location, a single value of Equation 38 was calculated, but a factor-dependent triangle $1 + k_i$ is included for triangles where the flow is laminar, the k_i is in the range [-1,0]. For triangles located near the rear, k_i is equal to 1 or 2. The integration of all factors $1 + k_i$ results in factor $1 + k$ of the boat given by Equation 43.

$$1 + k = \frac{\sum_i (1 + k_i) S_i}{\sum_i S_i} \tag{43}$$

When a boat is sailing in a straight line through calm waters, viscous resistance is the main force acting on the boat. In this case, the boat has most of the surface out of the water, but with a large part of the bottom touching the water. The pressure resistance would be very small, but significant water resistance is still necessary. If the boat is 2 or 3 times longer without changing its face area, it is necessary for the resistance to increase, and this is possible with the viscous resistance calculated in Equation 39 because it accumulates in the triangles of the hull tangential to velocity.

2.4.5 *Pressure Drag Forces*

As soon as the boat and the surrounding water are observed by Kerner (2016), it was concluded that the energy is dissipated in a way that goes beyond the viscous resistance. As observed in the same article, the waves created by the boat and with the deduction of how much resistance was applied to the boat from the Newtonian reaction and action principle, the resistance of the wave pattern was obtained. However, the slopes of the waves created in this way are considered small, simplifying the calculation and allowing the production of a manageable formula.

Since, near the boat, the slope of the waves cannot be considered small and the waves break, they give rise to a resistance called wave breaking resistance. Thus, depending on the model chosen to capture this resistance, it may be necessary to distinguish yet another resistance that comes into contact with water in a violent way so that the water becomes spray, being quite complicated to be implemented in a simulation.

A solution to simplify the implementation of these forces would be not to capture different components by categorizing and evaluating them, but to solve the *Navier-Stokes* equation numerically. However, it would be a bad idea in terms of the model's performance and complexity.

In this way, a very simple force called pressure drag force was introduced, which captures the necessary resistance. In short, attempts have been made to capture the forces that allow a boat to turn and the sliding forces that push the boat out of the water.

Pressure drag, as opposed to viscous water resistance, acts along the normal surface. This can be divided into a pressure drag for positive $\cos\theta_i$ values and a suction drag for negative $\cos\theta_i$ values. The pressure drag is represented by Equation 44.

$$\vec{F_{Di}} = -(C_{PD1}\frac{v_i}{v_r} + C_{PD2}(\frac{v_i}{v_r})^2)S_i(\cos\theta_i)^{f_p}\vec{n_i} \qquad if\ \cos\theta_i > 0 \tag{44}$$

Where f_p in Equation 44 is the falling power of the opposite scalar product, C_{PD1} and C_{PD2} are the linear and quadratic pressure drag coefficients and v_r is a reference velocity. The reference velocity has the function of facilitating the adjustment of the drag coefficients, the term velocity being equal to 1.

The falling force controls the speed or slowness that the drag drops when the normal transition of the triangle changes from parallel velocity to the velocity of the point when the drag force must reach its peak intensity, and then to one perpendicular to it when the drag strength must disappear. In this sense, the suction drag can be represented by Equation 45.

$$\vec{F_{Di}} = (C_{SD1}\frac{v_i}{v_r} + C_{SD2}(\frac{v_i}{v_r})^2)S_i(\cos\theta_i)^{f_S}\vec{n_i} \qquad if\ \cos\theta_i < 0 \tag{45}$$

Where f_S in Equation 45 is the suction drop power, C_{SD1} and C_{SD2} are linear and quadratic suction drag coefficients and v_r is the same reference velocity. A better turning behavior was observed with values of f_p and f_S less than 1, due to the angle θ_i being close to 90 degrees, the cosine being close to 0, and increasing almost linearly.

It was also observed that it was a very slow process, so a small boat slip angle would not cause enough side drag to slow the boat and help it to turn. On the other hand, in the values of f_P and $f_S \leq 0.5$, the lateral drag increases close to 90 degrees and the boat's turning radius improves a lot.

C_{PD1}, C_{PD2}, C_{SD1} and C_{SD2} are coefficients that can be modified as we please since the formulas are used as a simplified approximation of the real behavior and not in physics.

As the drag force is normal on the surface of the boat, if the boat has sufficient propulsion to fight the force that opposes its acceleration, the hydrodynamic pressure increases at the bottom of the boat, lifting it up. As it increases, the amount of resistance created by friction and drag forces decreases.

2.4.6 *Slamming Forces*

With viscous resistance and resistance to pressure drag, it is already possible to observe a more realistic model of a boat, although the boat can still penetrate the water more than acceptable and the overall response becomes deficient in rigidity. Instead of modifying the drag forces, it was decided in Kerner (2016) to introduce another force that would capture the violence of the fluid's response to sudden accelerations or penetrations.

Collisions that are very rigid are difficult to model using a force dependent on the depth of penetration or displacement when using a discrete integration scheme. As it was important that this force was arbitrarily rigid and the theory of impact forces very complex, the problem was shortened and this is resolved with a perfectly rigid response, where this rigidity was reduced as needed.

The process started with the assessment of the intensity of the collision with the water. If it is too violent, a stop motion force is applied entirely, and the boat stops as if it has hit the ground. Otherwise, that is, if the collision is of low intensity, none of the movement's stopping forces are applied. Between these two cases, several transition functions were combined:

- Linear;
- Power Functions;
- Sigmoid;
- Step.

These forces can be applied to triangles that penetrate violently, rather than being applied to other parts of the boat, to ensure that the rigid response is located where expected. Thus, one can begin by defining the impact force as a force proportional to the area of the triangle in which it is applied.

One of the precautions to be taken is in the case of the most extreme impact, where the sum of the impact forces in all the submerged triangles should be just enough to stop the movement of the boat completely, not exceeding this so that the boat does not jump out of the water. Another challenge is that the force must be present only when there is a change in submersion or velocity, but it must be null if its state is stationary. This means that we must make differences frame by frame to measure the submerged quantity of a part of a boat in the previous stage and compare it with the current stage.

As it is impossible to differentiate frame by frame, due to the different number of submerged triangles in each frame, we can follow the boat's original triangles and calculate for each frame how much they are submerged. To do this, it is necessary to prepare a double buffer of submerged surface areas $[A_j^{submerged}(t - dt), A_j^{submerged}(t)]$.

Considering a triangle of the boat T_j^b. If S^{Total} is the total area of the boat, S_j^b the surface area of the triangle T_j^b, $A_j^{submerged}(t-dt)$ the area of the surface that was submerged in the previous step and $A_j^{submerged}(t)$ the area submerged in the current step, then it is possible to express the volume of water swept by a time interval in the last integration step as shown in Equation 46 and the volume of water swept by a time interval in the current step as shown in Equation 47.

$$dV_j^{swept}(t-dt) = A_j^{submerged}(t-dt)\vec{v}_i(t-dt) \tag{46}$$

$$dV_j^{swept}(t) = A_j^{submerged}(t)\vec{v}_i(t) \tag{47}$$

Taking the difference, dividing by the area of the boat's triangle under consideration and dividing by dt, the equivalent of acceleration is obtained, as expressed in Equation 48.

$$\vec{\Gamma}_j(t) = \frac{V_j^{swept}(t) - V_j^{swept}(t-dt)}{S_j^b dt} \tag{48}$$

If the triangle is completely submerged in both stages of integration, we obtain Equation 49.

$$\vec{\Gamma}_j(t) = \frac{S_j^b\vec{v}_i(t) - S_j^b\vec{v}_i(t-dt)}{S_j^b dt} = \frac{\vec{v}_i(t) - \vec{v}_i(t-dt)}{dt} \tag{49}$$

However, if the triangle is completely submerged in the current integration step, but is completely out of the water in the previous step, we obtain Equation 50.

$$\vec{\Gamma}_j(t) = \frac{S_j^b\vec{v}_i(t) - \vec{0}}{S_j^b dt} = \frac{\vec{v}_i(t)}{dt} \tag{50}$$

Thus, we have an amount that captures the acceleration to which a triangle is subjected when the submerged area does not change, but becomes very large when the triangle submerges quickly. The magnitude Γ_j of the vector calculated in Equation 50 is the perfect candidate for determining whether impact forces should be applied to the triangle or not.

If the triangle moves away from the water, no impact force is applied. That is, when the product rises between the normal of the triangle pointing outward $\vec{n}_i$ and the velocity $\vec{v}_i$ is negative. When this product is positive, a maximum acceleration of Γ_{max} is introduced in which the full motion stop force is applied and below which a power function is used to more or less smoothly increase the impact force, as shown in Equation 51.

$$\vec{F}_j^{slamming} = clamp(\frac{\Gamma_j}{\Gamma_{max}}, 0, 1)^p \cos\theta_j \vec{F}_j^{stopping} \tag{51}$$

Where $\vec{F}_j^{stopping}$ in Equation 51 is the force needed to stop the movement and p is the force used to increase the impact force. A value of 1 causes the impact force to appear gradually. A value of 2 or more, however, does not visibly affect the boat below Γ_{max} and only appears close to that limit, thus obtaining a realistic rigid non-linear stop. One of the things that still needs to be defined is the stop expression $\vec{F}_j^{stopping}$ which is defined by Equation 52.

$$\vec{F}_j^{stopping} = m v_j \frac{2A_j^{submerged}}{S^{total}} \tag{52}$$

The expression in Equation 52 is divided by S^{total} to ensure that the sum of all impact forces does not reverse the movement of the boat. Assuming that all faces on one side reach Γ_{max} and the boat moves without rotation, which means that $\vec{v}_j = \vec{v}$. In this case, we would have the total impact force given by Equation 53.

$$\vec{F}^{slamming} = \sum_{submerged} \cos\theta_j m\vec{v} \frac{2A_j^{submerged}}{S^{total}} = \frac{2\sum A_j^{submerged}\cos\theta_j}{S^{total}} m\vec{v} \tag{53}$$

Where $A_j^{submerged}\cos\theta_j$ in Equation 53 is the surface area of the submerged part of a triangle projected in a plane perpendicular to the velocity $\vec{v}$. Since the boat is a closed surface, the maximum surface area of this projection is half of S_{total}, thus ending with a maximum of $m\vec{v}$. With that, we can conclude that the maximum impact force will only interrupt the boat's movement.

As there are several forces to be applied to the body, the velocity must be measured before the impact force is applied. Any resistive force that depends on the velocity must be calculated and applied after the impact force, as it is necessary to use the velocity measured after the impact force is applied to the body.

The exponent p in Equation 51 affects the dynamic behavior of the boat. The impact forces play an important role because, without these forces, a boat at speed can reach an oscillatory behavior in which it tilts up and down repeatedly, instead of stabilizing with a certain angle of inclination. Therefore, it is possible to adjust the p values balanced by pressure drag forces that can improve or completely eliminate the problem, providing a more realistic and dynamic simulation.

2.5 CHALLENGES

With the implementation of a simulation that supports a large number of boats floating at the same time, several problems arise, among them, the interaction of the boat with the water, the scalability of the algorithm and the stability of the vessel.

One of the challenges that arises is the challenge of avoiding situations where small changes in the height of the vertices introduce large sudden changes in the submerged area. This problem occurs because the triangles below and above the water are not being cut accurately. This problem can cause the body to jump suddenly or sink noticeably, this happens only with the introduction of instability in buoyancy.

The problem of scalability of the algorithm arises with the need to expand the algorithm to support different types of water, from the calmest to the most agitated and with the need to complicate the boat model or even increase the number of boats. For this to happen, it is necessary that the algorithm is scalable so that the movement of the boat model does not become unstable and very oscillating.

In addition to the algorithm not being capable of making the boat unstable, it must also maintain its performance regardless of how its resources are used. For this to happen, the number of boats cannot interfere with the algorithm's operation. That is, if a large number of boats must float, they must be as stable as just one boat floating.

The scalability of the algorithm interacts directly with the stability of the boat, as the more resources are exploited, the more unstable the boat can become, thus ensuring a robust model that covers these types of changes. As such, vertical scalability would be desired, as it adds more resources in order to make the simulation more efficient, thus being able to take advantage of the parallelism, that is, the algorithm to be executed in parallel for each boat that is floating.

With one of the objectives being the implementation of the GPU algorithm, this objective itself emerges as a challenge that deserves to be addressed, since it is required several resources at a computational level so that the vessel is stable and it is possible to simulate a large number of floating boats or complicate the model of the boat.

Another question we can ask is "How to simulate a large number of boats to float without losing performance". This issue is addressed throughout the implementation aiming to have a large number of boats floating in a realistic and stable way.

Therefore, when the question of boat stability arises there may be problems, such as errors in approaches, that when the model of the boat is very simple these errors may not be noticed. With the complication of the boat model, these problems become more and more noticeable, making the model less and less realistic. As the goal is to have a realistic model, it is necessary to avoid these problems as much as possible.

With the stable model, the GPU implementation follows to solve the scalability problem, that is, with the GPU implementation it is expected that performance problems that arise when the model becomes more complex or when a large number of boats floating are resolved.

The stabilization of the boat relies a lot on the correct calculation of the torque. Without this calculation correctly performed, it is impossible to have the boat stabilized, since there are 2 ways to calculate it, but one of these ways generates many approach errors.

Part II

CORE OF THE DISSERTATION

3 IMPLEMENTATION

In order to organize the implementation, a general algorithm was developed which is presented in Section 3.1 and consists of an overview of the steps to be followed in order to implement the article cited in Kerner (2015) and Kerner (2016) and mentioned in the state of the art located in Chapter 2.

As one of the goals is the implementation on GPU for better optimization in terms of performance and starting from an implementation on CPU made for this work, the implementation was passed from CPU to GPU.

The CPU's implementation of the formulas in Kerner (2015) and Kerner (2016) acts as an introduction and test of how forces should be applied to the boat, making it easier to fix problems with applying forces. This implementation is easier to understand and therefore it is immediate to make a bridge to what must be done in the GPU implementation. There are slight differences between them that make the GPU implementation more complicated to understand without a proper bridge with the CPU implementation.

In the CPU implementation was created a method that assigns a function to the height of all the points of a grid. The point with the given height is stored in a map that has a pair with the horizontal coordinates of the point as key and the height of the point as value.

To switch from CPU to GPU, it is necessary that the waves are read as a texture or a height map buffer. Since reading a texture is faster in terms of the GPU than reading a buffer, this was the structure chosen for storing wave heights.

The data of each boat was stored in Shader Storage Buffer Objects (SSBOs). The CPU setup of the SSBOs used throughout the implementation is done and called in the corresponding shader.

The data of each boat is stored in 11 different buffers. The buffer that contains the vertices of each boat has all the resulting triangles of the cut algorithm. The buffers used in the implementation are summarized in Table 1.

Buffer name	Description
vertices	buffer built with a size equal to triple the initial size of the boat. When cutting the triangles, each triangle can give rise to a maximum of 3 triangles, that is, in the worst case, all triangles are cut. This way, the buffer is filled with 9 vertices if the triangle is cut, otherwise, the last 6 vertices are filled with 0's.
colors	contains colors that identify each vertex of each triangle of the boat as a submerged or not submerged vertex. This buffer has the same size as the buffer with the boat's vertices.
forces	contains the linear forces exerted in each triangle. This buffer has the same size as the number of cut triangles of the boat, not the number of cut vertices of the boat.
position	actual position of the current boat
velocity	actual velocity of the current boat
angular_velocity	actual angular velocity of the current boat
angular_position	angular displacement of the current boat
torques	contains the torque exerted in each triangle and has the same size as the buffer that contains the linear forces.
boats_inv_inertia	inverse inertia matrices of all boats
boats_mass	masses of all boats
boats_ATotal	total areas of all boats
boats_length	length of all boats. The length is defined as the largest side of the boat.

Table 1: Description of each buffer used in the implementation.

3.1 GENERAL ALGORITHM

The general algorithm encompasses all the more explicit calculations and algorithms present in this implementation. Each step of the algorithm is explored in a different section, which is divided into 4 sections:

- Section 3.2 - implements the cutting algorithm;
- Section 3.3 - calculates the forces of each triangle;
- Section 3.4 - sum the forces of all triangles;
- Section 3.5 - applies the forces to the boat.

In CPU implementation the algorithm was divided into the steps in Listing 3.1, but in the GPU implementation this division is not efficient and we need to do different shaders that cover the steps in Listing 3.2. This division is better because in shaders is more efficient to calculate all the forces and sum them all together, not being necessary to use atomics to add them one by one.

```
Calculate the cutting algorithm
Calculate hydrostatic forces
```

```
Apply hydrostatic forces
Calculate hydrodynamic forces
Apply hydrodynamic forces
```

Listing 3.1: General algorithm of the CPU implementation.

```
Calculate the cutting algorithm
Calculate all forces of a triangle
Sum the forces of all triangles
Apply the forces to the boat
```

Listing 3.2: General algorithm of the GPU implementation.

To better understand the chaining of all tasks performed during the GPU implementation, we created a flowchart shown in Figure 13.

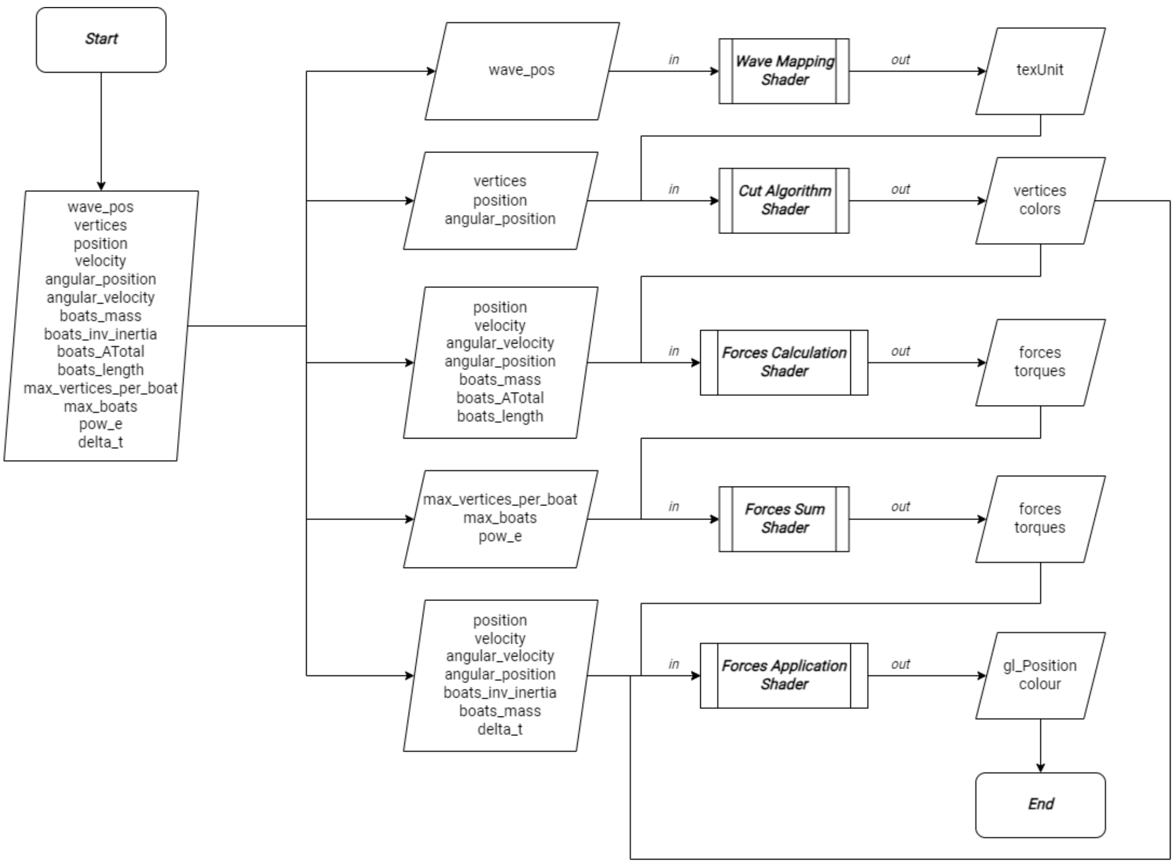


Figure 13: Summarized flowchart of all implemented shaders with in's and out's of each shader.

3.2 CUT ALGORITHM

To start the cut algorithm, it is necessary to verify how many vertices of the triangle are submerged. Depending on the number of vertices submerged, it is necessary to cut the triangle into the smallest triangles that are totally submerged or totally not submerged as shown in Listing 3.3. The linear transformations in the cut algorithm are in Listing B.10 and the distance of a vertex to the water surface, that is the height of a vertex, is calculated in Listing B.1.

```
vec3 A = rotate(boat_theta, (m_model * gl_in[0].gl_Position) + boat_position);
vec3 B = rotate(boat_theta, (m_model * gl_in[1].gl_Position) + boat_position)
vec3 C = rotate(boat_theta, (m_model * gl_in[2].gl_Position) + boat_position)

// verify if the vertice is underwater
int is_A_under = bool(dist_surface(A) < 0)?1:0;
int is_B_under = bool(dist_surface(B) < 0)?1:0;
int is_C_under = bool(dist_surface(C) < 0)?1:0;

switch (is_A_under + is_B_under + is_C_under){
    case 0: {
        // in this case, is not necessary to cut the triangle
    }
        break;
    case 1: {
        // the triangle is cut in 3 smaller triangles where only 1 is totally
            submerged
    }
        break;
    case 2: {
        // the triangle is cut in 3 smaller triangles where only 1 is not totally
             submerged
    }
        break;
    case 3: {
        // in this case, is not necessary to cut the triangle
    }
        break;
    default: break
```

Listing 3.3: Specification of cutting algorithm. Where *m_model* is the model matrix, *boat_theta* is the angle of rotation of the boat and *boat_position* is the actual position of the center of mass of the boat.

In the case where we have only 1 submerged vertex, it is necessary to divide the triangle into 3 smaller triangles as shown in Figure 14. Only 1 of these triangles is submerged and the other 2 triangles are not submerged. This division is done in Listing 3.4.

```
...
    case 1: {
```

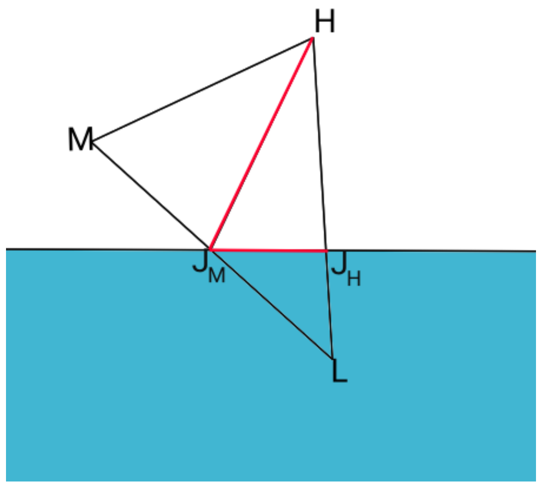


Figure 14: Simplified triangle cutting when only 1 of the vertices is totally submerged. Where J_M is the point where the vector between M and L intersects the water and J_H is the point where the vector between H and L intersects the water.

```
        vec3 L <- verify wich is the lowest vertice
        vec3 M <- verify wich is the medium vertice
        vec3 H <- verify wich is the highest vertice

        float hH <- returns the height of H;
        float hM <- returns the height of M;
        float hL <- returns the height of L;

        float tH = (-hL) / (hH - hL);
        vec3 JH = L + (tH * (H - L));

        float tM = (-hL) / (hM - hL);
        vec3 JM = L + (tM * (M - L));

        vec3 vertexPos = H;
        vertex[idx] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = M;
        vertex[idx+1] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+1] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = JM;
        vertex[idx+2] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+2] = vec4(colour,1.0);
        EmitVertex();

        EndPrimitive();

        vertexPos = JM;
        vertex[idx+3] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+3] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = JH;
        vertex[idx+4] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+4] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = H;
        vertex[idx+5] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+5] = vec4(colour,1.0);
        EmitVertex();

        EndPrimitive();

        vertexPos = JM;
        vertex[idx+6] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+6] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = L;
        vertex[idx+7] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+7] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = JH;
        vertex[idx+8] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+8] = vec4(colour,1.0);
        EmitVertex();

        EndPrimitive();
    }
        break;
```

```
...
```

Listing 3.4: Specification of the case where 1 vertex of the triangle are submerged in cutting algorithm.

In the case where 2 vertices are submerged, it is also necessary to divide the triangle into 3 smaller triangles as shown in Figure 15. Where 2 of these triangles are submerged and only 1 triangle is not submerged. This division is done in Listing 3.5.

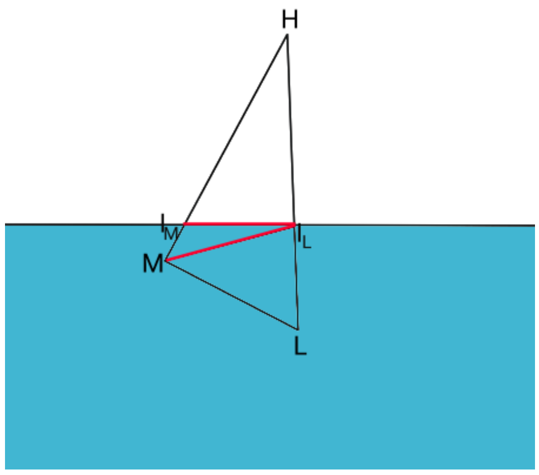


Figure 15: Simplified triangle cutting when 2 of the vertices are totally submerged. Where I_M is the point where the vector between H and M intersects the water and I_L is the point where the vector between H and L intersects the water.

```
...
    case 2: {
        vec3 L <- verify which is the lowest vertice
        vec3 M <- verify which is the medium vertice
        vec3 H <- verify which is the highest vertice

        float hH <- returns the height of H;
        float hM <- returns the height of M;
        float hL <- returns the height of L;

        float tL = (-hL) / (hH - hL);
        vec3 IL = L + (tL * (H - L));

        float tM = (-hM) / (hH - hM);
        vec3 IM = M + (tM * (H - M));

        vec3 vertexPos = H;
        vertex[idx] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = IM;
        vertex[idx+1] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+1] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = IL;
        vertex[idx+2] = vec4(vertexPos,1.0);
        colour = vec3(1.0, 1.0, 1.0);
        color[idx+2] = vec4(colour,1.0);
        EmitVertex();

        EndPrimitive();

        vertexPos = IM;
        vertex[idx+3] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+3] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = M;
        vertex[idx+4] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+4] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = L;
        vertex[idx+5] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+5] = vec4(colour,1.0);
        EmitVertex();

        EndPrimitive();

        vertexPos = IM;
        vertex[idx+6] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+6] = vec4(colour,1.0);
        EmitVertex();

        vertexPos = L;
        vertex[idx+7] = vec4(vertexPos,1.0);
        colour = vec3(0.0, 0.0, 1.0);
        color[idx+7] = vec4(colour,1.0);
```

```
            EmitVertex();

            vertexPos = IL;
            vertex[idx+8] = vec4(vertexPos,1.0);
            colour = vec3(0.0, 0.0, 1.0);
            color[idx+8] = vec4(colour,1.0);
            EmitVertex();

            EndPrimitive();
        }
            break;
...
```

Listing 3.5: Specification of the case where 2 vertices of the triangle are submerged in cutting algorithm.

Considering the cut algorithm presented in Listing 3.3 and Section 2.2, we started by implementing the algorithm for simple objects like cubes, spheres and torus. This algorithm allows the set of triangles in the model to be divided into two smaller sets, which are the set of submerged triangles and the set of non-submerged triangles, as shown in Figure 16. With these sets defined, it is possible to calculate the forces that are exerted on each triangle that are specified in Section 3.3.

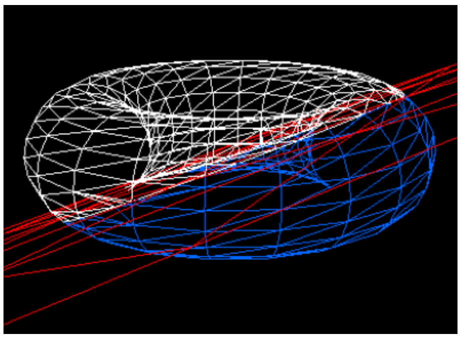


Figure 16: Results obtained from the application of the cutting algorithm. The submerged part of the boat is represented by the color blue and the not submerged part of the boat is represented by the color white.

To implement the cutting algorithm mentioned in Section 2.2, it is necessary to use the texture created to store the wave heights in order to have access to them to cut the object's triangles. The texture integration and the distance to the surface are done as shown in Listing B.1.

Since calculations are performed with points in global space and not local space, it is not necessary to do any additional calculations for model transformations before calculating the distance between each point and the water surface.

When initializing the buffers, it is important to pay attention to the "usage" parameter of the $glBufferData$ function in OpenGL, otherwise, garbage can be written in the buffer. We need to verify the frequency of access (modification and usage) and the nature of that access.

In order to know which boat we are cutting, an index passed from OpenGL to the shader is used. In addition to the SSBOs that store the vertices and their colors, the SSBO that stores the position of each object and the SSBO that stores the current angular displacement of each object are needed, these SSBOs are initialized with the initial position and the null angle, respectively. These variables are used to determine the exact position of each vertex at each frame in global space.

All shader SSBOs that make the cutting algorithm are passed to another shader, the shader that calculates the forces.

3.3 FORCES CALCULATION

The calculation of the forces consists of the calculation of buoyancy force, torque, viscous water resistance, pressure drag force and slamming force for each of the triangles as shown in Listing 3.6. We can divide the calculation of the forces into the hydrostatic forces calculation (Section 3.3.1) and the calculation of the hydrodynamic forces (Section 3.3.2).

When added to the hydrostatic forces, the hydrodynamic forces caused the boat to become unstable. This problem was solved when these forces were also added to the torque calculation.

```
...
layout (std430, binding = 2) buffer forces{
    vec4 force[];
};

layout (std430, binding = 3) buffer position{
    vec4 boat_position;
};

layout (std430, binding = 4) buffer velocity{
    vec4 boat_velocity;
};

layout (std430, binding = 5) buffer angular_velocity{
    vec4 angular_boat_velocity;
};

layout (std430, binding = 6) buffer angular_position{
    vec4 boat_theta;
};

layout (std430, binding = 7) buffer torques{
    vec4 torque[];
};

layout (std430, binding = 9) buffer boats_mass{
  float boat_mass[];
};

layout (std430, binding = 10) buffer boats_ATotal{
  float boat_ATotal[];
};

layout (std430, binding = 11) buffer boats_length{
  float boat_length[];
};

// compute all forces

void main(){
    ...
    triangle = {A,B,C};
    if(color[triangle] == blue_color){
        F = buoyancy_force(triangle) +
             viscous_water_resistance(triangle, resistance_coefficient) +
             pressure_drag_forces(triangle) +
             slamming_force(triangle);
        T = triangle_torque(triangle,boat_position,F);
    }
    force[j] = vec4(F,0);
    torque[j] = vec4(T,0);
}
```

Listing 3.6: Specification of the calculation of the forces of a triangle.

3.3.1 *Hydrostatic Forces Calculation*

As we can see in Section 2.3.1, we start by calculating the buoyancy force based on Equation 54. Where in Listing 3.7, $DENS$ correspond to ρ, that is, the density of the fluid, G is the gravity constant and h_center is the distance between the center of the triangle and the water surface. The distance of a vertex to the water surface is calculated in Listing B.1, the calculation of triangle centroid is shown in Listing B.2, the triangle normal is calculated in Listing B.3 and the triangle area is in Listing B.4.

$$\vec{F}_i = -\rho g h_{center} \vec{n}_i A_{\triangle_i} \tag{54}$$

```
vec3 buoyancy_force(vec3 A, vec3 B, vec3 C){
    float h_center = dist_surface(t_center(A,B,C));
    vec3 F = G * h_center * DENS * area(A,B,C) * normal(A,B,C);
    F.x = 0.0;
    F.z = 0.0;
    return F;
}
```

Listing 3.7: Specification of the calculation of buoyancy force of a triangle.

In order to calculate the torque of a triangle, we use Equation 55. Where in Listing 3.8, r is calculated based on Section 2.3.2.1, that is, the torque is applied to the triangle center and $force\_i$ is the total linear force exerted on the boat, that is, the sum between the buoyancy, gravity, and hydrodynamic forces. The triangle centroid is calculated in Listing B.2.

$$\tau_i = r_i \times \vec{F}_i \tag{55}$$

```
vec3 triangle_torque(vec3 A, vec3 B, vec3 C, vec3 center_of_mass, vec3 force_i) {
    vec3 r = t_center(A, B, C) - center_of_mass;
    return cross(r, force_i);
}
```

Listing 3.8: Specification of the calculation of torque of a triangle.

As soon as the implementation of Section 2.3 to calculate the hydrostatic forces started, the problem of the boat's stabilization was increasingly noticed, being the model very unstable. Kerner (2015) had an error in the calculation of the buoyancy force, where the area of each triangle was not used to calculate this same force applied to the triangle. This directly affects the calculation of the total force, as one of the forces used is weight and when the area of each triangle is not used, the mass of the object is not evenly distributed.

The torque depends on the buoyancy force applied to each triangle to be calculated, as shown in Section 2.3.2.1. The residual torque occurred due to very large approach errors that have accumulated over time.

As one of the goals is to have a realistic model, torque can only occur if there are forces that do not cancel out, which happens when the water has a ripple or, in a simpler case, it is slightly inclined.

In this sense, there are two methods to calculate the center of application of the torque:

- The method in Section 2.3.2.1 which calculates the center of the triangle instead of the center of buoyancy;
- The method in Section 2.3.2.2 that calculates the center of application of the force in the triangle, that is, the point at which the forces cancel each other out.

To carry out the method in Section 2.3.2.2, each triangle of the set of submerged triangles was divided into two distinct triangles. For each triangle, the application center was calculated and an auxiliary function was performed to calculate the force exerted in each of the triangles. Finally, the forces of each of these small triangles were added. This process was done for all triangles of the model, thus making the weighted sum of the centroids of the small triangles calculated in order to obtain the center of application of the torque in the submerged triangle.

For the implementation of torque, both methods were tested. Since the first method is simpler to implement and is more accurate in the area of the object, this one was chosen to avoid approximation errors in the areas required in the calculation of forces.

To calculate the torque, it was necessary to use the center of each triangle to construct the vector between this center and the boat's center of mass. With this vector, the cross product between the total forces applied to the boat and the vector was made, and thus the torque value was obtained.

After applying the hydrostatic forces to the boat it was noticed that the boat became very unstable (shown in Figure 17), in order to avoid approach errors it was introduced a threshold and the damping forces were calculated in order to avoid oscillations in the object and to simulate the interaction of the water with the boat. These forces were later removed and replaced by forces corresponding to the interaction of the water with the boat, that is, the hydrodynamic forces.

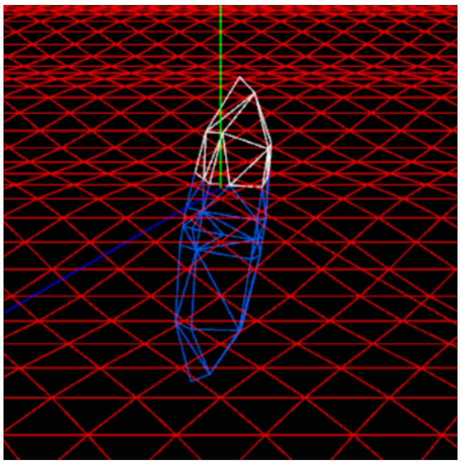


Figure 17: Results obtained from CPU implementation with hydrostatic forces and without threshold. After a few seconds of floating on waves, the boat sinks and becomes very unstable, which did not happen in still water. The submerged part of the boat is represented by the color blue and the not submerged part of the boat is represented by the color white.

3.3.2 *Hydrodynamic Forces Calculation*

With the hydrostatic forces calculated we have to calculate the hydrodynamic forces as shown in Section 2.4. To calculate the first hydrodynamic force, that is, the viscous water resistance of a triangle (Section 2.4.4), we use Equation 56. Where in Listing 3.9, $DENS$ correspond to ρ, Cf is the resistance coefficient, $v\_f\_magnitude$ is the magnitude of $\vec{v_{fi}}$ and v_f correspond to the relative flow velocity at the center of the triangle. The triangle normal is calculated in Listing B.3, the triangle area is shown in Listing B.4, the norm of a vector is shown in Listing B.5, triangle velocity is shown in Listing B.8 and resistance coefficient is calculated in Listing B.9.

In order to calculate the Viscous Water Resistance of each triangle, it is necessary to calculate the Reynolds coefficient mentioned in Equation 36, the area of the triangle to which the force is applied, the relative velocity of the flow at the center of the triangle given by Equation 42 and the magnitude of this same velocity.

$$\vec{F_{vi}} = \frac{1}{2}\rho\frac{0.075}{(\log_{10} R_n - 2)^2} S_i v_{fi} \vec{v_{fi}} = \frac{1}{2}\rho C_F S_i v_{fi} \vec{v_{fi}} \tag{56}$$

```
vec3 viscous_water_resistance(vec3 A, vec3 B, vec3 C, float Cf){
    vec3 n = normal(A, B, C);
    float w_magnitude = norm(triangle_velocity(A, B, C));
    float n_magnitude = norm(n);
    float m = (n_magnitude == 0) ? 1.0 : 1.0/n_magnitude;
    vec3 v_tangent = m * cross(normal(A, B, C), m * cross(triangle_velocity(A, B,
         C), normal(A, B, C)));
    float v_tangent_magnitude = norm(v_tangent);
    float m_vel = (v_tangent_magnitude == 0.0) ? 1.0 : 1.0/v_tangent_magnitude;
    vec3 tangential_dir = m_vel * v_tangent;
    vec3 v_f = w_magnitude * tangential_dir;
    float v_f_magnitude = norm(v_f);
    vec3 viscous_force = 0.5 * DENS * Cf * area(A, B, C) * v_f_magnitude * v_f;
    return viscous_force;
}
```

Listing 3.9: Specification of the calculation of viscous water resistance of a triangle.

As soon as the Viscous Water Resistance has been calculated, we need to calculate the pressure drag forces (Section 2.4.5). The pressure drag forces are calculated depending on the value of the cosine of the angle between the triangle velocity vector and the triangle normal. If this cosine is positive, the pressure coefficients are used, otherwise, the suction coefficients are used.

In this way, it is necessary to calculate the area of each triangle, the normal of each triangle, the cosine of the angle between the triangle velocity vector and the triangle normal, the pressure and suction coefficients and the ratio between the triangle velocity and a reference speed, to obtain the pressure drag forces.

In the calculation of these forces, we have 2 cases of study, where the $\cos\theta$ of the triangle value can result in 2 situations. The first situation happens when $\cos\theta$ is positive and we can use Equation 57 and the second happens when $\cos\theta$ is negative and we can use Equation 58. Where in Listing 3.10, C_PD1 and C_PD2 are the pressure drag coefficients, C_SD1 and C_SD2 are the suction drag coefficients, v is the ratio between triangle velocity and the reference velocity, f_P is the falling power of the opposite scalar product and f_S is the suction drop power. The calculation of the triangle normal is shown in Listing B.3, the triangle area is in Listing B.4, $\cos\theta$ is calculated in Listing B.6 and we can see the calculation of the triangle velocity in Listing B.8.

$$\vec{F_{Di}} = -(C_{PD1}\frac{v_i}{v_r} + C_{PD2}(\frac{v_i}{v_r})^2)S_i(\cos\theta_i)^{f_P}\vec{n_i} \qquad if\ \cos\theta_i > 0 \tag{57}$$

$$\vec{F_{Di}} = (C_{SD1}\frac{v_i}{v_r} + C_{SD2}(\frac{v_i}{v_r})^2)S_i(\cos\theta_i)^{f_S}\vec{n_i} \qquad if\ \cos\theta_i < 0 \tag{58}$$

```
vec3 pressure_drag_forces(vec3 A, vec3 B, vec3 C) {
    float a = pow(triangle_velocity(A, B, C), 2);
    float v = a < 0 ? 0 : sqrt(a);
    float v_ref = v;
    v == 0 ? 1 : v / v_ref;
    vec3 drag_force = { 0.0, 0.0, 0.0 };
    float cos_theta_i = cos_theta(A, B, C);

    if (cos_theta_i > 0) {
        float C_PD1 = 1;
        float C_PD2 = 1;
        float f_P = 0.5;

        drag_force = -(C_PD1 * v + C_PD2 * (v * v)) * area(A, B, C) * pow(
            cos_theta_i, f_P) * normal(A, B, C);
    }
    else {
        float C_SD1 = 1;
        float C_SD2 = 1;
        float f_S = 0.5;

        drag_force = (C_SD1 * v + C_SD2 * (v * v)) * area(A, B, C) * pow(abs(
            cos_theta_i), f_S) * normal(A, B, C);
    }

    return drag_force;
}
```

Listing 3.10: Specification of the calculation of pressure drag forces of a triangle.

The last hydrodynamic forces to be calculated are the slamming forces (Section 2.4.6). To calculate the slamming force it was necessary to calculate the area of the triangle, the cosine of the angle between the triangle velocity vector and the normal of the triangle, the mass of the object, the velocity of the object and the total area of the object.

In order to compute the slamming forces of a triangle we use Equation 59. Where in Listing 3.11, $submerged\_triangle\_area$ is the submerged area of a triangle and cos_theta is the cosine of a triangle with respect to its velocity. The submerged triangle area is calculated in Listing B.7 and $\cos\theta$ is in Listing B.6.

$$\vec{F}^{slamming} = \sum_{submerged} \cos\theta_j m\vec{v}\frac{2A_j^{submerged}}{S^{total}} = \frac{2\sum A_j^{submerged}\cos\theta_j}{S^{total}} m\vec{v} \tag{59}$$

```
vec3 slamming_force(vec3 A, vec3 B, vec3 C){
    return boat_ATotal == 0 ? vec3(0.0) : ((2 * boat_mass *
        submerged_triangle_area(A,B,C) * cos_theta(A, B, C)) / boat_ATotal) *
        boat_velocity;
}
```

Listing 3.11: Specification of the calculation of slamming forces of a triangle.

The main result of the implementation is shown in Figure 18.

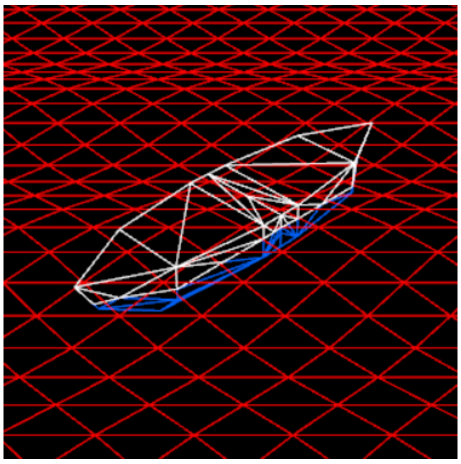


Figure 18: Results obtained from the main implementation with hydrostatic forces and hydrodynamic forces. The addition of hydrodynamic forces caused the model to stabilize. The submerged part of the boat is represented by the color blue and the not submerged part of the boat is represented by the color white.

In order to calculate the forces in the GPU, it was necessary to fill the SSBOs that contain the forces and torque with zeros so that there are no garbage problems that can affect their calculations and then it was created a shader that receives the results of the shader that makes the cutting algorithm, as well as the linear velocity and the angular velocity, the object's mass, the total area of the boat and the length of the boat. This length is determined from the longest side of the boat. The forces are only calculated if the triangle is submerged.

3.4 FORCES SUM

After the forces are calculated and stored in a buffer, it is necessary to calculate the total force of the boat. For this, was used a technique called reduce shown in Listing 3.12.

```
layout (local_size_x = 1024, local_size_y = 1, local_size_z = 1) in;

layout (std430, binding = 2) buffer forces{
    vec4 force[];
};

layout (std430, binding = 7) buffer torques{
    vec4 torque[];
};

uniform int max_vertices_per_boat;
uniform int max_boats;
uniform int pow_e;

int idx = int(gl_GlobalInvocationID.x);

void main(){
    int idx_step = max_vertices_per_boat / pow_e;
    int idx_step_ = idx_step / 2;

    if(idx_step * idx + idx_step_ < max_vertices_per_boat){
        force[idx_step * idx] += force[idx_step * idx + idx_step_];
        force[idx_step * idx + idx_step_] = vec4(0);

        torque[idx_step * idx] += torque[idx_step * idx + idx_step_];
        torque[idx_step * idx + idx_step_] = vec4(0);
    }
}
```

Listing 3.12: Specification of the sum of all forces.

The reduce technique sums forces 2 to 2 in a parallel way along the SSBO, thus reducing the number of buffer elements to be added to half compared to the previous iteration. In this way, the forces buffer needs to have its size equal to a power of 2.

3.5 FORCES APPLICATION

The angular acceleration was calculated to apply the torque to the object. This acceleration was calculated from the object's inertia tensor. The object's inertia tensor was obtained using the *Meshlab* software (Cignoni et al., 2008).

To apply the forces as shown in Listing 3.13, a geometry shader was created that receives 10 SSBOs containing the previously cut vertices, the colors of the vertices, the forces and torques already duly added to the first element of each buffer, the current position, the current velocity, the current angular velocity, the current angular displacement, the mass, and the inertia matrix of the object at the current index.

```
...
layout (std430, binding = 0) buffer vertices{
    vec4 vertex[];
};

layout (std430, binding = 1) buffer colors{
    vec4 color[];
};

layout (std430, binding = 2) buffer forces{
    vec4 force[];
};

layout (std430, binding = 3) buffer position{
    vec4 boat_position;
};

```

```
layout (std430, binding = 4) buffer velocity{
    vec4 boat_velocity;
};

layout (std430, binding = 5) buffer angular_velocity{
    vec4 angular_boat_velocity;
};

layout (std430, binding = 6) buffer angular_position{
    vec4 boat_theta;
};

layout (std430, binding = 7) buffer torques{
    vec4 torque[];
};

layout (std430, binding = 8) buffer boats_inv_inertia{
    mat4 boat_inv_inertia;
};

layout (std430, binding = 9) buffer boats_mass{
    float boat_mass;
};
...
uniform float delta_t;
int idx = (gl_PrimitiveIDIn * 9);
...

// compute the velocity and the position of the boat

// compute the angular velocity and the angular displacement of the boat (angle
    of the boat)

void apply_translate(vec3 A0, ... , vec3 C2){
    vec3 gravity = boat_mass * vec3(0.0, -9.8, 0.0);
    vec3 F = force[0] + gravity;
    vec3 new_v = next_t_velocity(boat_velocity, F, delta_t);
    vec3 new_pos = next_t_position(boat_position, boat_velocity, new_v, delta_t);
    vec3 trans = (new_pos-boat_position);

    A0 = translate(trans, A0);
    ...
    C2 = translate(trans, C2);

    boat_position = vec4(new_pos,1.0);
    boat_velocity = vec4(new_v,0.0);
}
```

```

void apply_rotate(vec3 A0, ... , vec3 C2){
    vec4 new_w = next_a_velocity(angular_boat_velocity, torque[0], delta_t);
    vec3 new_theta = next_a_position(boat_theta, angular_boat_velocity, new_w,
        delta_t);
    vec3 rot = vec3(new_theta-boat_theta);

    A0 = rotate(rot, A0);
    ...
    C2 = rotate(rot, C2);

    angular_boat_velocity = new_w;
    boat_theta = vec4(new_theta,1.0);
}

void main(){
    vec3 A0 = vertex[idx];
    ...
    vec3 C2 = vertex[idx+8];

    vec3 c_A0 = color[idx];
    ...
    vec3 c_C2 = color[idx+8];

    apply_translate(A0, B0, C0, A1, B1, C1, A2, B2, C2);

    apply_rotate(A0, B0, C0, A1, B1, C1, A2, B2, C2);

    gl_Position = m_pv * vec4(A0, 1.0);
    colour = c_A0;
    EmitVertex();
    ...
    gl_Position = m_pv * vec4(C2, 1.0);
    colour = c_C2;
    EmitVertex();

    EndPrimitive();
}
```

Listing 3.13: Specification of the application of all forces.

To apply the forces calculated in Section 3.3, we have to use the equations of linear motion. We can start by calculating the object's velocity from Equation 60 and Equation 61. Where in Listing 3.14, $boat\_mass[boat\_idx]$

is the mass of the boat with the index $boat\_idx$, F is the linear force, v is the current velocity and d_t is the time interval.

$$\vec{a} = \frac{1}{m} \times \vec{F} \tag{60}$$

$$\vec{v} = \vec{a}t + \vec{v}_i \tag{61}$$

```
vec3 next_t_velocity(vec3 v, vec3 F, float d_t){
    vec3 a = (1.0/boat_mass[boat_idx]) * F;
    return v + d_t * a;
}
```

Listing 3.14: Specification of the new velocity of the boat.

With the calculation of the boat velocity, we can calculate the boat's position with Equation 62. Where in Listing 3.15, pos is the actual position, d_t is the time interval, v is the actual velocity and new_v is the new velocity.

$$p = p_i + \frac{1}{2}\vec{v}t \tag{62}$$

```
vec3 next_t_position(vec3 pos, vec3 v, vec3 new_v, float d_t) {
    return pos + 0.5 * d_t * (v + new_v);
}
```

Listing 3.15: Specification of the new position of the boat.

In addition to applying linear forces, it is necessary to apply torque. To apply the torque we have to use the equations of angular motion. We can start by calculating the angular velocity of the boat with Equation 63 and Equation 64. Where in Listing 3.16, $boat\_inv\_inertia[boat\_idx]$ is the inverse inertia matrix of the boat with the index $boat\_idx$, T is the boat's torque, w is the actual angular velocity and d_t is the time interval.

$$\vec{\alpha} = I^{-1} \times \tau \tag{63}$$

$$\vec{\omega} = \vec{\omega}_i + \vec{\alpha}t \tag{64}$$

```
vec4 next_a_velocity(vec3 w, vec4 T, float d_t) {
    vec4 a_angular = boat_inv_inertia[boat_idx] * T;
    vec4 result = vec4(w,0) + d_t * a_angular;
    result.w = 0;
    return result;
}
```

Listing 3.16: Specification of the new angular velocity of the boat.

With the calculation of the boat's angular velocity, we can calculate the boat's angular displacement, that is, the angle of rotation of the boat with Equation 65. Where in Listing 3.17, $theta$ is the actual angular displacement, d_t is the time interval, w is the actual angular velocity and new_w is the new angular velocity.

$$\vec{\theta} = \vec{\theta_i} + \frac{1}{2}\vec{\omega}t \tag{65}$$

```
vec3 next_a_position(vec3 theta, vec3 w, vec3 new_w, float d_t) {
    return theta + 0.5 * d_t * (w + new_w);
}
```

Listing 3.17: Specification of the new angular displacement of the boat.

As a final step, all vertices were translated to a new position and the function that calculated the new position and the new velocity was divided into 2 functions. For memory accesses to be synchronized, it is necessary to use the $glMemoryBarrier$ function in OpenGL in each access made.

4

TESTS AND RESULTS

The tests and results that are performed throughout this chapter are made in *Microsoft Visual Studio 2019* using OpenGL, C++ and GLSL 4.6 as programming languages. The models used in the tests were made in *Blender* and their inertia was given by *Meshlab*. The computer used has an Intel Core i7-10750H 2.60GHz processor, an NVIDIA GeForce GTX 1660 Ti graphics card and a refresh rate of 144Hz.

4.1 TESTS

In this section, tests are presented in terms of performance and GPU time referring to the implementation of buoyancy in different boats.

The model for this implementation can be rendered in 2 different ways, instantiated and non-instantiated.

To render the model in an instantiated way it was necessary to use a variable called $gl\_InstanceID$ in a vertex shader to know the current instance and each information type of the boats was stored in one buffer per type, for example, all vertices of all boats were stored in one buffer.

In a non-instantiated way, the models were rendered in an iterative cycle, where the current index was passed to the shader as the current object index and all main buffers only contain the information of one object. The index here only serves to traverse the buffers that contain auxiliary variables for the calculations, such as mass, inertia, total area, boat length, actual velocity, actual position of the center of mass, angular velocity and angular displacement.

The first tests were done with boats of 7 kg of mass and approximately 12 m^3 of volume with 64 triangles each rendered without instances, that is, in an iterative way and each buffer has only the information of one boat.

The fps and the time spent on the GPU were measured from the rendering of the waves until the moment when the boats are rendered with the respective forces applied. First, the heights of the waves are saved in a texture and then passed to a shader that renders them on the screen. Then, the cycle that renders the boats start and within this cycle, the cutting algorithm is made, the force buffers are initiated with 0's, the forces are stored and summed with the reduce technique, the forces are applied to the current boat and the next iteration of the cycle is executed.

The results of the non-instantiated rendering are shown in Table 2.

The tests performed in addition to considering the performance in fps, concern the execution time of the entire algorithm in GPU in the two models rendering cases.

Number of boats	Performance (fps)	Time spent on the GPU (ms)
1	143.88(±0.062)	0.9994(±0.013)
50	143.622(±0.47)	4.86(±0.089)
100	91.6(±1.67)	9.88(±0.13)
200	48.38(±0.24)	19.38(±0.31)
400	25.5(±0.1)	38.66(±0.51)
800	12.98(±0.045)	76.1(±0.5)

Table 2: Performance analysis of boats and time spent on the GPU in a non-instantiated way.

Tests were carried out with the same boats as in the tests shown in Table 2, but with instances. The structure of the buffers started to contain the information of all the instantiated boats, with these buffers having a variable size depending on the number of boats to be rendered. The results are shown in Table 3.

Number of boats	Performance (fps)	Time spent on the GPU (ms)
1	143.99(±0.074)	1.4(±0.086)
50	93.4(±3.65)	8(±2.35)
100	66.4(±1.52)	15.8(±2.39)
200	34.78(±1.81)	28.6(±1.52)
400	17.6(±1.14)	53.2(±5.26)
800	8.8(±0.45)	126.8(±7.95)

Table 3: Performance analysis of boats and time spent on the GPU in an instantiated way.

In the graph shown in Figure 19, it can be seen, marked in blue, that a better performance is obtained with boats rendered without instances and in orange, it is observed that the performance is worse with boats rendered with instances.

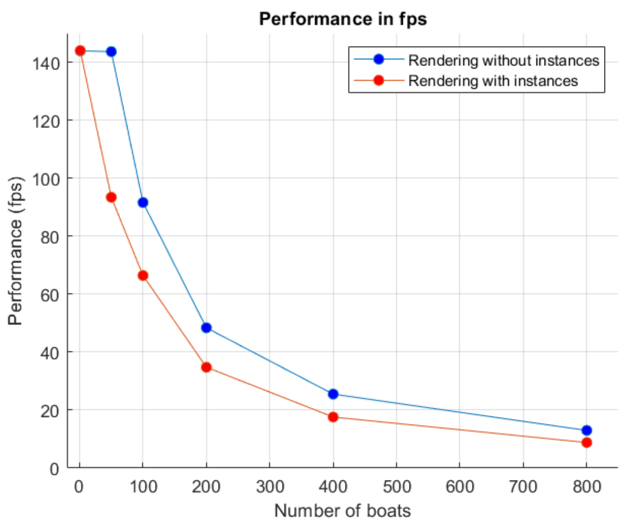


Figure 19: Relationship between increasing the number of boats rendered and performance.

In the graph shown in Figure 20, it can be seen, marked in blue, that less time is spent on GPU with rendered boats without instances and in orange, it is observed that more time is spent on GPU with rendered boats with instances. Furthermore, we can see that the non-instantiated version is linear with the number of boats as expected.

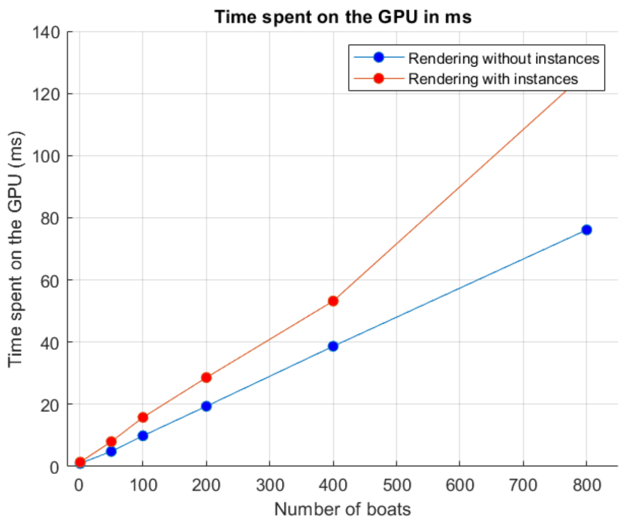


Figure 20: Relationship between increasing the number of boats rendered and time spent on the GPU.

In order to test the performance with the increasing complexity of the boats, the same tests as in Table 2 were carried out, that is, 5 measurements of the performance and time spent on the GPU were made and the mean and standard deviation were calculated for 4 boats with a different number of triangles. The boats as 64, 256, 576, and 7744 triangles, respectively. The results can be seen in Table 4.

		Number of triangles			
Number of boats	**Measurement units**	***64***	***256***	***576***	***7744***
1	**fps**	143.88(±0.062)	143.962(±0.063)	143.99	114.6(±3.44)
	ms	0.9994(±0.013)	0.904(±0.0089)	1.01(±0.014)	8.46(±1.46)
50	**fps**	143.622(±0.47)	143.934(±0.0767)	138.326(±0.217)	53.026(±0.21)
	ms	4.86(±0.089)	5.434(±0.012)	6.486(±0.0219)	17.074(±0.128)
100	**fps**	91.6(±1.67)	86.9(±0.312)	72.812(±0.158)	27.578(±0.044)
	ms	9.88(±0.13)	10.862(±0.102)	13.124(±0.074)	34.772(±0.07)
200	**fps**	48.38(±0.24)	44.506(±0.113)	37.388(±0.063)	14.074(±0.049)
	ms	19.38(±0.31)	21.724(±0.105)	26.118(±0.15)	69.468(±0.158)
400	**fps**	25.5(±0.1)	22.624(±0.0709)	18.948(±0.046)	7.134(±0.022)
	ms	38.66(±0.51)	43.718(±0.222)	52.194(±0.44)	138.566(±0.317)
800	**fps**	12.98(±0.045)	11.424(±0.054)	9.588(±0.043)	3.566(±0.017)
	ms	76.1(±0.5)	86.872(±0.46)	103.624(±0.24)	278.834(±0.505)

Table 4: Performance and time spent on the GPU analysis of different boats in the same conditions.

In the graph shown in Figure 21, it can be seen that the complexity of the boat does not influence the performance of the algorithm when the number of triangles is very low, but the performance decreases as the number of triangles increases a lot. It can also be observed that the performance of the algorithm decreases a lot as the number of boats rendered increases.

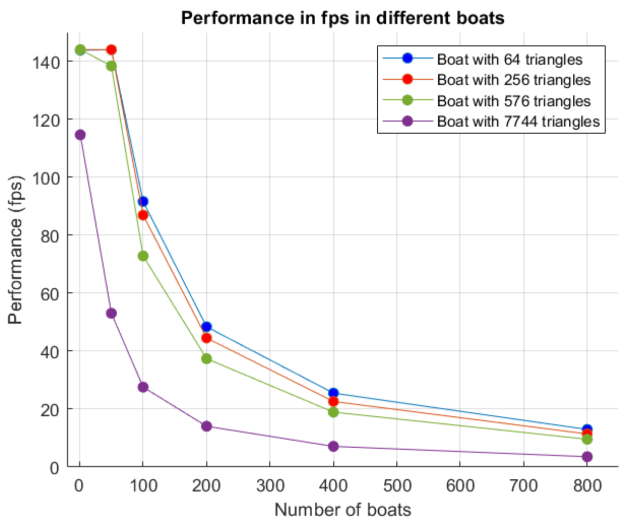


Figure 21: Relationship between increasing the number of boats rendered and their complexity and performance obtained.

In the graph shown in Figure 22, it can be seen that the complexity of the boat has little influence on the time spent on GPU when the number of triangles is very low, but the time spent on GPU increases a lot when the number of triangles also increases a lot. It can also be observed that the time spent on the GPU is more influenced by the number of boats rendered.

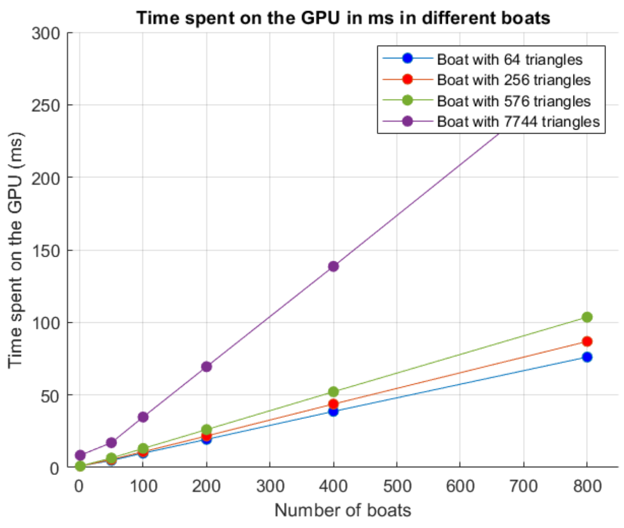


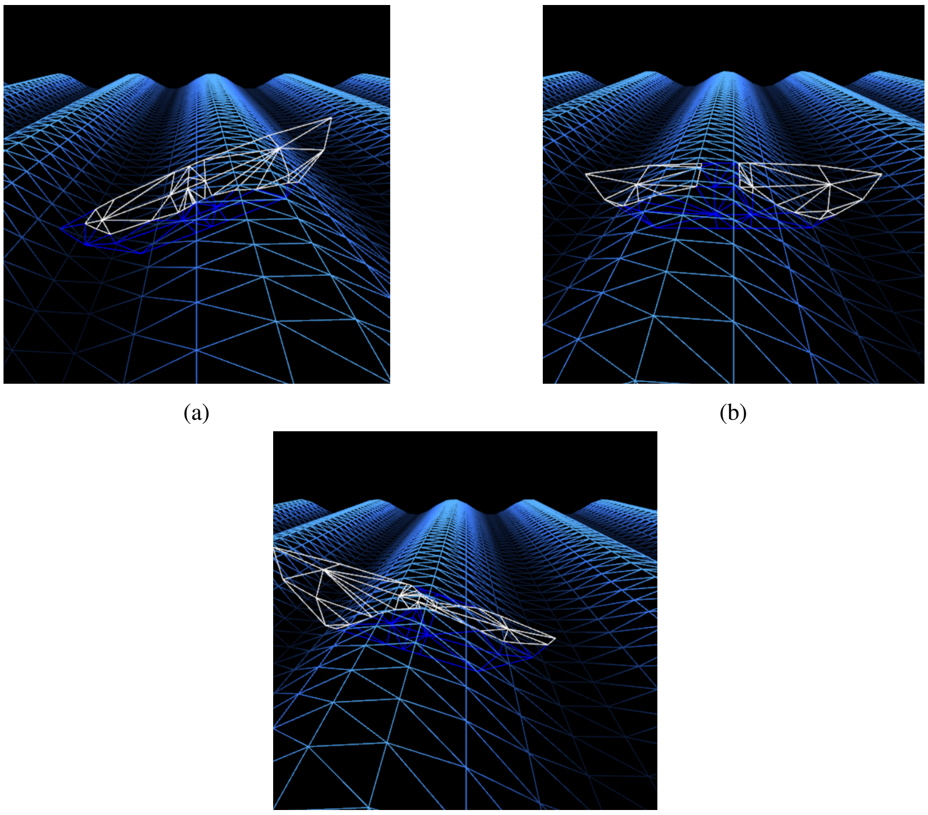
Figure 22: Relationship between increasing the number of boats rendered and their complexity and time spent on the GPU.

4.2 VISUAL RESULTS

In this section, the results regarding the implementation of buoyancy in different boats are presented.

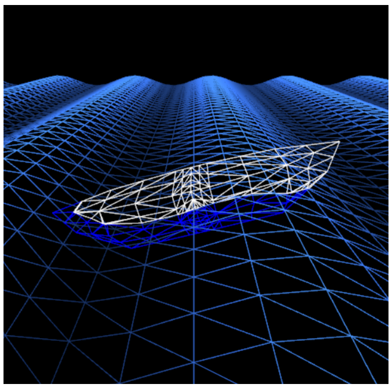
The results of the implemented algorithm can be seen in Figure 23, Figure 24, Figure 25, Figure 26 and Figure 27 in different positions, tested in gerstner waves. The boats that were tested have 64, 256, 576, 2800 and 7744 triangles, respectively.

With the implementation of the algorithm in Chapter 3 we can observe that the model is stable and can go up and down a wave depending on the forces that interact with it. The results show 4 boat models rendered smoothly in a non-instantiated way.

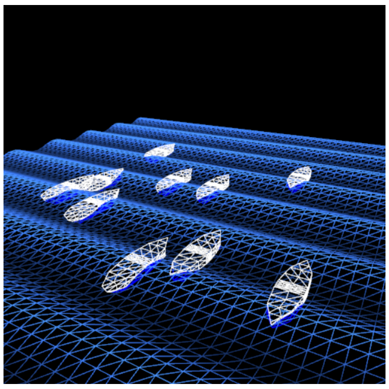


(c)

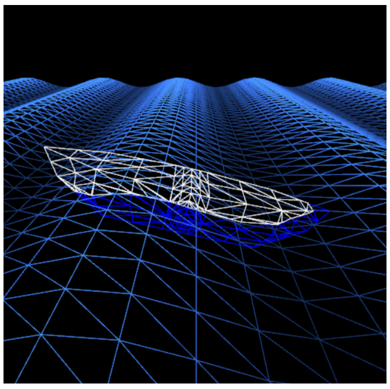
Figure 23: Results obtained from a boat with 64 triangles in different positions of the wave with amplitude and frequency equal to 0.8.



(a)

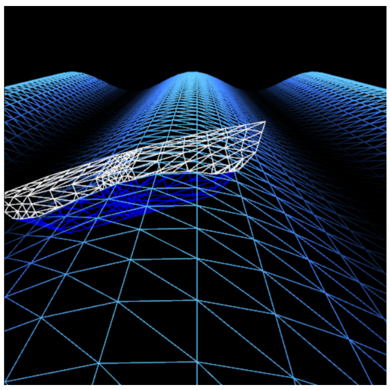


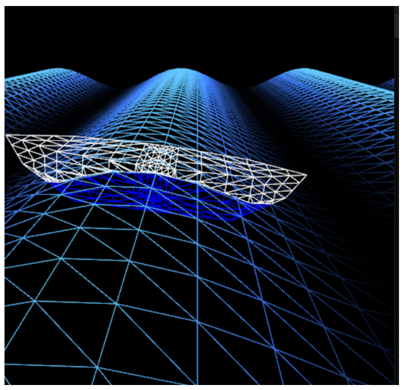
(b)



(c)

Figure 24: Results obtained from a boat with 256 triangles in different positions of the wave with amplitude equal to 0.5 and frequency equal to 0.8.


(a)


(b)

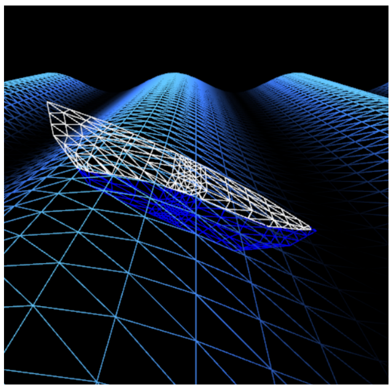

(c)

Figure 25: Results obtained from a boat with 576 triangles in different positions of the wave with amplitude equal to 1.0 and frequency equal to 0.5.

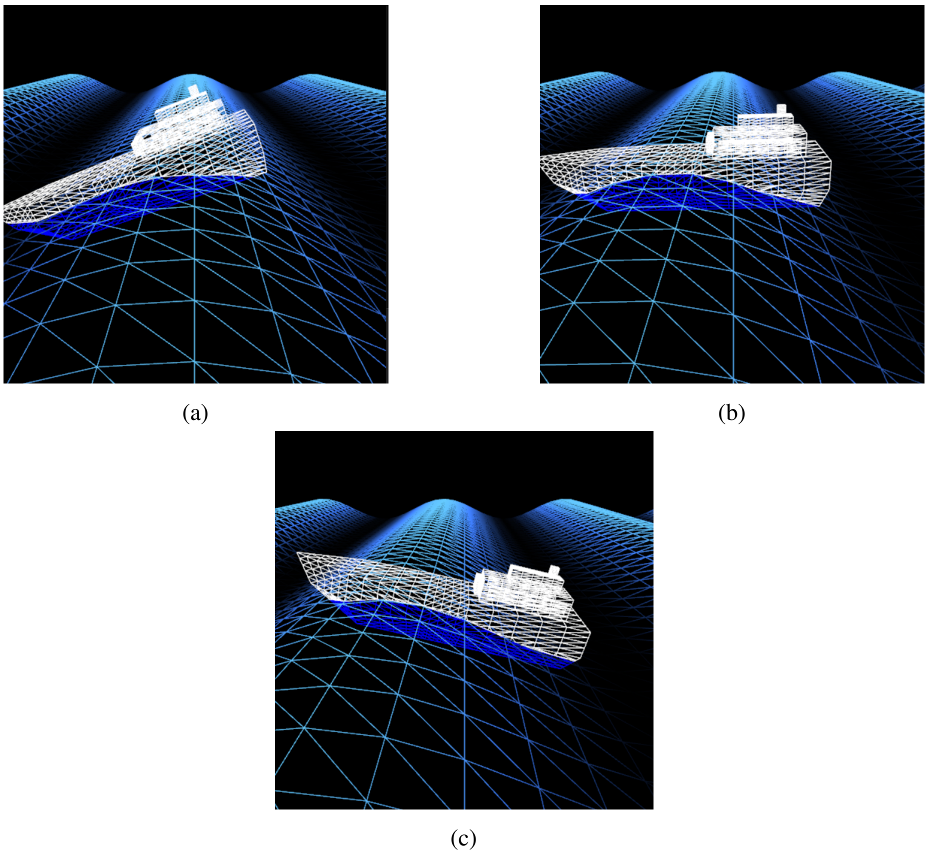


Figure 26: Results obtained from a boat with 2800 triangles in different positions of the wave with amplitude equal to 1.0 and frequency equal to 0.5.

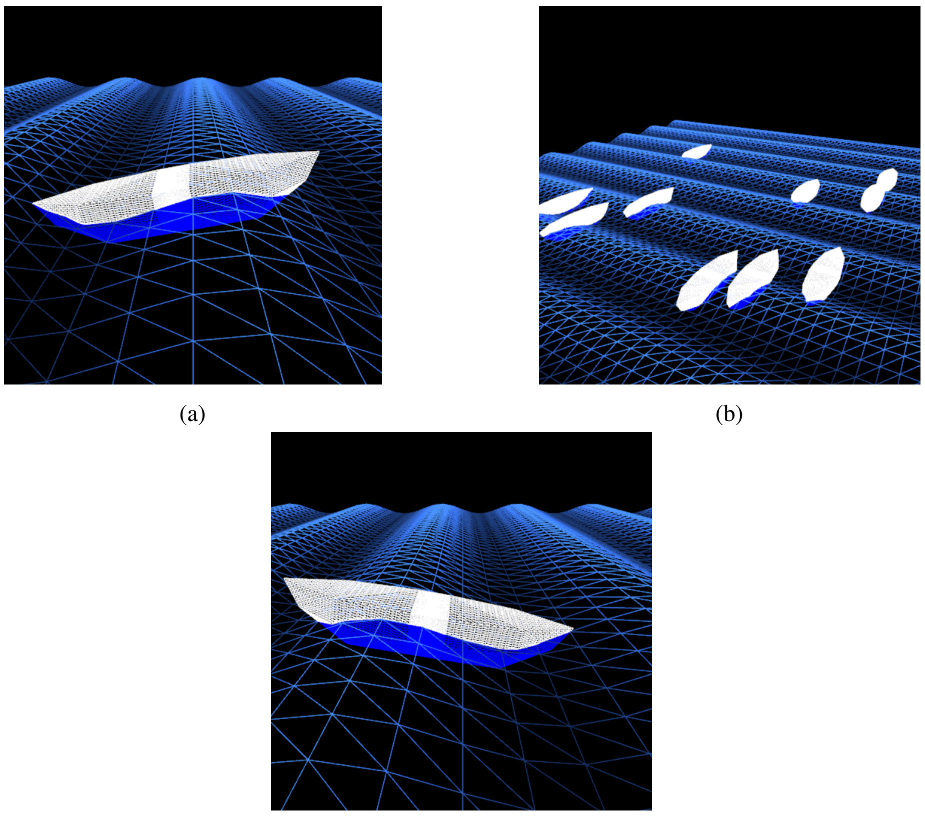

(a) (b) (c)

Figure 27: Results obtained from a boat with 7744 triangles in different positions of the wave with amplitude equal to 0.5 and frequency equal to 0.8.

5

CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

Starting with the main problem of making objects float realistically, we can summarize in a few steps what must be done to achieve this goal. In the realm of realistic object real-time simulation, there are very few articles that delve into the physics involved in a simple way that can be easily adapted in an implementation.

A major issue when it comes to simulating floating objects is the stability of the model, which can be solved if all the forces involved are applied correctly. Within the scope of calculating buoyancy, there are two methods for calculating it, the surface method and the volumetric method. Since the surface method is richer in literary work, like Kerner (2015), this was used throughout the implementation.

It was observed that if only hydrostatic forces are applied the model becomes unstable, this is due to the fact that hydrostatic forces only act correctly if the object is at rest in calm water. If the waters are not calm, there are more forces that must be applied, since the water starts to act on the object and not just the opposite.

The rendering of the object can be done in two ways, the first being non-instantiated, that is, iterative and as in this way the data of each boat was stored in buffers, the way to access them is faster and they do not take up as much memory space. However, in the instantiated form, the data for each boat was stored in a single buffer and took up more space in memory because the information of all boats is stored in one buffer and not just the information of a single boat, making it more expensive for the hardware to access the necessary data. The problem in the instantiated rendering should be solved by changing the data structure so that it does not contain the information of all boats, but only the information of a boat as the non-instantiated rendering does.

As mentioned in Section 2.5, during the implementation there were 3 major problems, namely the interaction of the boat with the water, the scalability of the algorithm, and the stability of the boat.

The interaction of the boat with the water is very important since the aim of the dissertation is to simulate floating objects. This interaction involves adding each of the static and dynamic forces correctly. The interaction of the boat with the water is directly linked to the problem of boat stability. Since these problems are interconnected, we can say that if the interaction of the boat with the water is not perfectly calculated, the stability of the boat is compromised.

One problem that arises during the implementation is the scalability of the algorithm. With the number of boats rendered increasing it is necessary to ensure that the algorithm continues to run smoothly. In addition to the number of boats rendered, with the increase in the complexity of the boats, the algorithm must not undergo major changes in the realism of the simulation.

The increase in the complexity of the boats has little influence when the number of triangles is very low, but the performance decreases and the time spent on the GPU increases when the number of triangles also increases. With this, it can be concluded that the scalability of the algorithm decreases in a seemingly linear way in terms of the complexity of the boats.

With the increase of rendered boats, the number of fps has an approximate evolution of the function of the inverse of x, and the time spent in GPU increases linearly.

5.2 FUTURE WORK

The first optimization that can be done occurs in determining the height of the vertices in the grid. There are two ways to calculate this height. The way done is based on dividing the grid into triangles. The other possible way to determine this height is by bilinear interpolation, thus gaining performance in the calculation of the forces.

Furthermore, the algorithm could be extended in order to support waves with crests accurately. This extension, when given a point to calculate its height, needs to search for the adequate triangle on the surface of the water, in contrast to the current implementation which can provide in constant time that triangle. It also has the complication of receiving one or more planes from the water's surface, which creates the decision of what plane to choose in the height calculation.

As the inertia value of the boats is obtained from an approximation given by *Meshlab*, an optimization would be to calculate the inertia value from the voxelization of the model, thus obtaining a more automatic and independent method instead of depending on an application to have this value. The calculation of inertia from voxelization is explained in detail in Section 2.3.2, where a matrix is made as a function of the object's coordinates.

In addition to the forces calculated to make the boats float, that is, the hydrostatic forces and the hydrodynamic forces, propulsion could be added to the boat so that it can have movement. In order to add movement to the boat, it is necessary to calculate the resistance forces that do not interact directly with the water but only with the boat and then calculate the propulsion that would give movement to the boat.

With the addition of the propulsion force and taking into account the weight of the boat in the water, it would be necessary to add effects of this interaction of the boat in the water, such as foam, splashes, and the trail left by the boat when moving in the water.

BIBLIOGRAPHY

G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. Cambridge University Press, 2000. doi: 10.1017/CBO9780511800955.

David M. Bourg and Bryan Bywalec. *Physics for game developers*, pages 71–83, 321–335. O'Reilly Media, 2nd edition, 2013.

Erin Catto. Erincatto buoyancy, Jan 2020. URL https://github.com/erincatto/erincatto.github.io/tree/master/files.

Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008. ISBN 978-3-905673-68-5. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

Edouard Halbert. Simship, Sep 2016. URL https://fetchcfd.com/view-project/240-SimShip.

Katherine Kaylegian-Starkey. Torque equation & examples | what is torque?, Jul 2022. URL https://study.com/academy/lesson/what-is-torque-definition-equation-calculation.html.

Jacques Kerner. Water interaction model for boats in video games, Feb 2015. URL https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games.

Jacques Kerner. Water interaction model for boats in video games: Part 2, Jan 2016. URL https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games-part-2.

Nejc Lesek. Thousand ships with real-time buoyancy, May 2016. URL http://nejclesek.blogspot.com/2016/05/thousand-ships-with-real-time-buoyancy.html.

Erik Nordeus. Make a realistic boat in unity with c#, Feb 2020. URL https://www.habrador.com/tutorials/unity-boat-tutorial/.

Michael Ray and Emily Rodriguez. Weight and buoyancy, Apr 2018. URL https://www.britannica.com/technology/naval-architecture/Weight-and-buoyancy.

F. H. Todd. Skin friction resistance of ships. *Journal of Ship Research*, 1(03):3–12, 1957. doi: 10.5957/jsr.1957.1.3.3.

James H. Williams. *Dynamics of systems containing rigid bodies*, page 279–285. John Wiley & Sons, 2006.

Part III

APPENDICES

A

SUPPORT WORK

$$\begin{bmatrix} x^2 + (1 - x^2)\cos\theta & xy(1-\cos\theta) - z \times \sin\theta & xz(1-\cos\theta) + y \times \sin\theta & 0 \\ xy(1-\cos\theta) + z \times \sin\theta & y^2 + (1 - y^2)\cos\theta & yz(1-\cos\theta) - x \times \sin\theta & 0 \\ xz(1-\cos\theta) - y \times \sin\theta & yz(1-\cos\theta) + x \times \sin\theta & z^2 + (1 - z^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{66}$$

Equation 66: General matrix of rotation. Where x,y,z are the axis of rotation and θ is the rotation angle.

B

LISTINGS

```
uniform sampler2D texUnit;
...
vec4 points_plane(vec3 a, vec3 b, vec3 c) {
    vec3 normal = normal(a,b,c);

    float d = -(normal.x * a.x + normal.y * a.y + normal.z * a.z);

    return vec4(normal.x, normal.y, normal.z, d);
}

vec4 plane_in(vec2 xz){
    float i = floor(xz.x);
    float j = floor(xz.y);

    float xmax = i + 1;
    float xmin = i;
    float zmax = j + 1;
    float zmin = j;

    float quad_level = (xz.y-j) - (xz.x-i);
    vec4 pl;
    vec3 A, B, C;

    if (quad_level < 0) {
        //first triangle
        A = vec3(xmin, texture(texUnit, grid_coord(vec2(i,j))).r,zmin);
        B = vec3(xmin, texture(texUnit, grid_coord(vec2(i,j+1))).r,zmax);
        C = vec3(xmax, texture(texUnit, grid_coord(vec2(i+1,j))).r,zmin);
    }
    else {
        //second triangle
        A = vec3(xmax, texture(texUnit, grid_coord(vec2(i+1,j+1))).r,zmax);
        B = vec3(xmax, texture(texUnit, grid_coord(vec2(i+1,j))).r,zmin);
        C = vec3(xmin, texture(texUnit, grid_coord(vec2(i,j+1))).r,zmax);
    }
    pl = points_plane(A, B, C);

    return pl;
}

float apply(vec4 pl, vec3 a){
    return (a.x * pl.x) + (a.y * pl.y) + (a.z * pl.z) + pl.w;
}

float dist_surface(vec3 a){
    vec2 xz = vec2(a.x,a.z);
    vec4 pl = plane_in(xz);
    float res = apply(pl, a);
    return res;
}
```

Listing B.1: Calculation of distance between a vertice of the boat and the surface of water. Where *texUnit* is the texture previously mentioned, *quad_level* is the value that identifies which of the triangles of the plane where the vertex of the object is located, *x* is the x coordinate of the vertex, *y* is the vertex coordinate, *xmin* is the the minimum coordinate on the x's axis, *ymin* is the the minimum coordinate on the y's axis and *grid_coord* is the coordinates in the grid. As the heights are stored in the channel r of the texture to access the height of the points it is necessary to access this channel.

```
vec3 t_center(vec3 A, vec3 B, vec3 C) {
    return (A + B + C) / 3.0;
}
```

Listing B.2: Calculation of the triangle centroid. Where A, B and C are the vertices of the triangle.

```
vec3 normal(vec3 A, vec3 B, vec3 C) {
    vec3 U = B - A;
    vec3 V = C - A;
    vec3 N = cross(U, V);
    if(N == vec3(0.0)) return vec3(0.0);
    else return normalize(N);
}
```

Listing B.3: Calculation of the triangle normal. Where A, B and C are the vertices of the triangle.

```
float area(vec3 A, vec3 B, vec3 C) {
    return norm(cross(B - A, C - A)) / 2;
}
```

Listing B.4: Calculation of the triangle area. Where A, B and C are the vertices of the triangle.

```
float norm(vec3 a) {
    return sqrt(a.x * a.x + a.y * a.y + a.z * a.z);
}
```

Listing B.5: Calculation of the norm of an vector a.

```
float cos_theta(vec3 A, vec3 B, vec3 C) {
    vec3 normalized_triangle_vel = normalize(triangle_velocity(A, B, C));
    return dot(normalized_triangle_vel, normal(A, B, C));
}
```

Listing B.6: Calculation of the cos_theta of a triangle. Where A, B and C are the vertices of the triangle.

```
float submerged_triangle_area(vec3 A, vec3 B, vec3 C) {
    float submerged_triangle_area;

    if (cos_theta(A, B, C) < 0) {
        submerged_triangle_area =  0.0;
    }
    else{
        submerged_triangle_area = area(A, B, C);
    }

    return submerged_triangle_area;
}
```

Listing B.7: Calculation of the submerged area of a triangle. Where A, B and C are the vertices of the triangle.

```
vec3 triangle_velocity(vec3 A, vec3 B, vec3 C){
    vec3 v_B = boat_velocity;
    vec3 omega_B = angular_boat_velocity;
    vec3 r_BA = t_center(A, B, C) - boat_position;
    vec3 v_A = v_B + cross(omega_B, r_BA);
    return v_A;
}
```

Listing B.8: Calculation of the triangle velocity at the center. Where A, B and C are the vertices of the triangle.

```
float reynolds_number(vec3 v) {
    float nu = 0.000001;
    //Reynolds number
    float velocity = norm(v);
    float Rn = (velocity * boat_length) / nu;
    return Rn;
}
```

```
float resistance_coefficient(float Rn){
    float Cf = 0.075f / pow((log10(Rn) - 2.0f), 2.0f);
    return Cf;
}
```

Listing B.9: Calculation of the resistance coefficient from reynolds number.

```
vec3 translate(vec3 trans, vec3 a){
    vec3 result = a + trans;
    return result;
}

vec3 rotate(vec3 rot, vec3 a){
    float d = norm(rot);
    float c = cos(d);
    float s = sin(d);

    float x = (d == 0 ? 0 : rot.x / d);
    float y = (d == 0 ? 0 : rot.y / d);
    float z = (d == 0 ? 0 : rot.z / d);

    float xx = x * x;
    float xy = x * y;
    float xz = x * z;
    float yy = y * y;
    float yz = y * z;
    float zz = z * z;

    mat3 m_rot = mat3(xx + (1 - xx) * c,    xy * (1 - c) + z * s,   xz * (1 - c)
        - y * s,
                      xy * (1 - c) - z * s, yy + (1 - yy) * c,      yz * (1 - c)
                          + x * s,
                      xz * (1 - c) + y * s, yz * (1 - c) - x * s,   zz + (1 - zz)
                           * c);

    a = translate(-boat_position, a);
    a = m_rot * a;
    a = translate(boat_position, a);

    return a;
}
```

Listing B.10: Linear transformations applied to a vertex of the boat. Where *trans* is the vector of the transformation translation, *rot* is the vector of the transformation rotation, *d* is the angle of rotation, $boat_position$ is the position of the center of mass of the boat.