AALBORG UNIVERSITY - COPENHAGEN

Dynamic surface rendering of water in liquid and solid states in real-time application

by

Georgi Anastasov and Svetla Tomova

A thesis submitted in fulfillment of the requirements for the degree of M.Sc. Medialogy

Copenhagen – Denmark

28 May 2015

Aalborg University Copenhagen

Semester: MED10
Title:
"Dynamic surface rendering of water in liquid and solid state in real-time application"

Project Period:
Feb. 2015 - May 2015

Semester Theme:
Master Thesis

Supervisor(s):
Georgios Triantafyllidis

Project group no.:

Members:
Svetla Tomova
<stomov13@student.aau.dk>
Georgi Anastasov
<ganast13@student.aau.dk>

Copies: 3
Pages: 93
Finished: 27.05.2015

Aalborg University Copenhagen
A.C. Meyers Vænge
2450 København SV, Denmark

Semester Coordinator:
Stefania Serafin

Secretary:
Lisbeth Nykjær
Phone: (+45) 9940 2470 | Email: lny@create.aau.dk |
Aalborg University | Frederikskaj 12, 1st floor | 2450
Copenhagen SV

Abstract:
Real time applications are being popular theme in many disciplines. As the age of technology allows more and more alternatives, we investigated one particular domain from the computer graphics – water simulation. Our implementation is based on modern techniques, widely used in games computer graphics, from basic texturing to advanced run-time reflections and bump mapping. We produced a sophisticated result with extensions to the common surface water rendering, by including additional features – run-time interaction with the surface water rendering, based on texture injection. Our proposed simulation model is divided into three main sections – visuals representing the rendering of the water, physics simulating the water to object coupling, and finally interaction of objects or input to water. The results showed significantly positive outcome about the last one, where an interaction texture was rendered and used inside the visualization of the water to create water ripple effects – widely investigated area.

Acknowledgments

We would like to express our appreciation to our supervisor, Georgios Triantafyllidis assistance professor in Department Architecture, Design and Media Technology AAU-Copenhagen, for his guidance, assistance and supervision.

Contents

List of Figures

Acronyms

CG – computer generated
RTC – real-time computation
VR – virtual reality
RT – real time
FX – effects
VFX – visual effects
API - application programming interface
GPU – graphical processing unit
CPU – central processing unit
HCI – human-computer interaction
GL – graphics library
DoF – depth of field

Preface

During our internship, we face numerous problems and one of them was creating computer graphic water. This particular problem attracted out attention and we decided to find a solution, regarding computer games and virtual environments.

In this thesis, we will be focusing on developing a dynamic surface rendering of water liquid and solid states in real-time application. The main purpose of the writing was to solve one of the problems that we are interested in and develop our skills further as well as create a demo that we can continue working on afterwards. We wanted to experiment with a number of techniques with the possibility of either find an optimization or general new technique as well as define a full scope of water render in real time computed applications.

The design and development of the water simulation was made by Georgi Anastasov and Svetla Tomova, during the spring of 2015 in Aalborg University, Copenhagen. The submission contains a writing paper and CD with all working files and serves the purpose of MS Thesis project.

Dynamic surface rendering of water in liquid and solid state in real-time application

Georgi Anastasov
MS in Medialogy, Computer Graphics
AAU Copenhagen Denmark
ganast13@student.aau.dk

Svetla Tomova
MS in Medialogy, Computer Graphics
AAU Copenhagen Denmark
stomov13@student.aau.dk

Abstract

Real time applications are being popular theme in many disciplines. As the age of technology allows more and more alternatives, we investigated one particular domain from the computer graphics – water simulation. Our implementation is based on modern techniques, widely used in games computer graphics, from basic texturing to advanced run-time reflections and bump mapping. We produced a sophisticated result with extensions to the common surface water rendering, by including additional features – run-time interaction with the surface water rendering, based on texture injection. Our proposed simulation model is divided into three main sections – visuals representing the rendering of the water, physics simulating the water to object coupling, and finally interaction of objects or input to water. The results showed significantly positive outcome about the last one, where an interaction texture was rendered and used inside the visualization of the water to create water ripple effects – widely investigated area.

Categories and Subject Descriptors

I.6. [Computing Methodologies] – simulation and modeling

J.5. [Computer Applications] – arts, fine and performing

General Terms

Simulation, Experimentation, Animation

Keywords

water, surface rendering, visuals, interaction, physics, texture, shading, ice

1. Introduction

Water in real life is immense and unpredictable and hard to explain. That is why it always has been an interesting topic in the field of games and virtual environment. All kinds of water characteristics as well as different methods and techniques, has to be considered when modeling and rendering water in CG world. Starting from the color of the water and followed by the flow of the liquid, reflection and refraction and etc.

This thesis serves to present the development of dynamic, realistic water surface rendering, using Unity3D platform. The focus of the simulation is to create real-time water liquid and its transformation into solid surface - ice. For the development of a realistic simulation of water, we relied on three component visuals, physics and interaction. It is important to be said that the three elements, are not equally valuable as interaction and physics take a smaller role in this project.

The problem addressed in this project, is to model and render realistic liquid and ice surface in real-time, using visuals and physics, combined with a small part of interaction intending to engage observer`s attention to a certain point.

2. Previous work

In this section will be split into three sub-sections. The first one will explain the meaning and use of CG water as well as the most common techniques in general. The next sub-section called Low level of analysis serves to present the key features in computer graphics, for water modeling and rendering, organized according to their purpose: visual, physics and interaction. Followed by the High level analysis sub-section where all these features are further analyzed.

2.1 General analysis

2.1.1 History of water rending in computer graphics

Computer graphic history started back in 1950 but the term “computer graphics” was entered by William Fetter in 1960 [1] to describe new design methods. Since then, the field of computer graphics was widely enlarged. Moreover, CG water or fluid simulation has become a valuable part of it.

The history of computer graphic water began with just a simple low quality blue and white texture representing 2D water and it was widely used in Nintendo games. Later on this technique was further developed in order to be added transparency to the water. In the 80s, computer graphics were popularized by the increasing need of them in films such as “Star Wars” and others. This period was marked as the golden era of videogames, the interest in computer graphics like water and fluids increased significantly. For creating CG water, more realistic particle systems were introduced by William T. Reeves [2] for creating 2D fluid water and simple water effects. After short period of time and a few scientific achievements in the field, computer graphic water was able to be presented as a 3D object. In the 90s, the computer graphics became more popular than ever as there were finally close to photorealistic. They were highly needed in games, films and multimedia. Film like “The Prince of Egypt” used computer graphics to present the parting of Red Sea. Another great example of all time, is “Titanic”, which proved how important are the CG effects for the overall appearance and immersion. Improving the realistic simulation of 3D water, in 2000s, computer graphics for water rendering in films like “Finding Nemo” and “Ice Age” reached new levels and achieved a huge success. Another film that brought computer graphic for water simulation to the next level, was “Perfect Storm” where

all kinds of water effects were involved. Using fluid dynamic system a stormy ocean and huge waves were created. [3]

Nowadays, water in games and films has been evolved impressively and achieve realistic fluid dynamics. Water in computer graphics come a long way from the single blue texture presenting 2D water to these days when we have the knowledge and technologies to create photorealistic water for films or real-time water for games.

2.1.2 Application of computer generated water

In the century of technology and media development more and more developers are struggling to implement different aspects from the reality into their work, whereas this applies for everything that we can naturally see, feel or even smell, and the extreme advance in computer graphics in the past couple of decades serves as a proof of that. Computer graphic water is no exclusion. Whether it is for movies, games, virtual environments or even more research-oriented use, a proper model is usually sought to serve its desired purpose.

CG objects are the core elements when one is designing media settings. In games, there is usually a limited world that the player can explore, therefore a more natural and sophisticated way is needed to boundary and describe the predetermined rules of the game or simply the part with the limitations of this virtual world – either a short fence [4], a high mountain or in the scenario of this writing an endless CG ocean/water. In the terminology of game development this type of restrictions to the game world is commonly known as the term “invisible wall” [5]. In the case of using a large body of water as a boundary can have its pros and cons – surely the user would not like the idea to be unable to interact with the water body leading to lowering the immersion of the player as immersion and interaction go hand by hand, but on the contrary the “endless” water could serve as infinite area, again as stated above – a more natural way to put a boundary to a so called open world or a virtual world where the user can move freely [6]. However, immersion and interactivity are playing major role in games, therefore every element from the environment needs proper feedback model. If the CG water does not react to a user input or vice versa, that could lead to lack of immersion. In order to have that both-ways reaction, the rendered water needs to be constructed as much possible as in real life. When the difference between the real life and virtual simulation is down to minimal value, we could then acknowledge that a high level of immersion is accomplished and the CG content is “perceived as real”[7]. This leads the development of the computer graphic water to a new level, one that requires more care to the details, but nonetheless creates different problematic areas. Usually when it comes to creating a CG model, a research is carried in order to define these problematic areas – Where does the model is being used? What purpose does it server? and so forth.

Unfortunately, not all problematic areas are solved in a realistic approach, when it comes to games, as there are computational issues when developing CG content. This is a result of the hardware limitations and usually creating a realistic water simulation in games is hard to achieve, therefore most of its "sub" problematic areas are usually "faked", which we will discuss further in the research. There are two major groups of simulations – on one hand there is real time computing, where the data is acquired, processed and affected result is returned "sufficiently quickly" [8] and on the other there is the pre-computed model, where all of the previous mentioned stages are calculated before the "materialization of the results"[9]. Games are partially falling under the RTC category, since they sometimes make use of pre-computation (ex. In-game Cinematic), while computer augmented movies are entirely from the second category. Therefore, when addressing the issue of creating CG content in terms of movie industry there are a lot more opportunities and the end results can have exact model of behavior as it would be in real life. The use of computer graphics for water modeling and rendering in movies, requires a sufficient amount of data and a realistic model approach or modeling technique, usually described using a certain number of physically correct formulas and laws, resulting in a simulation that could not be easily differed from reality, as expected from the viewers to perceive. We will mention some of these physical methods later, as they can find their usage inside RTC water simulation.

Virtual environments, in the sense of virtual reality (shortly VR), are another up going trend. The idea behind one being able to interact and place himself inside a CG environment, can be traced almost a century ago [10], but the approach towards the topic became much more realistic in the past few decades also known as the computer graphics revolution [11]. The use of virtually created water could have huge potential, for example – a sightseeing of the Niagara Fall, or even a virtual walk on the surface of Europa (one of the Jupiter's closest moons, known to have mainly an icy water crust). Although the idea could lead to great results, the tools are limited, as VR requires the use of RTC. We could easily characterize the virtual environment issues to be the same as the ones marked in game development and games in general. The only difference is the outcome of the product, as the last one's purpose is to bring a certain level of joy/fun etc., while the other one could be potentially used in many different areas, for example - advertising, sightseeing, or even with a research purpose. One such example from the time of this writing is introduced by Nvidia (a leader in graphical processing units, shortly GPUs, manufacturer) where researchers showcased a new integrated calculation of bouncing/ambient lighting, calculated in real time [12], that served as part of a CG reconstruction of the events from July 1969, when Niels Armstrong landed on the Moon surface. Thus having the ability to simulate RTC water can also have its application in research and scientific areas, with the ability to see and extract knowledge or results at run time.

2.1.3 General approaches in water simulation

Water simulation, in both RTC and pre-computational approach, is handled by various ways, depending on the desired results. This again, as mentioned earlier, is restricted by the limits of the hardware. We categorize most of the possible solutions of animating water regarding the computational requirements.

- **Surface water**

First approach is as we refer to it as surface water. This type of simulation is the fastest one, as it is a simulation done onto a 2D surface plane that can be both used in 2D and 3D space. Surface water simulations allow fast computation of the geometry. This general approach of using a single plane geometry, that holds the visualization of water, is vastly used even nowadays in top title games such as “Crysis”, “Borderlands”, “Elder Scrolls V: Skyrim” and etc. Commonly surface water simulation is referred as procedural water, where no physics are applied, rather a constant model of movement is implemented, either texture movement (tilling and offset) or per vertex movement. Although its simplicity, this method of water simulation can easily be enhanced by combining the 2D water representation with advanced methods of texture handling (which we will discuss later in this writing) resulting into extremely realistic simulations of water bodies, see Figure 2.1.

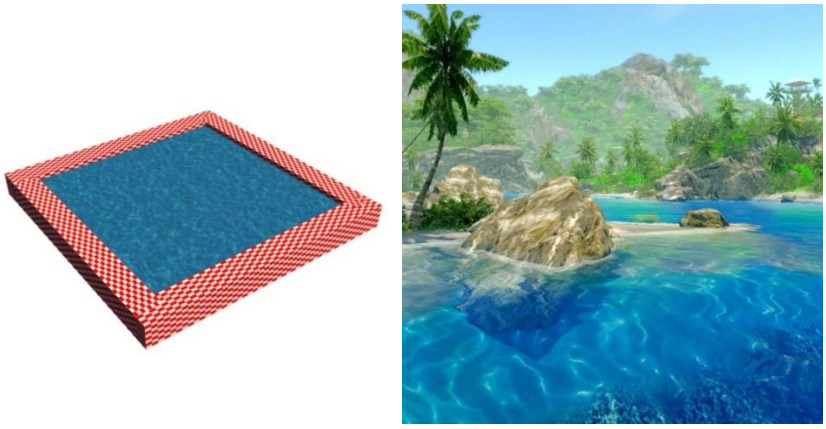


Figure 2.1.: Surface water rendering - On the left is a simple rendering of a plane with texture; on the right is enhanced version of water surface in a popular FPS game “FarCry”.

- **Fluid water**

A much more realistic results can be accomplished, at the cost of computational time, by using a technique, which involves a third dimension, simply said a volume water. There a few different sub categories, when developing volumetric water simulation, but the general approach is to answer the question – how is the fluid and its movement represented. In the previous technique the water was represented by a 2D surface

plane with vertex or pixel movement, while in the volume scenario we need something that will hold the data of each water volume consistency, regarding our consideration of what is under the water surface – mass, velocity etc. In computer graphics this is usually approached in two different ways, namely Eulerian grid-based simulation and Langrangian particle-based simulation [13], see Figure 2.2. The difference between the two mentioned models, is that in the Eulerian one, we are assigning values of fluidity and movement based on specific position in a grid world space through which the fluid passes, while the Lagrangian model keeps track of these values for each fluid particle, while that particle passes through the world space [14].

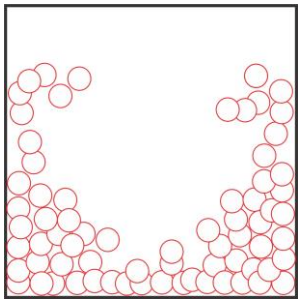


Figure 2.2.: Particle-based water simulation - the volume of the water is also taken into consideration.

As the name implies, the first volumetric technique is model [15], where the simulation space for the fluid, is divided into small grid cells, each one of which plays the role of a data container and all of these containers combined together are defining a parametric flow field. The most common issue with this type of simulation is that empty grid cells are computed as well. There are a few optimizations to this approach, which are mainly affecting the key feature in the simulation – the grid. The following grid types can be implemented in grid-based simulation:

- Fixed grid cells [16] – the space is equally divided and each cell contains parametric information used for animating the water, see Figure 2.3.a).
- Adaptable and tall grid cells [17] – are both based on discarding amount of computation, by not combining cells in areas of the grid, where the animation of the water is either active or less than a threshold value , see Figure 2.3.b).

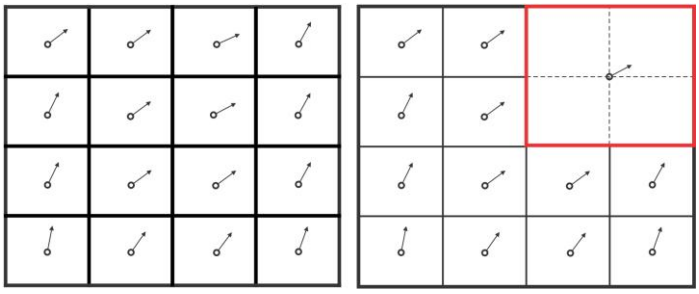


Figure 2.3.: Eulerian grid-based water simulation - a) Illustration of fixed grid cells on the left; b) Illustration of adoptable grid cells on the right;

The second type of volumetric simulation stated above is the particle based water simulation also characterize as Lagrangian model [15]. This is generally more realistic approach, as the physical calculations are done for each water particle, thus simulation more "lifelike" movement of the water volume, again at the cost of even more computational time. The last can be said to be the main technical restriction in use for RTC applications. The idea behind is that each particle has its influence onto the rest of the particle mass and the amount of that influence categorizes few distinct ways of accomplishing the simulation:

- Particles without inner-interaction – most common simulation because of its simplicity, as the particles do not interact with each other, or in other words with no collision between the particles and no particle influence forces, for example attraction particle force [18]. This is vastly used in computer graphics for creating visual FX, as the computational time is considerably lesser than the other cases. Considering the topic of this writing, this type of simulation can be used for creating supporting water FX like water splashes and water foam, Figure 2.4.a).
- Particles with inner-interaction – oppositely to the previous type of simulation, this one is much more heavy on the computational criterion, as all possible inner forces are consider when calculations are applied, for example collision, attraction force, spring force [18] and etc., Figure 2.4.b). This type of simulation is usually used in pre-computational approach and for research and scientific purposes.
- Other particle simulations – there are a few more ways to simulate fluidity, namely smoothed-particle hydrodynamics (SPH), Lattice Boltzmann method but they are beyond the scope of this paper.

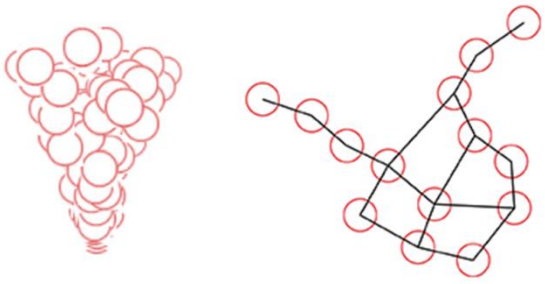


Figure 2.4.: Lagrangian non-grid based water simulation - a) particles without inner interaction being emit on the left b) particles with inner interaction influencing each other on the right.

- **Height field water**

This general approach for simulating water is considerably useful, because of its simplicity. The water volume is presented by number of cubes with different heights, calculated either by a certain formula [19] or by extracting values from a height map, as seen in Figure 2.5. Height field water is usually used when physically correct simulation is needed, with regards to object-to-water

interaction and vice versa – for example a boat that creates distortion wave ripples in the water. However there are two main problematic areas with this type of water simulation:

- Breaking waves – since the idea is to control height field values using an output value from a formula, meaning that for each cube point P with coordinates (x,y) we have only one height value [20] leading to inability to create split geometry or in water simulation context breaking wave.
- Number of cubes – when it comes to simulating large bodies of water, this method becomes quite ineffective, as calculations on the height field distortion is applied to the whole field, unless the simulation area is restricted [21].

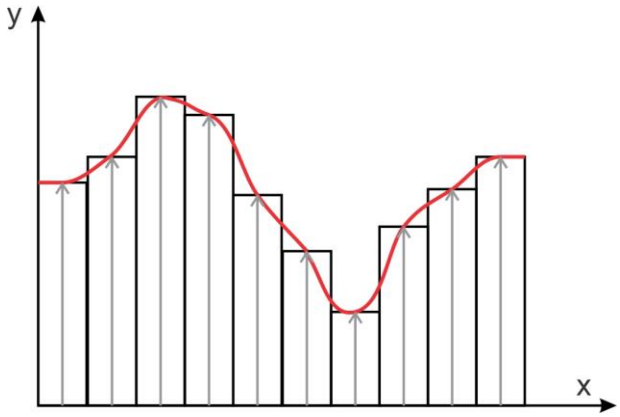


Figure 2.5.: Heightfield water simulation scheme - Illustration of wave formation using height field values.

2.1.4 Computer graphic pipeline (basic overview)

- **Geometry structures**

Simulation, in the sense of visualization and animation of CG content, can be executed by a set of stages commonly known as the computer graphics pipeline. In order to fully understand how we can accomplish our goals we need to have common knowledge about what are the basic elements and structures that we can operate with. As a simple summarize, we can assume that computer graphics are entirely based on math and mathematical principals, which are transferring numerical representation into visual. In order to achieve that we need basic geometrical form to describe those visual representations, namely points, lines and areas, more commonly known in computer graphics context as, respectively, vertices, segments and faces(see Figure 2.6):

- Vertex – a data container, or a structure, that contains information about position in either 2D or 3D space and some other additional parametric values [22] - for example Normal, Tangent, Texture coordinate and etc.
- Segment – the enclosed space between two vertices

- Face – a polygonal geometry structure, build from more than 2 segments

In computer graphics these structures are used to form the shape of the models, or in other words the geometrical presentation of the model. Usually faces are divided into the smallest possible polygons – formed by triangles, as the triangle is equilibrium geometry, meaning that no matter what we do (move, rotate, scale or even skew) the inner sum of the angles will always be 180^{o}, and also because triangles are basic geometry created by the use of 3 non-collinear points, which in Euclidean geometry defines a unique plane or in our case unique triangle.

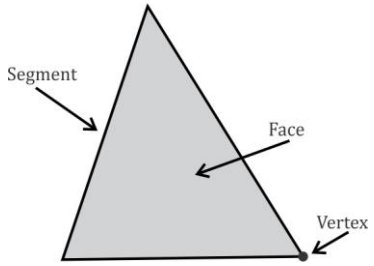


Figure 2.6.: Basic computer geometry, namely triangle, described by the three basic elements in CG – vertex, segment and face.

- **Computer Graphics Pipeline**

Assuming that a model is being created, the data of this model is being handled by the so called graphics engine, which on its own turn is based on some low level programming language. Most common one is the OpenGL, standing for Open Graphics Library – an extension of the popular programming language C/C++ that used for rendering 2D and 3D vector graphics [23]. The application programming interface (API) is used to access and execute algorithm instructions on the GPU. The main idea of the GPUs is to handle multiple tasks at the same time (parallel processing), focusing mainly on the hardware-accelerated rendering output. In order achieve that output, the model's geometry data is being passed between the following stages [24] (note that not all stages are included, but only the most notable ones), see Figure 2.7:

- CPU to GPU – a list of all the vertices, defining the model's geometry, is passed from the CPU to be process by the GPU;
- GPU to Vertex program – a set of instructions from the API allowing the programming manipulation of each vertex;
- Vertex program to Primitive Assembly – triangulation of the geometry based on index list passed from the model's data, simply creating wireframe representation of the model out of the vertices position;
- Assembled primitives to Rasterization – mapping the assembled primitives to the screen pixels also known as interpolating the pixel data, that each triangle covers;

- Rasterized image to Fragment program – a set of instructions from the API allowing the programming manipulation of each fragment (covered pixel);
- Fragmented image to Testing – testing applied based on buffer testing, for example depth testing (if an object is covered/in-front of another object), or transparency blending
- Tested fragment to frame buffer – data is finally passed into a frame buffer, used in sequential rendering on the screen. Meaning that while one rendered image is on the screen, another one, named frame buffer, is being filled with the final fragment outputs and when that process is finished, the two render frames are swapped [25

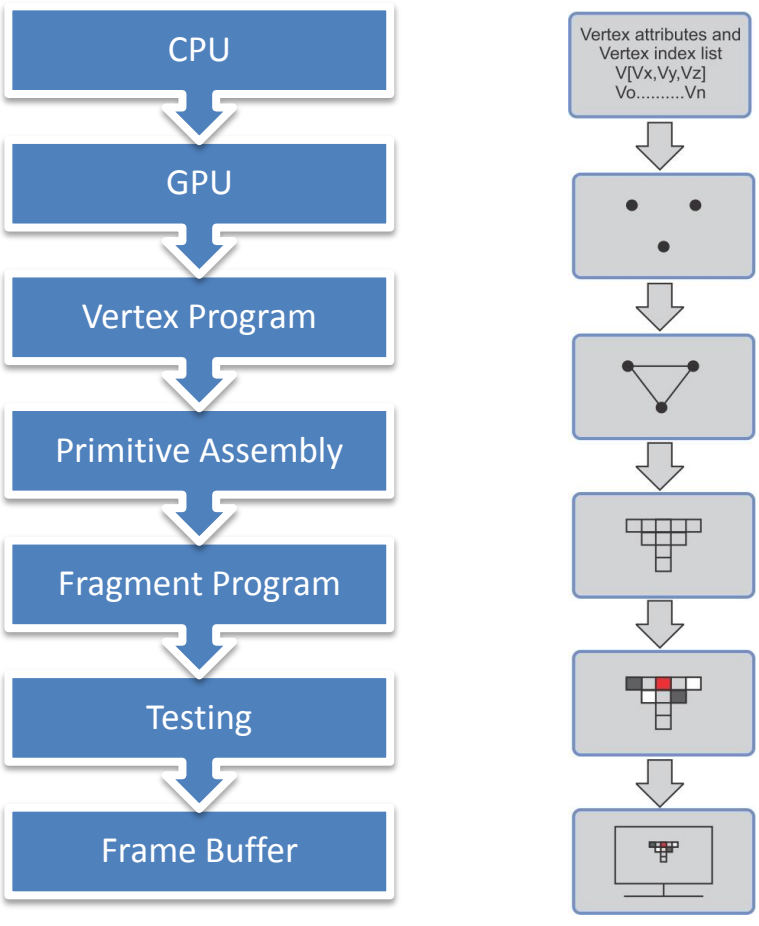


Figure 2.7.: General CG pipeline - a) consecutive stages on the left side and b) on the right side - graphical explanation of the main stages

As seen from above, the two programmable joints in the CG pipeline are the vertex and the fragment program, usually referred to as Shader program, or simply a Shader, which allows the developer to manipulate both the vertex's input/output as well as the fragment's final output color. This is allowance was brought in the first years of the 21st century and it had a truly noticeable impact on the graphics hardware, transforming it from fixed to programmable [26]. The

programmability is consisted of applying changes to the output rendering, for example applying tint color or texture color (fragment operation) or calculating light affection of the model (could be both vertex and the fragment operation).

- **Data transformations and coordinate systems**

As mentioned earlier, the model's data is being passed through out a number of stages and that transfer is accomplished by the use of mathematical transformations. If a point in 3D space is presented by P <x, y, z> we can define different spaces and transform that point in relation to those spaces using matrix multiplications. A simple example of this can be observed in a robot arm kinematics, where the robot's main processing unit needs to know where all of the arm's joints are relatively to a common space, while each individual part can have its own "local" space [27]. The same thing applies for the computer graphics transformations and the following multiplications are applied to transform the vertices from their local position into the screen space [26], see Figure 2.8:

- Model transformation – from local space (position of model's vertices relatively to the model's origin) to world space (where all models are positioned relatively to one global origin)
- View transformation – from world space to camera space (where all objects are positioned relatively to the camera origin/position)
- Projection transformation – from camera to clip space (a geometrically defined space, named frustum, that defines what is viewed by the camera; geometry outside that space is clipped)
- Perspective transformation – from clip space to normalized device space, simply applying perspective division.
- Viewport transformation – from normalized device space to window/screen space (mapping perspective geometry to screen pixels)

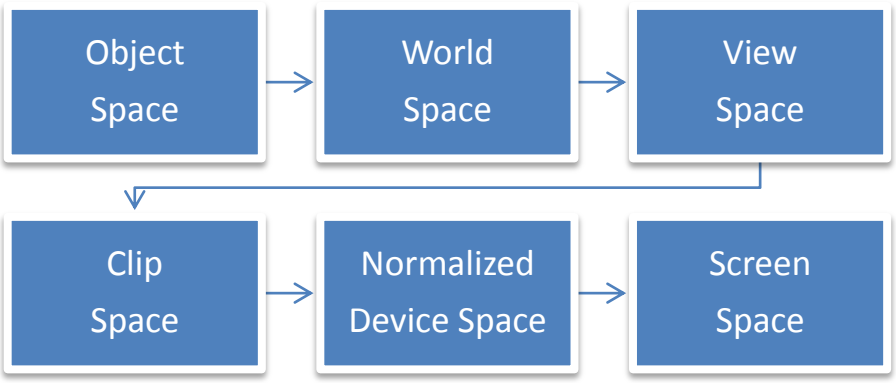


Figure 2.8.: Scheme of the data transformation from Local space to Window space

2.1.5 Importance of optimization

As mentioned earlier, there are two ways of computing the data, reminding for convenience – RTC and pre-computation. In order to achieve a significantly good results when working with RTC we need to have optimized model of the simulation. This is regarding how thing are being implemented – for example an equation for light calculations can be implemented on both per vertex and per fragment stage. The difference is that if a triangle has only 3 vertices (formula calculations will be executed 3 times), it can have thousands of fragments on the screen (formula calculations will be executed thousands of times). Although that, modern GPUs have sufficiently enough processing power, to handle those kind of heavy calculations. To sum up the implementations and enhancements of the water animation and VFX done inside either one of the Shader program parts is a matter of choice between efficiency (vertex implementation) and aesthetics (fragment implementation).

2.2 Low level of analysis

This part of the research aims to study computer graphics for water simulation focusing on the three main subdivisions – visuals, physics and interaction in order to analyze the elements that each one of them includes for realistic further simulation of water.

2.2.1 Visuals

Water in general doesn't have a color, but lakes, seas and oceans appears to have a blue color as it reflects the color of the sky. This appears to be one of the theories. Another explanation of why ocean looks blue is because the red, yellow and orange lights absorb more strongly that the blue light. [28] Meaning that, most of the light that is scattered and reflected back, is blue. Sometimes the water can have a green color because of the sediment or the plant around it. Like rivers appears to be green as the environment around plays an essential role in the formation of the color.

For creating more realistic CG water, adding transparency to the water is beneficial. Transparency could be explained as how easily the light can pass thought the surface. [29] In general, the transparency of the water depends on the amount to particles in the water. In other words, the water will be more transparent when it has less particles in it. For instance, the reason why the plants grow in shallow areas, around the coast, is because the light can penetrate thought the water and reach the plants. Regarding the proper simulation of water along the coast, coastal blending should be added. It includes blending of the water color and proper wave distribution of the waves.

Wondering how the wavelengths of light interact with the environment and the surfaces, reflection is one of the terms that has to be considered. It is based on physics of the light when the light bounces of a surface and the human eye reacts to

that event. [30]For specular reflection, the law says that the angle at which the light hit the surface, at the same one it will be reflected. For realistic feeling and better immersion of the observer, reflection is extremely important feature of water surfaces.

Another essential term in that field is the refraction. When the light passes through the surface, resulting in a bent light beams, and everything that is inside the liquid and also reflecting light (no need to have mirroring effect) has a distortion in the visual representation. [31]Refraction is explained as light travels with different speed, regarding different materials. Also includes Chromatic Dispersion when colors are refracted with different angle based on their wave length.

The amount of light reflected and refracted based on the viewing angle is described by the so called Fresnel effect. Observing from above, it is hard to see much of the reflected light but from around your eye level, you will be able to see much more reflected and refracted light. [32] Addressed to the Fresnel effect and transparency properties is the depth of water. It is a visual explanation of the liquids density.

Regarding the underwater simulation and coloring, light plays an essential role for realistic computer graphic water. The water absorbs light differently from lowest to highest energy - starting with the red, orange, yellow, green, blue and violet. The same way is used for absorption of light underwater. The penetration of the light depends on the depth. Also objects underwater appear to the observer to be closer and larger because the refraction under water is greater than the one in the air. [33] Other part of the underwater simulation, are caustics. Caustics are the projection of the refracted light from a refractive surface to another one. In computer graphics the focus of caustics usually falls on the aesthetics, not on the physics.

When talking about water visual effects, water spray and foam has to be mentioned. Water spray and foam are details of the water surface, usually describing the motion of the water and the air captured in bubbles, floating onto the water surface. The water foam depends on the many circumstances like the wind and waves. The foam should appear after the peaks of the waves and disappear over time. The water foam is formed by numerous bubbles sticking to each other. They are essential for the realism of water simulation. Water bubbles are formed when air churned in the water surface by the motion of the waves. That is why they usually appear in the upper part of the water surface. [34] For simulation of highly photorealistic water bubbles, it is valuable to know that depending on the weather, bubbles appear differently. For instance, when the sun is shining a close look on the bubbles on top of the water foam will show that all kinds of the spectrum light could be seen in the water bubble.

When analyzing the visual water properties, transformation from liquid to solid body includes describing the state when the water turns into ice. At certain

temperature (freezing point) water becomes a solid body – ice. Depending on the presence or absence of air bubbles in the frozen object, the ice could look more white in the middle or transparent, respectively. In the environment ice appears in different forms depending on numerous factors affecting it. Ice on lakes forms a thin layer on top of the water surface like an ice crust and then spreads downwards. On rivers, formation of ice is considerable harder because of the moving water surface. Then a thin layer above the water could appear increasing its width around the coast as the water there is moving slowly. Similar to the river, the ice in the ocean could appear in the form of a drift ice floating in the water or ice bergs.

2.2.2 Physics

The water physics is about the interaction of the water with the world, its behavior and component-wise explanation. Physics is about the matter construction, its motion and the variables describing these states.

One major topic regarding water physics is called buoyancy. It is an upward force exerted by a fluid that opposes the weight of an immersed object, which is part of the calculation of how objects are floating in fluid environment. Depending on the mass of the object it will sink, float or stay neutralized. [35] Meaning that if the upward buoyant force if greater than the weight of the object it will float, if the opposite happen it will sink. When the object and the water have the same density, it will be neutral.

Other physics property is viscosity – a measure of the fluid resistance to deformation, also explained as “thickness”. Different fluids have different viscosity. While water have lower viscosity and makes bigger splashes, honey have the opposite properties are splashes are considerably smaller. Water surface behavior describes the motion of viscous fluids from using just simple sine wave to more advanced ones like Navier-Stokes equations.

In the sense of analyzing fluid`s motion and changes, it had to be mentioned the term force field. It describes the inner interaction of the water particles on a lower level of chemistry, in the case of motion, how the react in-between and how that reaction changes the whole mass, mostly used for computational fluids.

In computer graphics effects like water ripples are valuable for the overall realism and immersion of the observer. Therefore, physical explanation is needed. Water ripples are formed when the weight of the object “displaces” part of the water from its equilibrium and recoils back up. That is followed by the spreading circle water ripples, expanding by size over the water surface. In other words, water ripples is how the water responds to contact/interact with objects.

Regarding the water states, water can turn to solid object or gas depending on the temperature. When water passes on its freezing point, transformation from water to ice occurs starting with a formation of layer of ice, where the thickness of

the layer depends on different factors like water movement, temperature and etc. Upon freezing, ice expands its mass as water crystalizes into hexagonal form, containing more space than the liquid. [36] This is the reason why iceberg can float. Regarding rivers, they are not expanding when freezing as the quantity of the water coming, decreases. On the other hand, when water boils, water starts to evaporate rapidly and the water mass reduce.

2.2.3 Interaction

Interacting with a CG objects is based on mainly two things – the properties of that object and the freedom imbedded by the code. For example if we have a 3D mesh of a cube, possible interactions based on its properties would be: motion (translation, rotation, scaling) or changing color, texture etc. Furthermore, other interactions imbedded could be playing with the topography of the object – extruding polygons, deleting edges etc. This process of interaction is accompanied by two main stages – first one is user`s interaction towards the object and second stage – its respond to this interaction back to the observer by informational feedback (visual, sound, tactile etc.).

When object of interaction is water, each one of the described above water properties (physical or visual) can be used for interaction between the user and the CG water. The main purpose of interaction is to immerse the user by using his emotion and desires.

3. Methods and Techniques

3.1 Reflection

Acquiring information of the world around an object in 3D space can be quite difficult task, but at the same time it will most certainly give a high impression of details and realism. There are a few ways to reflect the world into the overall color output when rendering an object.

3.1.1. Cube map reflection

This is probably the most inexpensive way of achieving reflection, regarding computational time. It was proposed in mid 1990s [37] and the idea behind, is to use reflection vector field that would generate spherical UV coordinates [38] for performing texture mapping. This process is also known as environment mapping [39]. The method can be divided into two stages, pre computational and RTC. The first one includes a generation of the cube map (a 360^{o} image, which is then

sampled into six separate textures that can form a cube; this is done in order to perform optimization as a cube can be constructed by the use of 6 quads, or 12 triangles, while a sphere would require much higher number of faces). (see Figure 3.1) The edges of the cube map are then smoothed out in order to have smooth transition between face to face. As regarding the RTC, a calculation of the reflection vector using Equation 3.1, is performed so that run-time environmental mapping is induced.

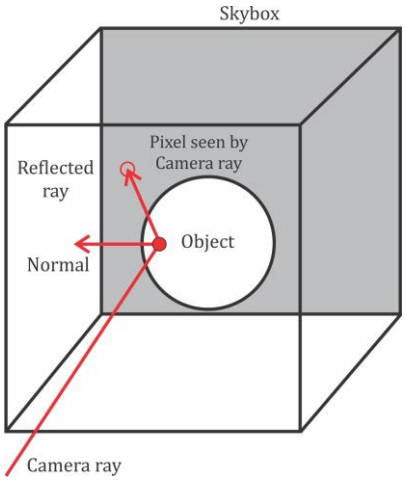


Figure 3.1.: Scheme of the process of cube reflection, using the camera ray vector onto the normal vector in order to find the reflected ray and the pixel see by the camera ray.

In order to calculate the reflection vector R we need to perform ray tracing (see Figure 3.2). First we project the induced camera ray vector I onto the surface normal vector N, Equation (3.1). Based on assumptions like head-to-tail method for vector estimations, Equation (3.2), and the fact that the dot product of two perpendicular vectors is equal to zero, Equation (3.3):

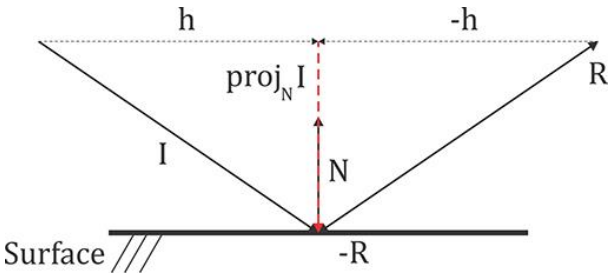


Figure 3.2.: Illustrating the reflection vector - the process of ray tracing by projecting vector I onto the normal vector N.

$$\boldsymbol{proj_N I = c * N} \quad \{\boldsymbol{c} \in \boldsymbol{R}\} \tag{3.1}$$

$$\boldsymbol{h = I - proj_N I} \tag{3.2}$$

$$\boldsymbol{h \cdot N = |h| * |N| * \cos(90^{\circ}) = 0} \tag{3.3}$$

We can then substitute the variables and we find the scaling factor of N that is used to define $\text{proj}_N I$ as well as $\text{proj}_N I$ itself:

$$\rightarrow \boldsymbol{c} = \frac{\boldsymbol{I} \cdot \boldsymbol{N}}{\boldsymbol{N} \cdot \boldsymbol{N}} = \frac{\boldsymbol{I} \cdot \boldsymbol{N}}{|\mathbf{1}| * |\mathbf{1}| * \cos(\mathbf{0})} = \boldsymbol{I} \cdot \boldsymbol{N} \rightarrow \boldsymbol{proj_N I} = \boldsymbol{I} \cdot \boldsymbol{N} * \boldsymbol{N}$$

Since the inverse reflection vector $-\mathrm{R}$ has the same projection onto N we can again substitute and get our final formula that presents the reflection vector using I and N, Equation (3.4):

$$-R = proj_N I - h = proj_N I - (I - proj_N I)$$

$$-R = I \cdot N * N - (I - I \cdot N * N)$$

$$\boldsymbol{R} = \boldsymbol{I} - \mathbf{2} * \boldsymbol{N} * (\boldsymbol{I} \cdot \boldsymbol{N}) \qquad (3.4)$$

Although this type of reflection is widely used even nowadays, it has some downsides and the most noticeable one is the fact that the reflection does not take into consideration objects that inside the scene, meaning that the reflection is only restricted to what the cube map holds as texture colors, see Figure 3.3.Therefore this type of reflection is categorized as pre computed reflection.

Figure 3.3.: Cube map reflection rendering - a) A simple rendering of a plane with Cube map Reflection on the left; b): Early version of the water used in the popular RPG game Elder Scrolls IV: Oblivion on the right.

3.1.2. Scene reflection

A much more sophisticated approach that produces higher "lifelike" results is achieved when everything in the scene is reflected back into the surface, of course at the cost of considerably higher computational time, since everything that will be reflected needs to be computed at run-time. In order to achieve this effect, we need to have something that will secure us run-time input of the reflected image color

value [40] or in other words a second camera view point [41]. This basic idea is that we will reflect everything around the water plane and will return the captured values back into the water surface [43], see Figure 3.4.

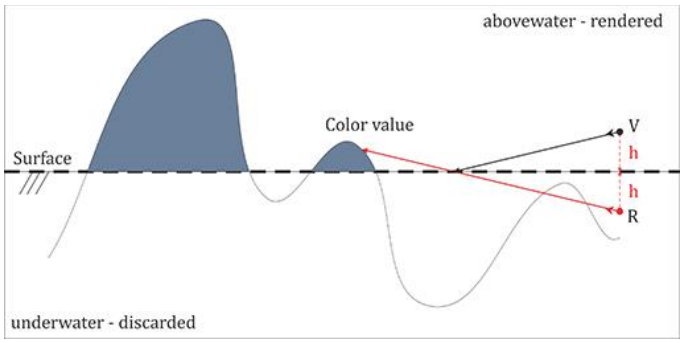


Figure 3.4.: Vector scheme of scene reflection - using second camera view point to capture everything above the water plane that has to be reflected.

If we consider a ray casted from the camera view point V, which is at height h above the water surface, we can reflect V around the surface ($ref_S V$) to find the direction of the reflection vector R. Reflecting point around axis is quite simple but in our case we need to solve a much more complicated problem, since the simulation is taking place in 3D space, where all the combinations of translation, rotation and scale should be considered. In computer graphics all of this is packed in matrices and matrix multiplications (usually to solve vertex transformations) but in this case we need to calculate the so called reflection matrix that will "transform" our camera position into exact reflected one around the water plane. This can be done using the Householder reflection [44] formula, Equation 3.5, where A is the reflection matrix, I is the identity matrix and N is the three dimensional $[x, y, z]$ normal vector of the reflection plane.

$$\boldsymbol{A = I - 2NN^T} \tag{3.5}$$

In order to get the normal vector we need to look into the definition of the plane – in Euclidian geometry a plane can be described by a number of ways – for example by three non-collinear points or a line and a point also non-collinear. If we are to describe a plane mathematically with an equation (point-normal equation of a plane [45]) where $N = (a, b, c)$, Equation 3.6:

$$\boldsymbol{ax + by + cz + d = 0} \tag{3.6}$$

We can now substitute and calculate our reflection matrix, Equation 3.7:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - 2 * \begin{bmatrix} a & b & c \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$\rightarrow A = \begin{bmatrix} 1-2a^2 & 0-2ab & 0-2ac \\ 0-2ab & 1-2b^2 & 0-2bc \\ 0-2ac & 0-2bc & 1-2c^2 \end{bmatrix} \quad (3.7)$$

There is, however, one small issue with this type of reflection. Everything captured by the reflection view point will be rendered, which could lead to possible occlusions (Figure 3.4). To prevent this we need to discard the rendering of everything below the water plane, or in other words we need to clip all the geometry.

3.2 Refraction

The physical event that occurs when we see reflection is explained in nature, by the light that bounces off of the reflection surface. But not all the light gets reflected – some of it passes through the surface – phenomena known as refraction. In computer graphics we can easily achieve this effect by working a solution with transparency, but this will not provide us with the power to affect what is seen underwater and will simply result a blending color. However, there are a few methods to achieve refraction:

3.2.1 Clipping plane refraction

As mentioned in the previous section, we can use rendering of the scene to create reflection. We can execute the same algorithm, but this time without creating/changing/reflecting the camera position. We can simply use the current render frame and clip everything that is above the water and store this color information in a texture that will later be projected onto the water plane [46]. This is one of the most convenient and popular ways of executing run-time refraction, see Figure 3.5.

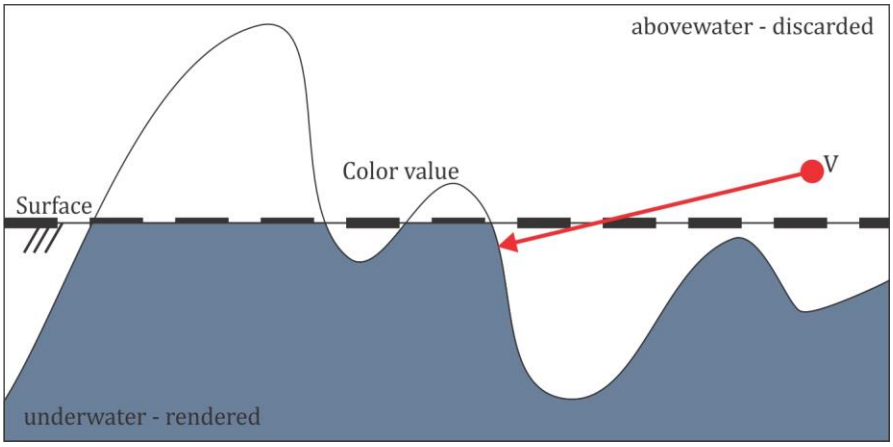


Figure 3.5.: Illustration of scene refraction using second camera view point (vector V) to capture everything under the water

3.2.2 Cube map refraction

As the name implies, this is pre-computed approach where, logically corresponding technique described in cube map reflection is used. We perform a cube map texel

lookup[46] and we map its color information to the corresponding fragment. Again we need to calculate the corresponding refraction vector so we can define proper UV set-up for performing this lookup. Using the Snell's Law, which states that when light passes from one material to another (assuming they have different density) the light's direction changes, resulting in a refracted light/image or in computer graphics context – fragment color. Given the Snell's Law Equation 3.8, it describes how the incoming view ray I is refracted/changed, when passing through a border between materials with refraction indices η_1 and η_2, into refracted vector T, see Figure 3.6:

$$\eta_1 \sin\theta_I = \eta_2 \sin\theta_T \quad (3.8)$$

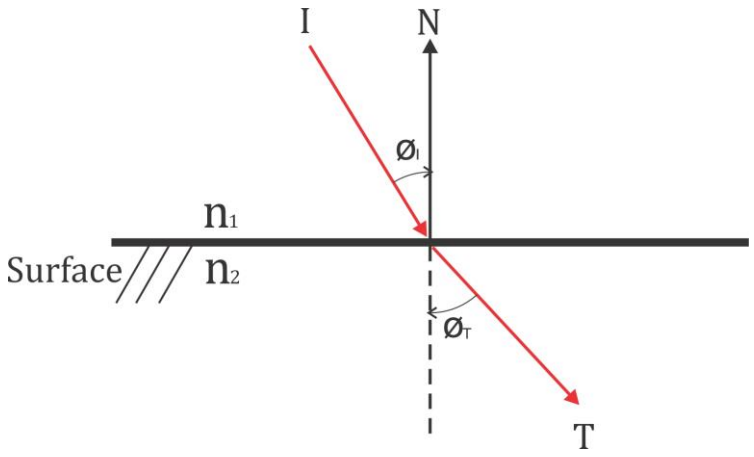


Figure 3.6.: Vector scheme of cube map refraction, where η_1 is air and η_2- water.

3.3 Chromatic Dispersion

A side phenomena/effect that occurs when light passes through the water surface, is the so called chromatic dispersion (from optics, different wave frequencies have different phase velocities). As Snell's Law is also applicable to velocities this leads to light colors being refracted at a different angle when passing through the refractive media [47]. Since in computer graphics our final output is displayed on a screen matrix composed by RGB LEDs, if one is to simulate a dispersion effect, only the refractive indices of Red/Green/Blue are needed and refraction image needs to be calculated once for each color channel using the corresponding refractive index, see Figure 3.7.

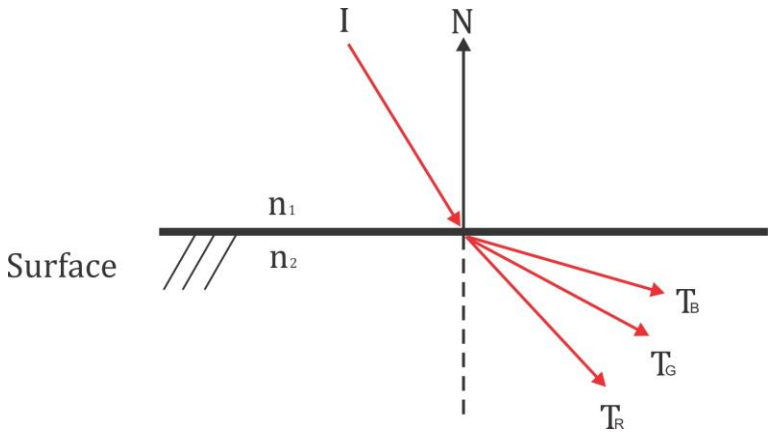


Figure 3.7.: Vector scheme of chromatic dispersion, where T_R, T_G and T_B represent the refractive indices of the tree color channels - RGB.

Extensions of the chromatic dispersion are the caustics and the Crepuscular rays (Figure 3.8), again explained by the refraction of light. When light waves are refracted there is a chance that at a certain place where light is projected (after refraction), the light rays can create a concentration area of light. Creating caustics in RTC is challenging task, which is beyond the scope of this project and it is usually preferred to be faked, rather accurately simulated, as it requires considerably high amount of operational memory. As regarding the crepuscular rays, more commonly known as "god rays" in game development [48], they are a result of high contrast between light and environment and are usually used in games to enhance the realism of the environment as playing part of simulating the impression of volumetric light (which can sometimes be seen when the observer is situated underwater and the light scatters in the water depth).

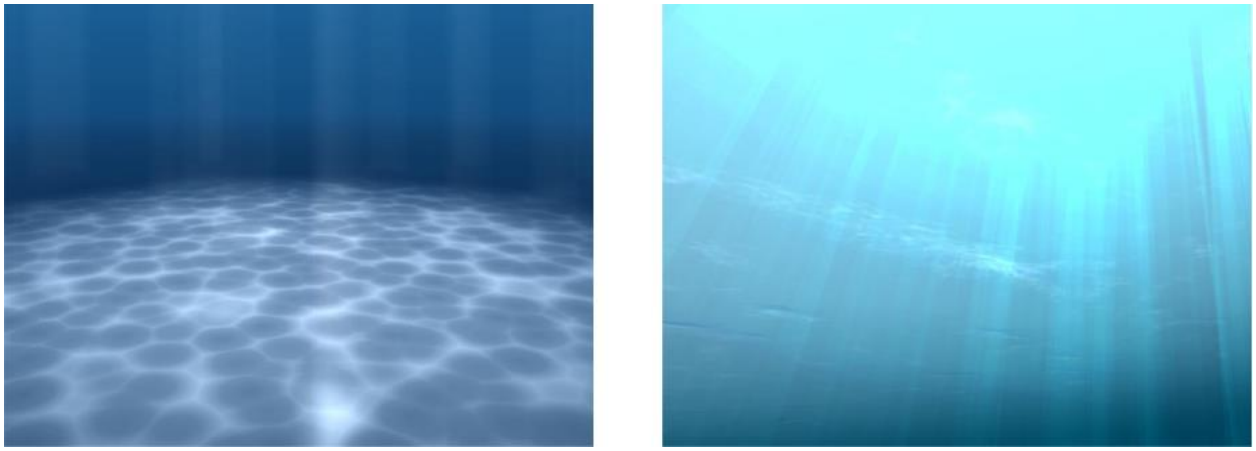


Figure 3.8.: *Water additional FX: a) Image of water caustics made by Alastair Aitchison, on the left side. b) Image of crepuscular rays in water for improving the realism in games, on the right side. Image is made by Frank Kane*

3.4 Clipping algorithms

To perform the more sophisticated methods described in the previous two sections, namely RTC reflection and refraction, we need to clip some of the geometry. Generally, the process of discarding fragments (not rendering them) can be achieved by having a certain test that

those fragments are tested against and if the testing returns a “pass” value the fragment is rendered on the screen, else it is not. The most common test is the depth test, with which opaque objects are drown on top of each other.

Another useful test used for discarding fragments is the stencil test that makes use of the stencil buffer. As some other buffers used during the computer graphics pipeline, this one usually contains one integer value within the range of 0-255 for each pixel to be rendered [49]. The value can then be used for comparison testing of whether the fragment drawn belongs to the water or not, assuming that the water plane itself has written a reference value for all of its rendered fragments [50] (see Figure 3.9).

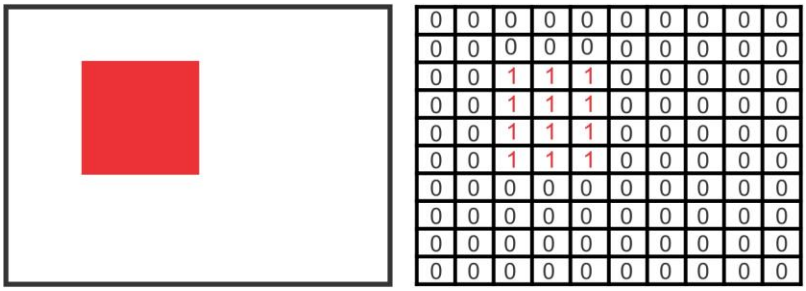


Figure 3.9.: Semantics of the stencil buffer – each rendered fragment can write/read a value inside the stencil buffer

Nevertheless, using the stencil testing could be quite an efficient way, as the stencil testing only calling one additional function – comparison between integers. However, there is even more simplified approach that only changes one of the stages inside the graphics pipeline – applying a transformation to the projection space [51]. The projection space is defined by the camera frustum in order to apply clipping of geometry that is not “seen” by the camera, thus the definition of the camera frustum is a volume space, in which only geometry that is inside that volume will be rendered on the screen, else discarded (or clipped in case of being partially inside). Frustum volume is usually a truncated pyramid, which can be characterized in a few different ways, by a corresponding set of parameters [52] (also used later for forming the projection matrix):

- **Standard perspective projection**

  It uses θ_{fovy} , which is the field of view angle in y direction; the distance n from the camera space origin to the near clipping plane; the distance f from the camera space origin to the far clipping plane; aspect ratio α of the near clipping plane, as shown in Figure 3.10.

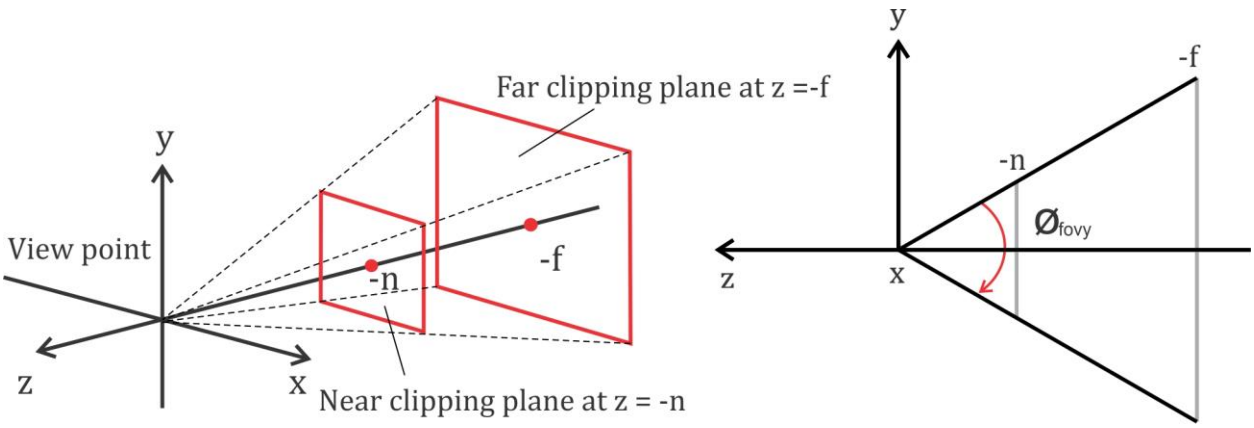


Figure 3.10.: Standard frustum volume definition - a) On the left - Illustration of the near and far clipping plane, forming the frustum. b) On the right, illustration of the angle θ_{fovy} specifying the field of view

- **Oblique perspective projection**

It uses the same distances n and f from the previous set of parameters, but this time the near clipping plane and its relation to the far clipping plane is not described by the θ_{fovy} and α, but rather with specific coordinates of the near clipping plane, namely r (right), l (left), t (top) and b (bottom), as seen in Figure 3.11.

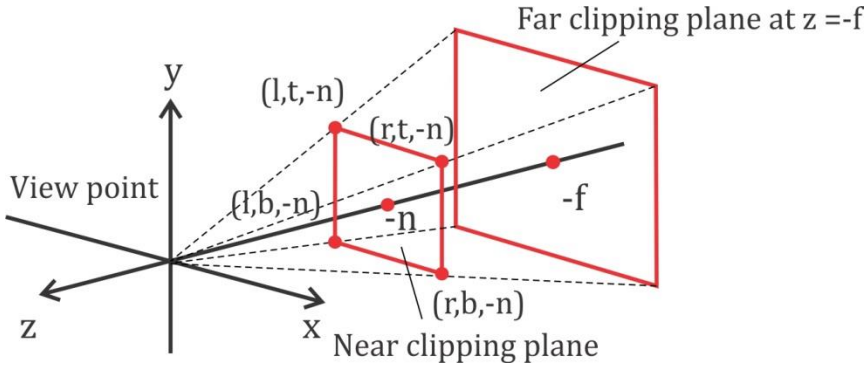


Figure 3.11.: Oblique frustum volume definition – using the four points with coordinates based on $r; l; b; l; -n$

- **Orthographic projection**

The same as the previous one, except this time the frustum space is a box rather than a truncated pyramid (leading to no perspective changes when perspective division is applied later in the graphics pipeline, which is beyond the scope of this project).

In the scenario of creating (both reflection and refraction) for water the clipping can be achieved by setting up the projection frustum to do the clipping of the unnecessary items (logically underwater for reflection, above water for refraction) as described in [51]. Since the water is represented by a plane, the changes to the projection frustum should be so that the near clipping plane matches the water plane, see Figure 3.12.

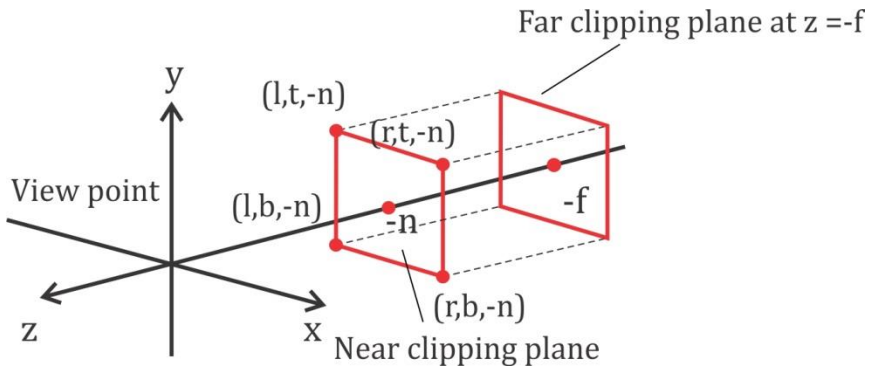


Figure 3.12.: Orthographic frustum volume definition – projection applies no perspective division

3.5 Fresnel effect

When light reaches a boundary between two materials is either reflected or refracted, thus the amount of both reflected and refracted light is given by the Fresnel effect equations. In the case of computer graphics, the amount of reflected image color fragments seen in respect to the refracted ones, based on the viewing angle [53]. In other words, the Fresnel term provides the reflection-refraction light ratio and simply explains the effect that no reflection is seen when looking straight down, towards the water surface, and logically no refraction when the eye horizontal line matches the water surface, described as "simplified or approximated solution" as seen in Figure 3.13. [54]

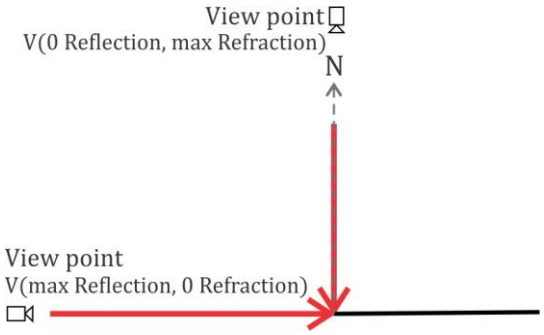


Figure 3.13.: Vector definition of Fresnel effect when Reflection is 0 and Refraction maximum and vice versa.

Having a completely realistic model that explains the calculation of the Fresnel equations is including all of the properties of waves. For example, polarization of waves used to define the reflectance coefficient of materials - R but this will not be researched in this paper. Despite that, there is a method that does "a realistic compromise" to that physically accurate model known as Schlick's approximation [55], Equation 3.9 and Equation 3.10, lately described and used in [56], with the main difference from the previous "simple solution" that Schlick's approximation (Equation 3.9) takes into consideration not only the angle but also the refraction indices of the two media:

$$\boldsymbol{R(\theta) = R_0 + (1 - R_0) * (1 - \cos\theta)^5} \quad (3.9)$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \quad (3.10)$$

, where θ is the angle between the camera viewing position V and the half-angle H between the incoming light and reflected light (thus $H \equiv N$, where N is the normal of the surface); n_1 and n_2 are the corresponding refraction indices of the two materials; R_0 is the reflectance when light is coming parallel to the water surface (in other words, when viewing straight downwards, or parallel to the water surface normal).

3.6 Flow map

As described earlier in section "Fluid water", there are two approaches towards simulating water – either with a surface (procedural water) or a particle based simulation. The later one allows more realistic implementation of real-life events, since each individual particle can be accessed and proper physical behavior can be applied to it. When the motion of a fluid is to be simulated, commonly referred to as Computational Fluid Dynamics (CFD), a model that describes the motions is needed. In most cases that model is described by Navier-Stokes equations – an equation with multivariable functions, so that it can lead to calculating multiple equations that derive from the main one (partial differential equation – PDE). Solving the Navier-Stokes equations, make use of fluid properties like density of the fluid ($\rho = m/V$), pressure ($p = p(t, x)$), velocity of the fluid for each particle $x = [x_x\ x_y\ x_z]^T$ and etc. All these properties would lead to inducing a realistic simulation model of the fluid.

In the core of creating dynamics of the fluid is the movement, partially described as a velocity field. In the case of surface water there are two main areas that dynamics can be implemented – the vertex movement and the fragment movement. The last one is mainly deriving from the assumption that each fragment can be either scaled of offset. In computer graphics when a mapping between UV coordinate and texture element (texel) is applied, there is ability to either stretch or translate the mapping, as seen in Figure 3.14.

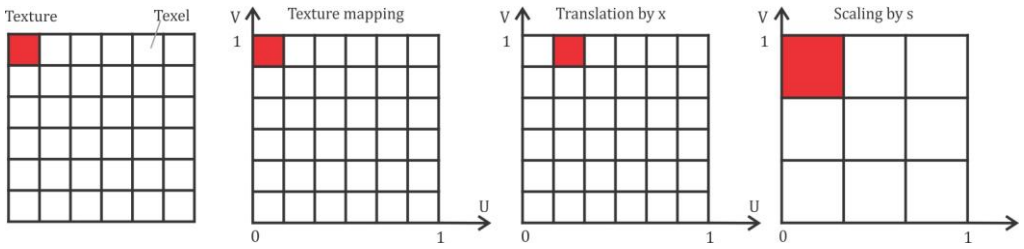


Figure 3.14.: Illustration of a) b) the process of texture mapping, c) translation of the mapping by x, where x is the distance between the UV coordinates and d) scaling by s.

Initial ideas and implementations were done via procedural texture animation [57], commonly known in computer graphics context as scrolling texture, where the UV mapping applied to a model is such, that all the texels are animated. Or more properly said, all the corresponding (u, v) are translated over time, visually resulting in a directional scrolling texture. However, this technique brings not so realistic results as all texels are moving in

one direction. The technique that creates higher realistic visualization of individually moved pixels is called Flow visualization [58]. Flow visualizations take consideration of advection – a velocity (flow) field that basically describes the fluid's motion mathematically using vectors (vector field). The carried/ translated value is called scalar field – in the case of texture mapping, that would be the color value of the corresponding texel without applying advection. Simplified explanation is that a fragment can be translated or scaled in reference to an input velocity field. Considering Equation 3.11, point P(u, v) is a point from the surface, then P can be transformed via the velocity field $V(x, y)$ over a period of time delay t into point P_T=$F^t(P)$, where $F^t(x, y)$ is a mathematical representation of the flow field [58]:

$$\frac{dF^t(x,y)}{dt} = \boldsymbol{V}(\boldsymbol{F^t}(\boldsymbol{x}, \boldsymbol{y})) \qquad (3.11)$$

However, there are two main concerns with this technique. The flow mapping is done into a finite area R (the UV coordinate space of the model). Meaning that the movement of the fragments will end at some point, where there is no flow field data to continue the flow mapping. When no flow mapping is implemented, texture UV mapping is usually done via either repeating the UV coordinates or clamping the last known pixel value and as mentioned in [58] the same can be applied here – either extrapolate the velocity field outside R, or continue with the last know velocity value. The corresponding advected texel value has the same problem since we do not know the color value outside R. The standard solution [58] is to use a looping movement defined by a periodic function, for example $\sin(t)$. Using periodic functions like $\sin()$ or $\cos()$ could lead to non-linear visual results (slower animation at both ends of the period, in oppose to faster animation during mid-period). The use of linear functions would give constant flow/advection when solving the extrapolation problem, also called Smoothly Interpolating Layers [59], see Figure 3.15.

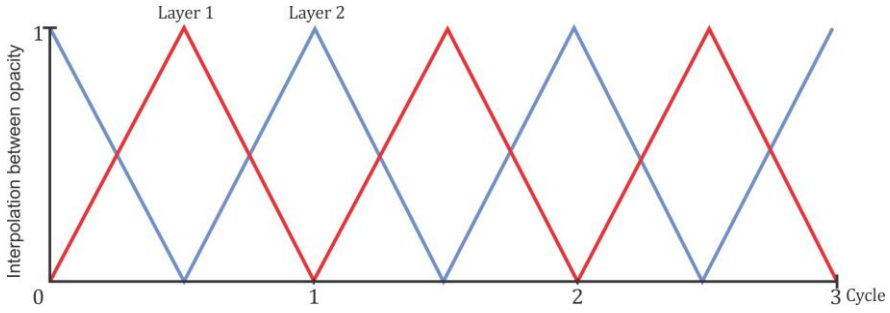


Figure 3.15.: Looping function for flow mapping - Interpolating layers by changing the opacities of the two layers.

Since the entire method is implemented for enhancing surface renderings and makes use only of textures and texture mapping, the technique is divided into two stages:

- **Preparation stage**

Generating of the flow vector field is done prior the animation, in order to artistically determine the flow animation direction, based on terrain referencing or static obstacles in the water. Generating a flow vector field requires to create data field V, containing all

velocities $V(x, y)$, thus we need a memory container to hold that information. In the sense of working with data, textures are represented by their width and height properties forming a texel grid, whereas for each texel a *float4* value is created, holding the corresponding value for each RGBA texture channel. Therefore, for creating a flow vector field, the first two channels can be used, as shown in Figure 3.16. If we consider that vector $v_0(x_0, y_0) \epsilon F$, where $F(r, g, b, a)$ represented flow texture, then $x_0 = r;\ y_0 = g$.

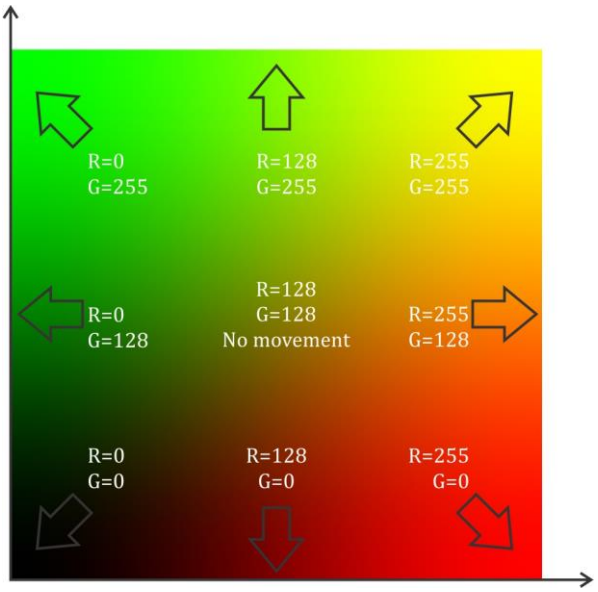


Figure 3.16.: *Flow map directions - using only the red and the green channels, or their corresponding float values r and g.*

There are a number of external software tools that can be used for creating proper flow field textures, for example Houdini [Reference], but the main principle behind all of them is to interpolate color values based on created vector field in respect to orientation in the RG color space. Color and other values passed from the vertex shader to the fragment shader, are usually linearly interpolated, using a weight property $w \in [0; 1]$, that specifies the percentage of value a and value b that is being used to fill in the missing values in-between a and b in Equation 3.12, (see Figure 3.17).

$$\boldsymbol{I_{linear} = a + w * (b - a)} \qquad (3.12)$$

Figure 3.17.: Linear interpolation - If $w = 0$, the interpolated value would return full weight on a, opposite to $w = 1$ resulting in full weight on b

It is important to note that, in order for the linear interpolation to produce color image, a set of at least three vectors need to be created exactly the same time as in rendering a triangle on the screen, in order for the interpolation to give values for texels in an area, rather on a line. (see Figure 3.18 a)). In the scenario of linearly interpolating

data inside a triangle – the weight of the triangle is used, which is beyond the scope of this project.

Another more convenient approach towards interpolating data, resulting in even more proper final output image, used as vector field, is the use of bicubic interpolation (see Figure 3.18 b)). It will take into consideration all the values from all the advection vectors and will produce a smooth continuous interpolation between their colors. Here vector's color means the RG values corresponding to the vector's x/y-axis orientation. The area p or the surface, where the data can be interpolated, using bicubic interpolation, can be presented by Equation 2.13. It is used to smooth out neighboring linear interpolations (which number is n^2(cubic)).

$$\boldsymbol{p(x,y) = \sum_{i=0}^{n}\sum_{j=0}^{n} a_{ij} x^i y^j} \qquad (3.13)$$

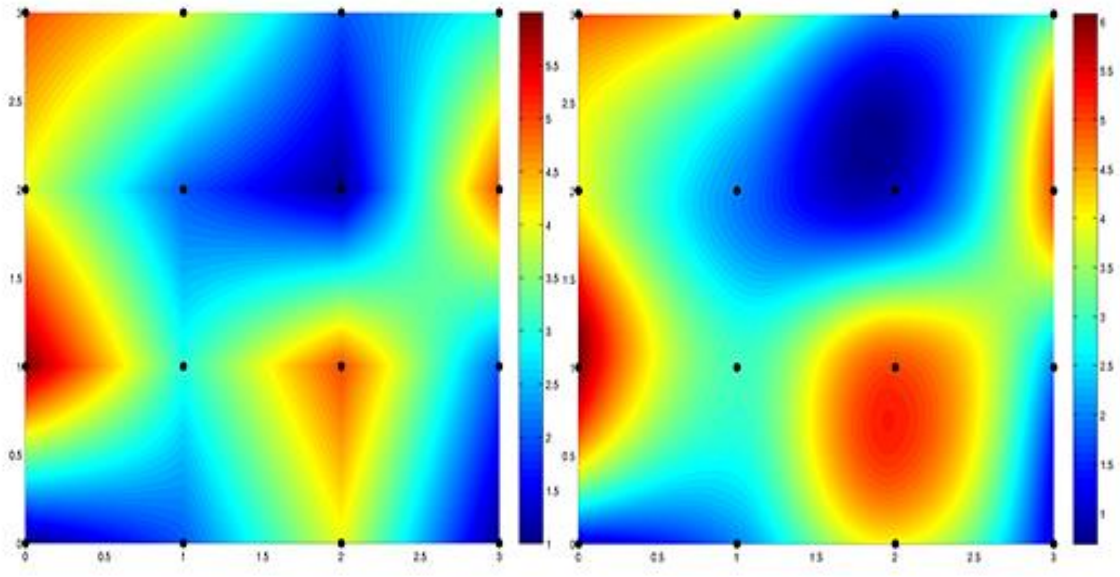


Figure 3.18.: Multidimensional Data Interpolation - a) Illustration presenting bilinear interpolation and b) bicubic on the right.

- **RTC stage**

The generated flow map can be implemented in the fragment shader in order to produce moving flowing UVs, which can be used for animation of desired coordinates. An example is a simple water texture or more advanced – normal maps, later used and described in light calculations.

3.7 Light

In computer graphics light has a similar meaning to the light in real-life – it allows things to be seen or not, as well as it creates shadows. Note that in computer graphics light and shadows are different topics. Simulating light can, like anything else connected with

computer generated content, be either RTC or pre-computed. (see Figure 3.19) The last one is usually used when creating virtual lit scenes for movies or animations, as it allows building the light model upon light properties like light photon tracking and other more complicated physical attributes, resulting in a high-precision model, of course at the cost of great computational time needed, therefore this type of lighting simulation is not used in real-time applications. In contrast, RTC simulation model is entirely used in real-time applications, and sometimes also in pre-computed for creating fast prototypes. RTC light takes into consideration a small fraction of light properties enough for explaining light in a fast computational design. These properties are later used within the shading process of the objects, inside the CG environment, to create a no entirely accurate light model. Internally, there are many differences between the two types of simulating light, but generally the output of a RTC light is sophisticated enough to create high believability impression in the user.

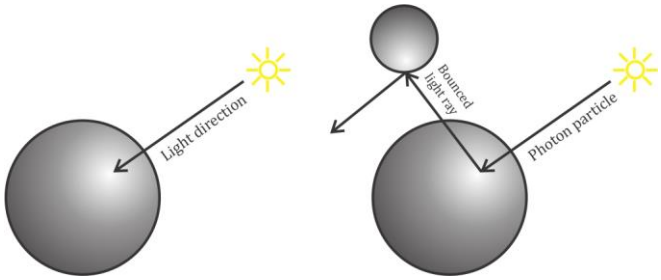


Figure 3.19.: Light computation models - a) RTC lighting simulation on the left and b) pre-computer lightning simulation on the right.

3.7.1 Light sources

The computer generated environment can be lit. It can be a number of different types of lights, usually in RTC is used directional, spot and omni. Each type of virtual light has a meaningful explanation and usage. There are, however, a few common properties used across all light simulations, namely intensity of the light, color of the light and position in world space. Apart from these, each individual type of light has some specific characteristics:

- **Directional light**

  As the name implies this type of light is only consisting of one directional vector and the objects that receive light are hit equally throughout their light exposed surfaces. Meaning that, the light has no fall-off when creating shadows or in the context of light simulation – attenuation [60]. Light is produced by a light source and usually in RTC it has infinite distance from the scene. Directional lights are often used to substitute sun light. (see Figure 3.20 a))

- **Spot light**
  Oppositely to the directional light, this type is a geometrical explained light, meaning that the light starts from the light source and forms a cone. The radios of the cone

can be used as attenuation parameter - the unequally distribute light in the lighting model [60] (falloff based on distance from the center of the cone base). It is usually used to create focus lights in scenes, for example car headlights. (see Figure 3.20 b))

- **Omni light**

  Omni light or more commonly known as point light, is such that light starts from the lighting source and is spread equally outwards in all directions. [60]This type of light can also make use of attenuation (falloff based on distance from the lighting source). It is usually used to create dynamic environmental lights – for example a torch-light. (see Figure 3.20 c))

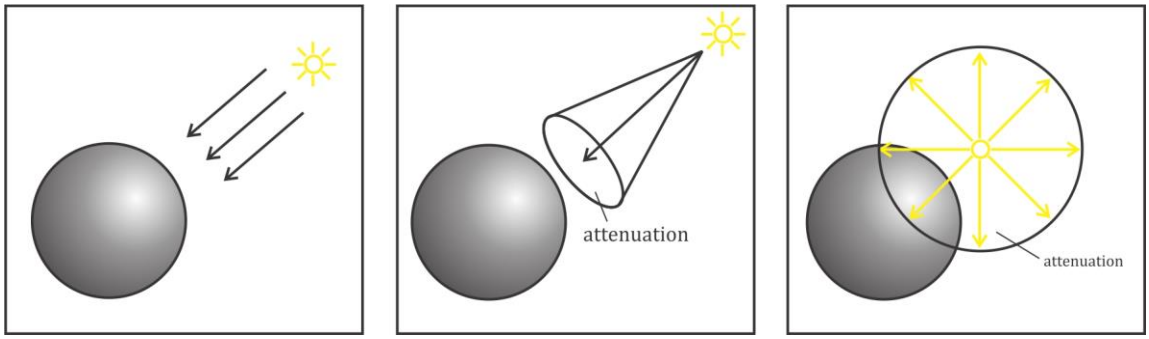


Figure 3.20.: CG environment light types - a) Directional light; b) Spot light; c) Point (Omni) light.

3.7.2 Light categorization

When proper type of light is placed in the scene, it is not directly used to illuminate the computer generated models inside the scene. The approach is to instruct the models how to receive light in their rendering process – more commonly referred to as shading. Applying shading, on its hand, leads to the development of lighting models, as mentioned earlier, which are responsible for changes that will occur in the final output, based on parameters taken from the light and its computation. Lighting models can be categorized based on different parameters:

- **Smoothness** – based on the interpolation between the different triangles in a model we can define the following shading models, Figure 3.21:
  - Flat shading – when no interpolation is applied, resulting in the use of the triangle's normal, in order to calculate the color change that would be applied to the final output. In other words, the entire triangle will be rendered with only one color. This type of shading is usually used when fast computational times are required (for example, edit mode in used inside 3D software packages)
  - Gouraud shading – when a simplified interpolation is applied only for the color fragments, forming the triangle. The shading changes the color only at the vertex level, meaning that the light is computed, based on the each vertex normal and the light direction.

- Phong shading – the same as Gouraud shading, except this time the vertex normals are also interpolated, in order to create interpolated data for each fragment, later used to calculate the shading changes applied to the final color output.

Figure 3.21.: Smoothness light types – flat model on the left; Gouraud in the middle; Phong on the right

- **Reflectance** – It is based on the surface property of how light is reflected from the surface.
  - Lambert – when the reflected light only includes diffuse reflecting light or the shading applied, only based on how much the surface is looking towards the light, resulting in an evenly reflected light in all directions. In real life, this type of reflectance is called "matte", meaning that there is no bright (specular) spot. If we consider Equation 3.14, where direction L is representation of the light rays coming from the light source, and N to be the normal vector, pointing perpendicularly out of a surface S, we can then estimate the angle of light falling onto the surface. Simply said, how much the surface is rotated towards the light direction, as seen in Figure 3.22 (note that L and N are normalized, since only their direction is compared):

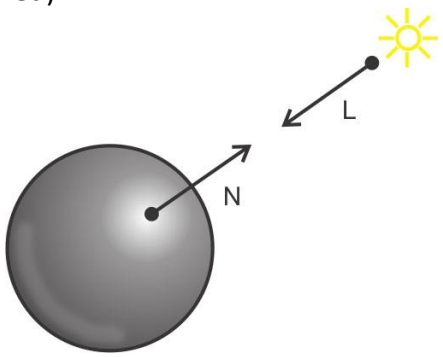


Figure 3.22.: Lambert Reflectance – only the surface normal and the light direction vectors are considered

$$\boldsymbol{L \cdot N = |L||N| \cos \alpha \ = \cos \alpha} \qquad (3.14)$$

The result from the dot product is referred to as diffuse factor D. If we consider tex to be the color of the fragment based on either texture element or texture element and so other color changes, and L_{color} to be the color of the

light, the final color output of the fragments, in the boundary of the surface is given by Formula 3.15:

$$\boldsymbol{output_{color} = L_{color} * tex * D} \quad (3.15)$$

- Half-lambert – this is an extended shading model proposed and integrated by Valve. Since the $\cos\alpha$ returns a value $\in [-1, +1]$, the polygons of the model not facing the light can end up having too darker tone changes, leading to the impression of a flat surface. As seen in Equation 3.16, the diffuse factor D is scaled by $^1/_2$ and squared in order to move the return value of $\cos\alpha \in [0,1]$, resulting in more soft shading changes in the final output, see Figure 3.23.

$$\boldsymbol{output_{color} = L_{color} * tex * (D * 0.5 + 0.5)} \quad (3.16)$$

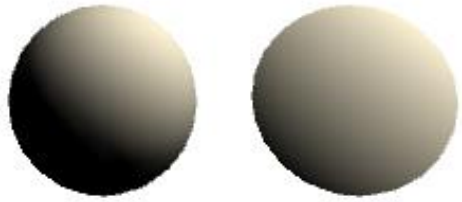


Figure 3.23.: Half-Lambert Reflectance – mid tones are enhanced

- Blinn-Phong - an extension to the Phong shading is to add specular reflection, or the brightest spot on top of the lit object, usually used to explain the shininess of the objects. In computer graphics, the term specular is used to explain the visual phenomena of perfectly reflected from the object's surface light into the viewer's eyes. In order to calculate the specular color, we need to reflected light ray R_{light} and compare its direction to the viewer's direction V, as seen in Figure 3.24. If we take the resulted specular reflection on the power of $n_{shininess}$, we can increase the amount of specularity. R_{light} can be found using the reflection formula in Section 3.1 Reflection in this paper. Let S be the specular factor, and S_{refl} be the specular reflection.

$$\boldsymbol{R_{light} = L - 2N(N \cdot L)} \quad (3.17)$$

$$\boldsymbol{S = (R_{light} \cdot V)^{n_{shininess}}} \quad (3.18)$$

$$\boldsymbol{S_{refl} = L_{color} * S} \quad (3.19)$$

$$\boldsymbol{output_{color} = L_{color} * tex * (D + S)} \quad (3.20)$$

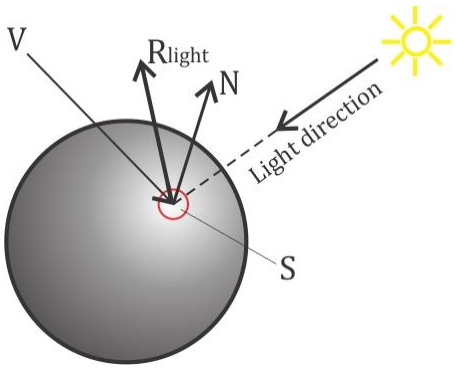


Figure 3.24.: Blinn Phong shading – includes not only diffuse reflection but also specularity

- **Computation** - as mentioned earlier, light computation can be either implemented on per vertex or per pixel lights. (see Figure 3.25) The difference is that, the later one would require considerable higher computational time, yet it would yield better results.
  - Vertex light
  - Pixel light

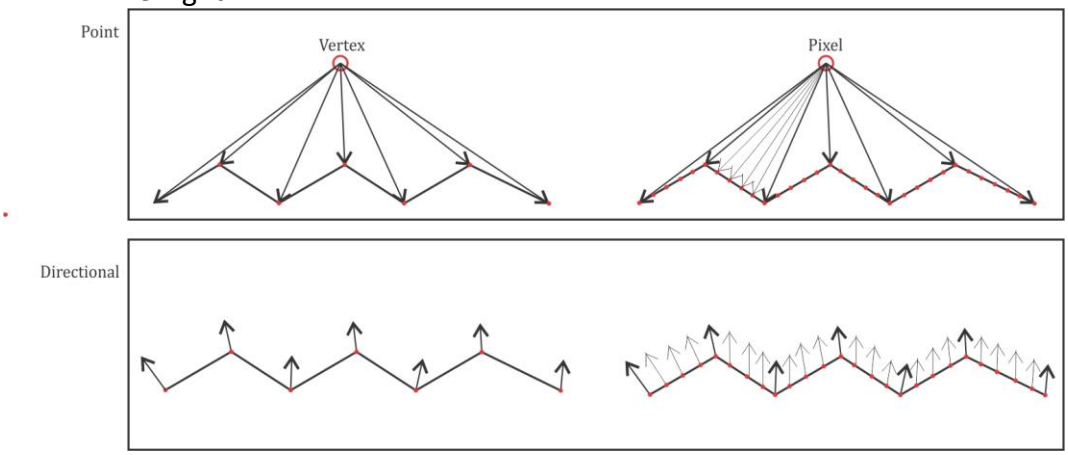


Figure 3.25.: Vertex and Pixel light computation - the first image illustrates Point lighting: one the left per vertex and on the right per pixel. The same is illustrated on the second image, regarding the Directional lighting.

- **Realism** – all light models can also be classified according to how close they are from reality, see Figure 3.26.
  - Photorealistic – for example Phong or Blinn-Phong
  - Non-photorealistic – cartoon shading, where a distinct line is seen between objects lit and unlit parts, or no smooth shading.

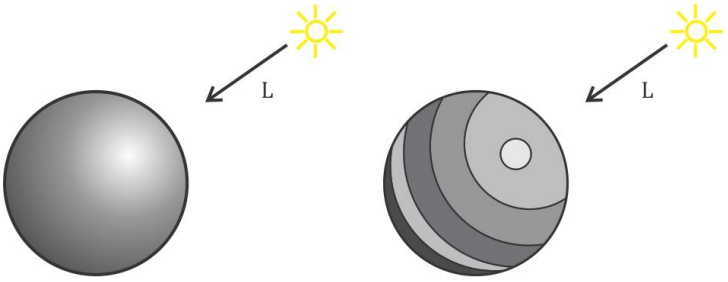


Figure 3.26.: Realistic (on the left) and Non-realistic (on the right) light shading.

3.7.3 Bump Lighting model

Apart from the described lighting models above, there are other ones, rather more complicated, but producing higher results. One particular one involves a technique called bump mapping, widely used to substitute high polygonal models with low resolution ones, where the information of the high polygonal model convex and concaves are stored in a texture. This texture is referred to as a bump (normal) map, from which normals per pixel are extracted in order to substitute the interpolated normal, created in the computer graphics pipeline, see Figure 3.27:

Figure 3.27.: Bump light model – without (on the left) and with (on the right) normal map

Using this technique can allow adding details to the rendering without actual computation of geometry, needed to define these details. The extraction of the custom normals from a texture is again done using channel information, used to recreate normals direction. Thus each individual pixel normal needs decoding, meaning that each component of the normal vector direction needs to be computed. Considering Equation 2.21, where texture Tex holds normal directions $N[n_x n_y n_z]^T$, we can extract two of the channels to define the n_x and n_y, while the n_z can be found assuming that N is normalized direction, meaning its length is equal to 1:

$$\sqrt{\boldsymbol{n_x^2 + n_y^2 + n_z^2}} = \mathbf{1} \qquad (3.21)$$

Since we are transforming the originally interpolated normal into a custom (extracted) one, we also need to apply a transformation multiplication. Generally, the first transformation stage, described in section “Data transformations and coordinate systems”,

states a transformation from local (model) space to world space. The same thing happens to the normal vectors. The local space of the normals is described by three main components, namely the normal itself N, the tangent T to the surface, from where the normal originates and the binormal direction B is perpendicular both to the normal and the tangent. Using N, T and B, we can then transform each custom normal from local to world space and use them in calculating a bump light model, see Figure 3.28.

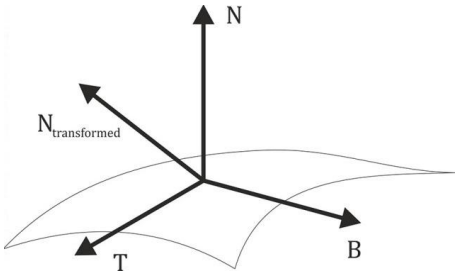


Figure 3.28.: Vector scheme of bump light – computing the reconstructed normal inside the normal vector space

3.8 Vertex displacement

In the section "Flow", we described a technique for translating pixels based on reference texture, defining a flow vector field. However, in order to enhance the water simulation, another dimension of movement/animation can be included. Animating fragments can have good results, but the rendered output (in the case of surface water) will still be flat geometry. In order to deal with this, a vertex displacement can be applied. As for animating vertices, it can be either high or low costly, regarding computational time. The workload can also be computed by either the CPU (vertex position changes before rendering) or the GPU (vertex position changes inside the vertex shader).

There are a number of ways to implement a vertex displacement. In most cases, regarding surface water, the vertex animation is procedural. The simplest solution is to use looping, for example sine wave, as seen in Figure 3.29. More complex model can be created by adding sine waves with different amplitude resulting in a variation [19]. Other more complex approaches [61] are connected with the use of tessellation or Fast-Fourier Transformation (FFT), which are beyond the scope of this project.

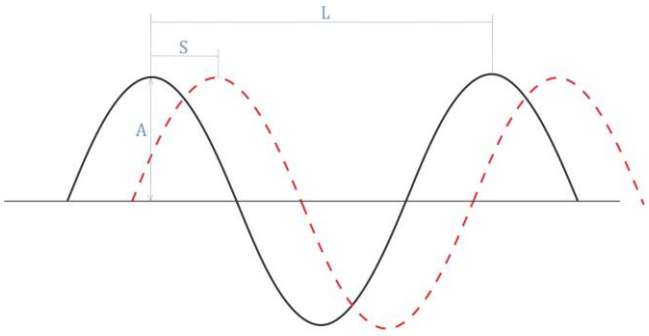


Figure 3.29.: Vertex displacement - L is the wavelength, S is the amplitude and A is the height of the wave.

3.9 Physical models

In computer generated environments not only the rendering is important. Object interaction plays major role in shaping the overall immersion. Here by object interaction is meant the physical events that take place during the simulation. The most common physical event in games for example is the collision and the collision detection, based on a boundary boxes. More advanced physical events involve implementing physical behaviors to the objects in the world, resulting in an explanation of how objects can interact when collision between them occurs. Therefore, a physical model needs to be included to define the water-to-object interaction, or in other words – water physical model. Once more, the amount of difference between CG physical model with lifelike one, is at the cost of computational time.

There are a number of forces that a fluid exerts on an object interacting with the water. The first, and probably most obvious one, is the stream force F_{stream}, meaning that objects coupling with the water inherits the velocity created from the water particles. Friction also occurs whenever objects interact (in the case of water physics, friction is referred to as water drag F_d). Lift force is a complex part of fluid physics, perpendicular to the flow direction L. (see Figure 3.30) Some of these forces are usually ignored in computer simulations for surface water. However, the next force acting upon objects intersecting with water is playing important role, since it provides strong visual cue – buoyancy B [62].

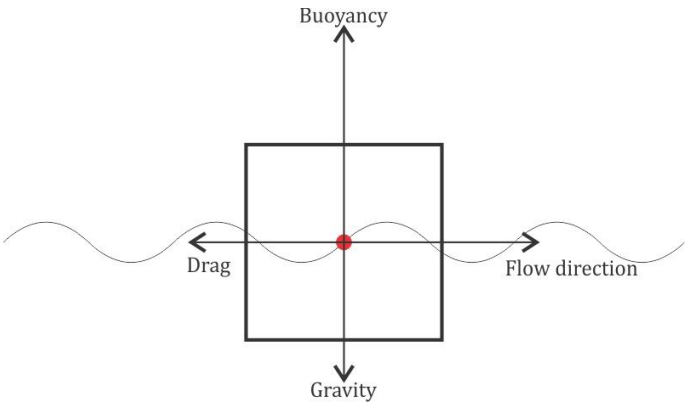


Figure 3.30: Forces affecting the object – buoyancy, gravity, drag and fluid stream

Buoyancy force describes the behavior of why objects float and is based on Archimedes' principle that the buoyancy is equal to the weight of displaced fluid. If we consider ρ_f to be the density of the fluid, g to be the gravitational force and V as the volume of the submerged part of the object inside the fluid, the buoyancy can be given by the Formula 3.22:

$$\boldsymbol{B = -\rho_f g V} \qquad (3.22)$$

Unfortunately, this type of force calculation is not integrating any relevance to the geometrical form of the object. If we consider a cube to be the smallest geometry form that forces can be equally applied to its sides, we can then use an approximation [62] of the geometry model. One such approximation is the algorithm of the Marching cubes [63], used to describe geometry by discrete volume grid (cube grid). Another explanation, deriving from the marching cubes, is the Cuberille Approach [64], in which the geometry to be rendered is transformed into "blocky" surface. Thus, dividing the model geometry into cubes, usually referred to as voxels, (which is only part of the margin cube algorithm, full one will not be used or explained) will allow applying forces per cube, leading to more realistic results at the cost of more computational time, see Figure 3.31.

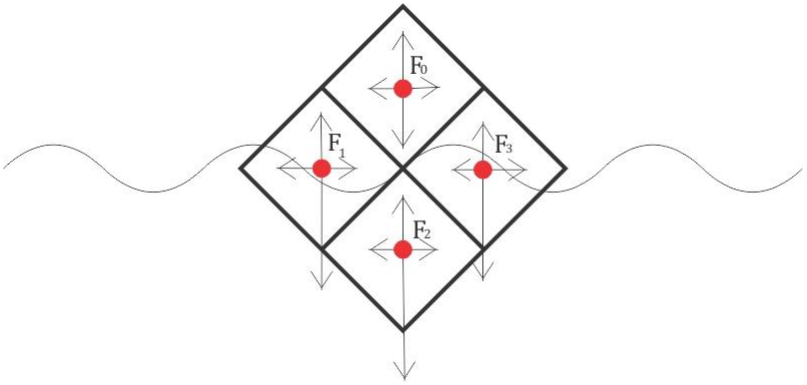


Figure 3.31.: Cube approximation – by dividing the object into cubes (voxels) and applying forces per cube.

3.10 Interaction

As described in the previous Section 3.9 "Physical models", water is affecting the objects by virtue of physical model. But in real world interaction is based on action and a responsive reaction. Therefore, we classify the reaction of the water when objects enter it as object-to-water interaction. Here we can also see a double meaning in the word "object". We can consider interaction to occur when objects within the scene interact with the water, but we can also logically perceive the same situation within the field of Human-Computer Interaction, where the user can interact directly with the CG water via a controller (simplest example is a mouse, but more advanced ones can also be used – gesture interaction). Further in this section we will address the first case as "object-to-water" and the second as "user interaction".

3.10.1 Object-to-water

This topic can be seen in a few different ways, depending on the output reaction of the objects, included in the interaction (object and water surface). It can either be object-to-fluid interaction, fluid-to-object or both ways at the same time [65]. We already discussed fluid-to-object interaction in the section "Physics", since this is more oriented to what happens with the object when it enters the water. In this section we will investigate what can be the reaction of the water itself.

The event in real-life, when a ball hits a water volume is described visually as water being displaced in a matter of water splashes, foam and ripples. In computer graphics, on the other hand, simulating these events could be high costly, regarding computational time. If we are to target RTC, a more fake approach needs to be considered. Usually splashes and foam are easily simulated using particle systems without inner interaction, see section "Fluid water". As regards ripples, situation is a bit more complicated. One possible solution is again using a particle system (emitting particles parallel to the water surface) with animated texture being plugged inside the particle rendering, but this could lead to a number of other issues. A more sophisticated approach is to implement grid interaction based on height field water [66]. The main concept is to create water distortion, based on calculating position of the mesh vertices. If we consider the surface of the water to represented as 2D array $[i, j]$, where each element in this array is a vertex position from the surface, the height field can then also be represented by this array. Therefore, position from the grid can be analogically referred to as a height while the change of the column's height over time is the vertical velocity. Assuming that a change in the height occurs, based on that change at frame X, we can predict the height of the water at every point for the frame $X + 1$ [67]. However, there are two general problematic areas when simulating water ripples. The first one is connected with the fact that there is one height value per vertex, resulting in the inability to create broken surface or in the context of water simulation – breaking wave. The second general problem is that in order for the simulation to work, it needs to iterate throughout all the vertices. Thus, the grid size (or the area of simulation) is of great importance – higher grid size, results in higher computational time. There are existing solutions for these problems [68], but they are beyond the scope of this writing.

Water ripples is challenging task. It is worth mentioning that it can be implemented in both the CPU and GPU. There is ,however, even more simple way of achieving interaction, which is partially described in the Line integral convolution technique, used to visualize the motion of a fluid using texture advection [69] (relatively close to the flow mapping explained in section "Flow map"). The interaction with the texture advection is explained as using "dye injection texture" [70], which is combined together with the advection texture in order to form the advection step, used to translate the corresponding UV position, when mapping is applied later on, see Figure 3.34.

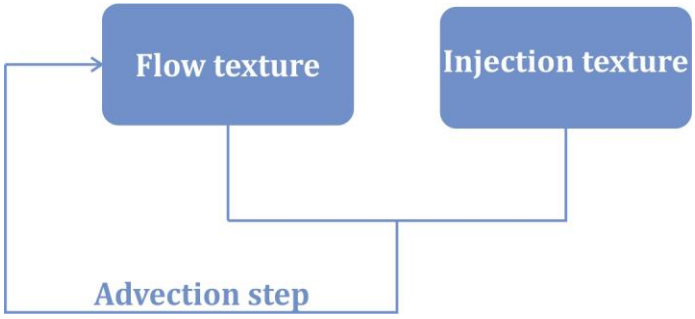


Figure 3.32.: Object to water interaction – based on combining current texture with additional texture

4. Prerequisites

In this section we will list things that will be taken into consideration during the implementation process.

4.1 Concept origin

In the following sections we will implement a number of techniques in order to create water surface rendering with respect to physics, visual and some basic scene interaction that can later be used in the context of real-time applications, such as games or virtual environments. Our focus falls onto the visual aspect of the implementation, as our main interest field in computer graphics. To further complicate the simulation model, we also wanted to create a transition of water states between liquid and solid state. Our main interest in conveying this dynamic RTC animation is in respond to the fact that we couldn't find any noticeable models of such animation. In most cases, water simulation or ice simulation are investigated separately, or otherwise – with no respect towards RTC applications. The initial problematic area, namely water simulation originated during our previous semester internship in DADIU (The National Academy of Digital Interactive Entertainment), where we had to create 2D water simulation for a side-scrolling game. Our proposal is to categorize water simulation in RTC application in respect to a three component model flow, namely visuals, physics, interaction, as seen in Figure 4.1, in which each individual component has unique role in the final composition for creating water simulation.

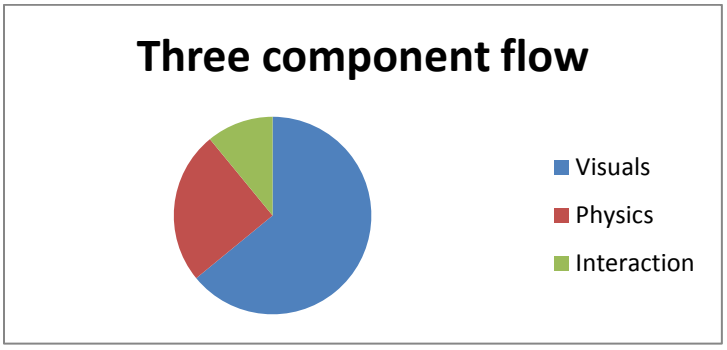


Figure 4.1: Three component flow

4.2 Requirements

As described earlier in Section 3. Methods and Techniques, when designing a model, that will be executed in real-time, the most important aspect of it is responsiveness time as part of the HCI. This project involves creation of CG content, part from HCI. In order to have proper simulation model, we need to make sure our model is accurately following few of Dholakia's six "critical components" [71] when creating interactivity (in our case HCI),

namely responsiveness, real time interactions. The rest of the components are not involved as requirements.

- **Responsiveness**

Responsiveness describes the communication as an act of sending and receiving message, or in other words, when an action is applied onto the product, what would the response be. A lack of response could lead to bad design. In computer generated space/application, common example is the "frozen" screen, where basically nothing happens if no user control is applied, resulting in decreased immersion. Regarding the product of this paper, this is related to the constant animation of RTC water.

- **Real time interactions**

It refers to how fast a communication is elicited. As mentioned earlier, RTC techniques are optimized versions of pre-computational. Still the time needed for the RTC is of essence. In computer graphics, each frame rendering can be measured in μs (milliseconds), thus the frames rendered per second (FPS) is a measurement used to define the communication time in the simulation. Delays or simply said lower FPS value, could lead to undesirable effects.

Based on the above statements and assumptions we have made, we are interested in achieving results that would satisfy the following prerequisites:

- High frame rate (FPS), resulting in smooth simulation.
- Computation focus on the GPU, rather the CPU, meaning that we implement most (if not all) the techniques to be calculated mainly the GPU.
- The final model should consist of techniques from all the three major categories that we described earlier, namely visuals, physics and interaction in order to output full-scale simulation.
- Proper showcasing of the implementation results, inside either a game or virtual environment.
- Sophisticated level of optimization.

4.3 Tools

All implementations will be done inside Unity3D v5.0.0f4 as a graphical and programming environment with the use of Nvidia CG language and C#. We also make use of few external tools for image editing used to prepare some of the textures, used inside the graphical processing. World Machine is used for creating terrain environment. A freeware ALGO Flow Field Editor is used to create flow maps using linearly interpolated vector field.

4.4 Settings

We are deploying a scene/virtual CG environment, where we showcase the use of surface water we developed. The environment is populated with a terrain, with a predefined

landscape. Throughout the terrain, a river passes – main element of the simulation. To produce meaningful screenshots in the following chapter, we make use of checker-board textured spheres, used as distinctive geometry for visual cue and testing of elements like reflection in the water, refraction and etc. The final image output is enhanced by a few image corrections, included in the Unity3D GL – color correction, DoF (depth of field). We have also implemented a few GUI controls to control the simulation – position of the sun, rotation of the camera and buttons for restarting scene events (for example floating sphere).

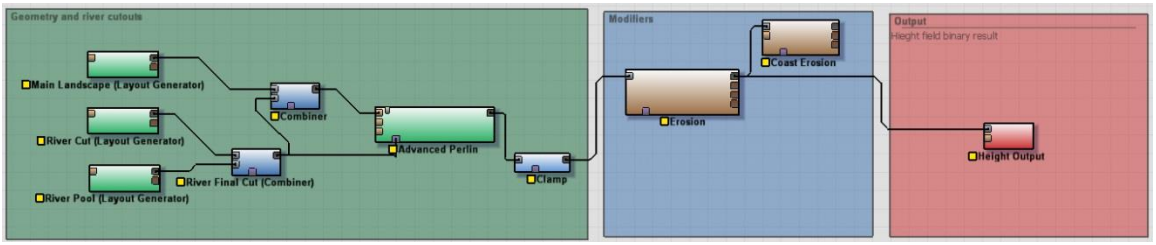


Figure 4.2.: Terrain plan - generating the terrain using World Machine engine.

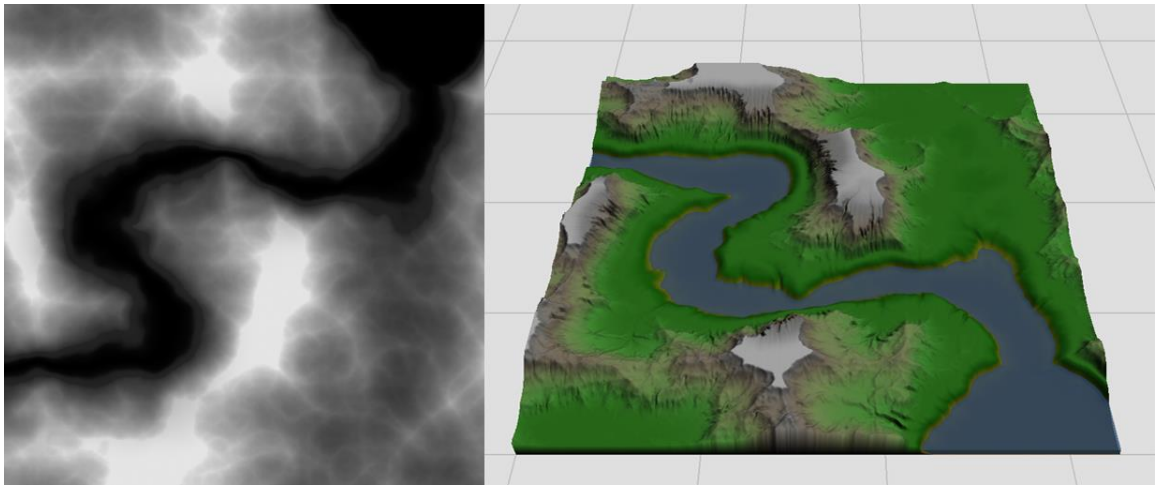


Figure 4.3.: High field map (on the left) and terrain rendering (on the right side)

5. Implementation

5.1 Surface water rendering

The following sections will describe set of techniques, used in order to create as realistic as possible simulation of surface water, whereas each individual technique has its own purposeful meaning.

5.1.1 Reflection

From the options described the research part of this writing, we defined that the best solution is the use of RTC reflections that include scene objects being reflected. RTC reflection is divided into two stages – first obtaining the reflection image and secondly applying that image inside the surface rendering. Both stages are implemented in a specific Unity3D function – *OnWillRenderObject()* – called when the object is going to be rendered (assuming that this object is our reflective surface). In order to do this, we need to execute the following recipe (here reflection plane is used as a substance to the water plane in a more meaningful context):

i. Create a reflection camera that will render the reflected image at run-time, given in Listing 5.1. In order to optimize the process, the reflection camera is used only for creating the reflection image and then it is passed into a cache variable which can later be discarded.

```
1.  Camera CreateReflectionCamera(Camera main)
2.  {
3.          Camera reflCamera = mCameras[main] as Camera;
4.          if (!reflCamera)
5.          {
6.                  GameObject go = new GameObject("Water Reflecion Camera", typeof(Camera), typeof(Skybox),
7.                  typeof(FlareLayer));
8.                  reflCamera = go.GetComponent<Camera>();
9.                  reflCamera.enabled = false;
10.                 reflCamera.transform.position = main.transform.position;
11.                 reflCamera.transform.rotation = main.transform.rotation;
12.                 go.hideFlags = HideFlags.HideAndDontSave;
13.                 mCameras[main] = reflCamera;
14.         }
15.         return reflCamera;
16. }
```

Listing 5.1.: Reflection camera.

We need to make sure that the reflection camera has the exact same properties as the main camera, so that all elements and layers rendered by the main camera are also visible to the reflection camera, given in Listing 5.2.

```
1.  void CopyCameraProperties(Camera source, Camera dest)
2.  {
3.          if (dest == null)
4.          {
5.                  return;
6.          }
7.          if (source.clearFlags == CameraClearFlags.Skybox)
8.          {
9.                  Skybox mainSky = source.GetComponent<Skybox>();
10.                 Skybox reflectedSky = dest.GetComponent<Skybox>();
11.                 if (!mainSky || !reflectedSky.material)
12.                 {
13.                         mainSky.enabled = false;
14.                 }
15.                 else
16.                 {
17.                         mainSky.enabled = true;
18.                         mainSky.material = reflectedSky.material;
19.                 }
20.         }
21.         dest.clearFlags = source.clearFlags;
22.         dest.backgroundColor = source.backgroundColor;
23.         dest.farClipPlane = source.farClipPlane;
24.         dest.nearClipPlane = source.nearClipPlane;
25.         dest.orthographic = source.orthographic;
26.         dest.fieldOfView = source.fieldOfView;
27.         dest.aspect = source.aspect;
28.         dest.orthographicSize = source.orthographicSize;
29. }
```

Listing 5.2.: Main camera properties same as for the reflection camera.

ii. Define the reflection plane geometry, point-normal mathematical representation using the reflection plane's normal N and the origin of that normal P (for a surface geometry, P is the world position), given in Listing 5.3. Transform both N and P into camera space in order to recalculate the oblique projection matrix of the reflection camera (transform the water surface as a clipping near plane)

```
1.  Vector4 CameraSpacePlane(Camera cam, Vector3 pos, Vector3 normal, float sideSign)
2.  {
3.          Vector3 offsetPos = pos + normal * clipPlaneOffset;
4.          Matrix4x4 currentViewMatrix = cam.worldToCameraMatrix;
5.          Vector3 clipPlanePos = currentViewMatrix.MultiplyPoint(offsetPos);
6.          Vector3 clipPlaneNormal = currentViewMatrix.MultiplyVector(normal).normalized * sideSign;
7.          return new Vector4(clipPlaneNormal.x, clipPlaneNormal.y, clipPlaneNormal.z, -
8.                                          Vector3.Dot(clipPlanePos, clipPlaneNormal));
9.  }
```

Listing 5.3.: Define the reflection plane inside the camera view space

iii. Transform the projection matrix of the reflection camera, using a reflection matrix around the reflection plane, to get the mirror rendered projection. Here add a small change to the reflection matrix, explained in Section 3.1 Reflection. In computer graphics, as mentioned earlier, vertices are transformed between different spaces, but these spaces are all connected, or more grammatically correct – they are affine spaces. This means that, in order to have translation, scaling and rotation or any other linear transformation matrix, we need it to be 4x4 affine matrix, while points are being represented by a 4D vector - $P < x, y, z, w >$, or homogeneous

coordinates – for point w=1, for transformation matrices w$< 0\ 0 \ldots 1 >$ (see Equation 5.1).

$$\boldsymbol{M_{refl}} = \begin{bmatrix} & & & \boldsymbol{t_x} \\ & \boldsymbol{A} & & \boldsymbol{t_y} \\ & & & \boldsymbol{t_z} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \quad (5.1)$$

iv. Transform the position of the reflection camera around the reflection plane, using the reflection matrix.

v. Rotate the reflection camera around its local view direction.

vi. Invert the culling order of all geometry in order to inverse the shading, since the produced image is a reflection and light is calculated based on normals of the triangles. By inverting the culling order, we basically flip the normals into their reflected ones.

vii. Render to texture

viii. Return the culling to its normal state for the next render frame of the main camera.

ix. Send the rendered reflection image data to the shader, handling the water surface rendering.

x. Generate UV-coordinates inside the shader for the reflection texture. Since the produced reflection image (see Figure 5.1 and Figure 5.2) is a rendered frame of scene, (can also be called screen capture) we need to overlay it/project it onto the screen, where the water surface is rendered.

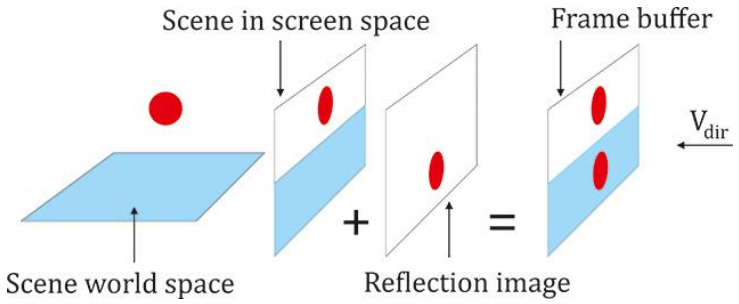


Figure 5.1.: Illustration of the process of reflection.

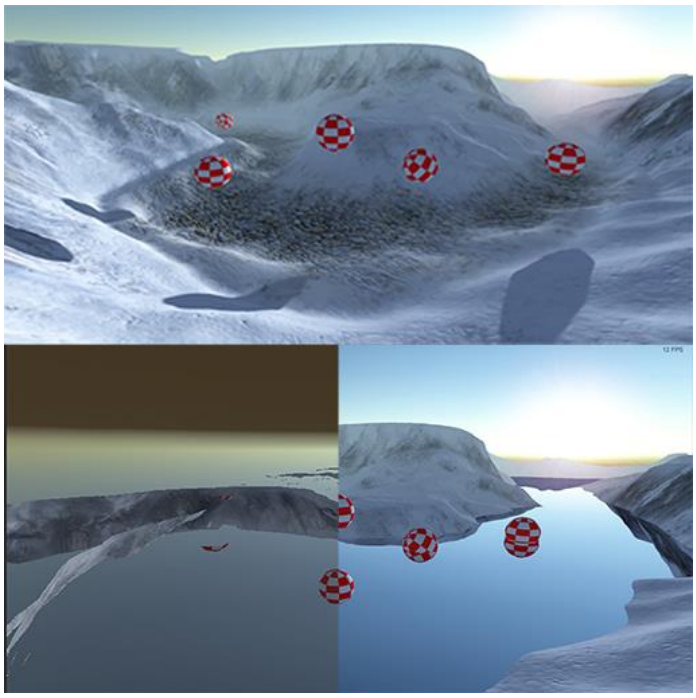


Figure 5.2.: Reflection results image - on the top – image of the scene without reflection. Below, on the left side is an image captured from the reflection camera and on the right side is the final result.

5.1.2 Refraction

Handing the refraction can be achieved in just the same way as the reflection, but that lead some problems we noticed with the shadow mapping. If we apply the previous technique, meaning that everything above the reflection plane that is cutout by the transformed oblique projection matrix, will not throw shadows underwater (since it is entirely or partially not rendered),as seen in Figure 5.3. Therefore, we decided to make use of Unity3D built in function *GrabPass* – a CG extension method that renders everything that is on the screen and stores it in a texture, without the need of additional cameras or rendering textures. The resulted image data is then sent to the following Pass () of the shader, responsible for the rendering.(see Figure 5.4) Use the same coordinate projection as the one in reflection will yield a result of no actual change in the final rendering, but it will provide us with masked fragment data of what is seen under the water surface – later used with distortion of the water.

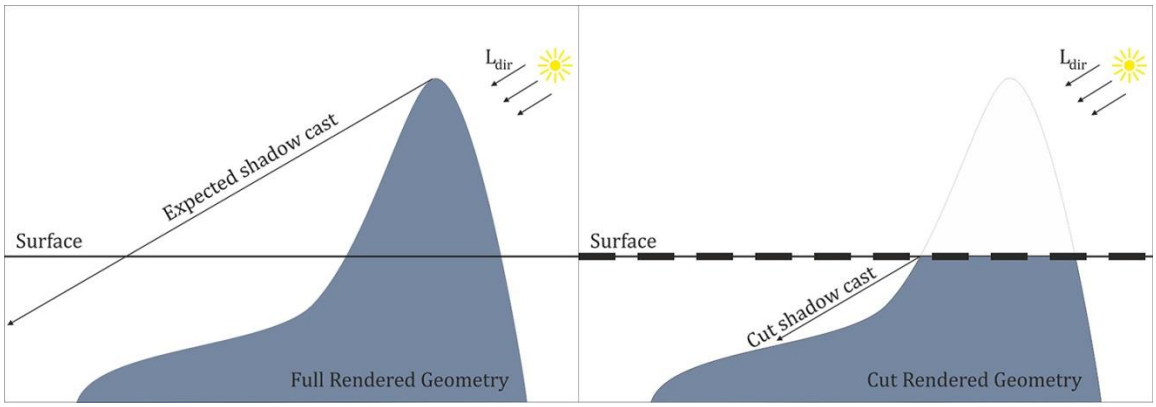


Figure 5.3.: Illustration of the refraction process.

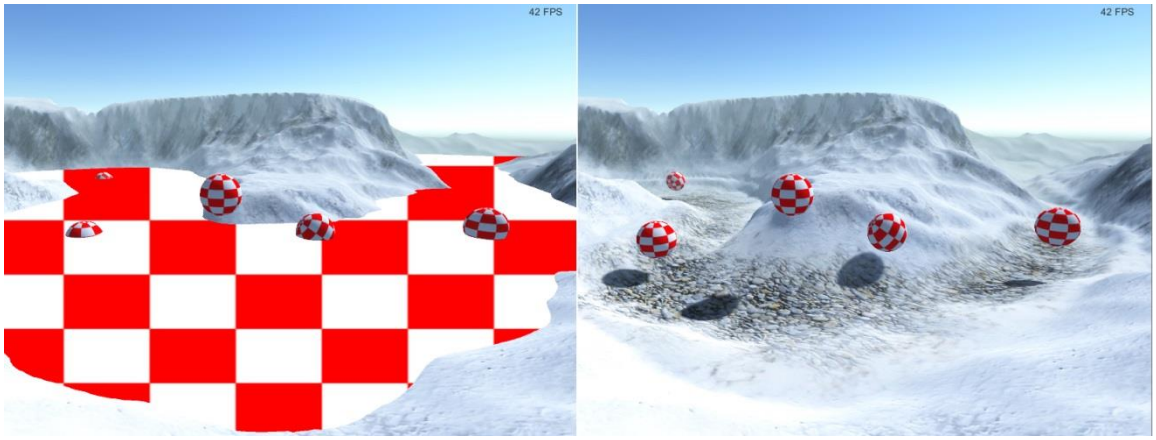


Figure 5.4.: Refraction coordinates - a) Scene showing (by using checkboard) where the refraction will be implemented (using GrabPass), on the left side and b) Scene with the refraction applied.

5.1.3 Fresnel effect

As mentioned earlier, the Fresnel effect gives the ratio between how much of the light is reflected and how much is refracted. Using the same logic, we need to define how much of the reflected image data is going to be seen over the refracted image data. This technique is called color blending and is based on linearly interpolate between two sets of data, based on a reference weight value – in our case that is the Fresnel factor. In order to calculate the Fresnel factor, we used both scenarios mentioned in Section 3.2 Refraction – the simplified Fresnel factor and Schlick's approximation.

- **Simplified Fresnel Factor**

  It requires, as inputs, the view direction and normal direction (see Equation 5.2). We need to make sure that both of them are in the same space and also both are normalized (length = 1). We can extract normals from the vertex input structure, which results in normal vectors in local/model space. After transforming them into world space, by multiplying with a Model2World matrix, we get interpolated world normals

(Listing 5.4, codeline 12). As for the view vector – its ray start is defined by the position of the view and its end - by the position of the viewed fragment (in case we are implementing the Fresnel equation factor inside the fragment shader) both in world space (Listing 5.4, codeline 13). Results can be seen in Figure 5.5.

$$\boldsymbol{R(\theta) = (V \cdot N) * correction} \quad (5.2)$$

```
1.  vert()
2.  {
3.          ...
4.          o.posWorld = mul(_Object2World, i.vertex);
5.          o.normalWorld = normalize(mul(float4(i.normal,0.0),_World2Object).xyz);
6.          ...
7.  }
8.  frag()
9.  {
10.         ...
11.    // FRESNEL - Simplified
12.         float3 nDir = normalize(i.normalWorld);
13.         float3 vDir = normalize(_WorldSpaceCameraPos - i.posWorld.xyz);
14.         float fresnelFactor = dot(vDir,nDir)*_FresnelCorrection;
15.         float4 fresnelColor = lerp(refl, refr, fresnelFactor);
16.         ...
17. }
```

Listing 5.4.: Fresnel effect, using Simplified Fresnel factor.

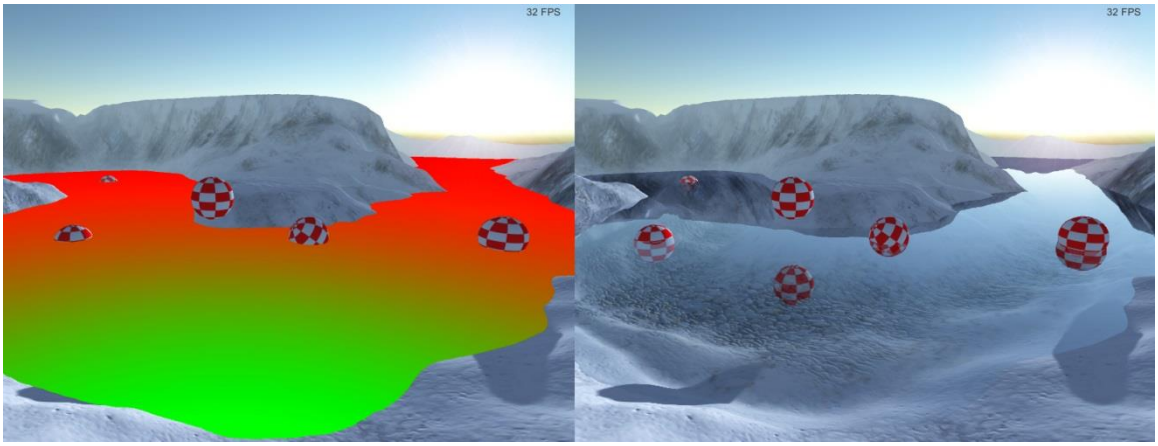


Figure 5.5.: Simple Fresnel debug and results - a) On the left - Illustration of the Fresnel effect, where the green color represents the refraction and in red is the reflection, calculated by using the Simplified Fresnel factor. b) On the right – the final result.

- **Schilick's approximation**

To calculate the Fresnel factor, we need to have the refraction indices of both medians – water and air (see Equation 5.3). Using the formulas from Section 3.2 Refraction (repeated here for convenience)(see Listing 5.5), we get the following results (Equation 4.4)(see Figure 5.6) :

$$\boldsymbol{n_{air} = 1.000293 \approx 1}$$

$$\boldsymbol{n_{water} = 1.330}$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 = \left(\frac{1-1.33}{1+1.33}\right)^2 \approx 0.02 \quad (5.3)$$

$$R(\theta) = R_0 + (1 - R_0) * (1 - \cos\theta)^5 = 0.02 + 0.98 * (1 - (V \cdot N))^5 \quad (5.4)$$

```
1.  frag()
2.  {
3.          ...
4.     // FRESNEL - Schilick's approximation
5.          float fresnelFactor1 = 0.02f + 0.98f*pow(1-(dot(nDir,vDir)),5);
6.          float4 fresnelColor1 = lerp(refr, refl,fresnelFactor1);
7.          ...
8.  }
```

Listing 5.5: Fresnel effect, using Schilick`s approximation.

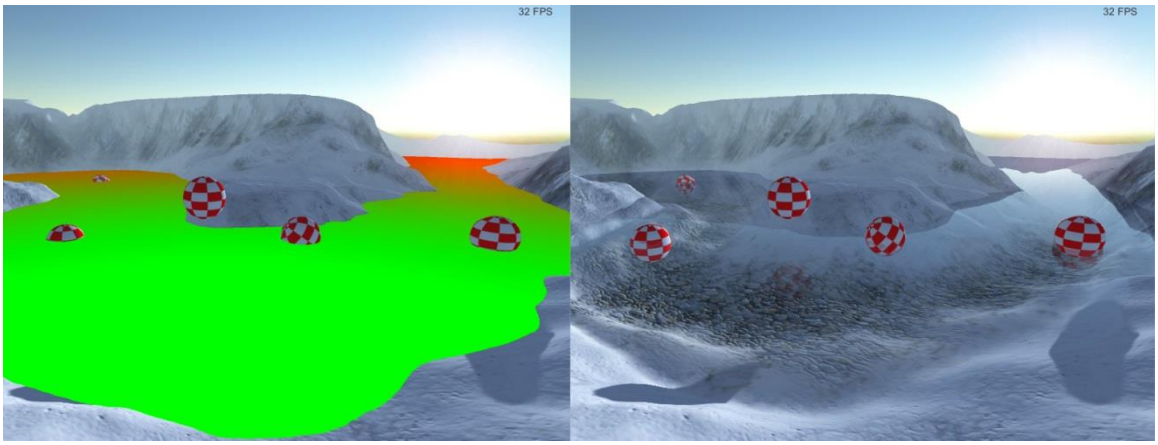


Figure 5.6: Schilick's approximated Fresnel debug and results - a) On the left - Illustration of the Fresnel effect, where the green color represents the refraction and in red is the reflection, calculated by using the Schilick`s approximation. b) On the right – the final result.

Both Fresnel factor equations fulfill the task of creating a ration between reflected and refracted image data, whereas the first one provides more artistically enhanced version, while the second one is more nature accurately.

5.1.4 Normal mapping

This technique is used to create small details like waves as well as some distortion to the previously generated reflection and refraction image data sets. More importantly, the water waves are the only visual cue that serves as proof of water being in a motion state. Therefore, the flow vector field technique has to be implemented on per normal map basis, meaning that we need to create flowing normal field.

- **Flow map generation**

Generation of the flow field is a pre-computational stage, where we took a render image of the landscape, see Figure 5.7. Based on that image, we then created a vector field

inside a free flow generator external tool that linearly interpolated color values between the vectors inside that field.

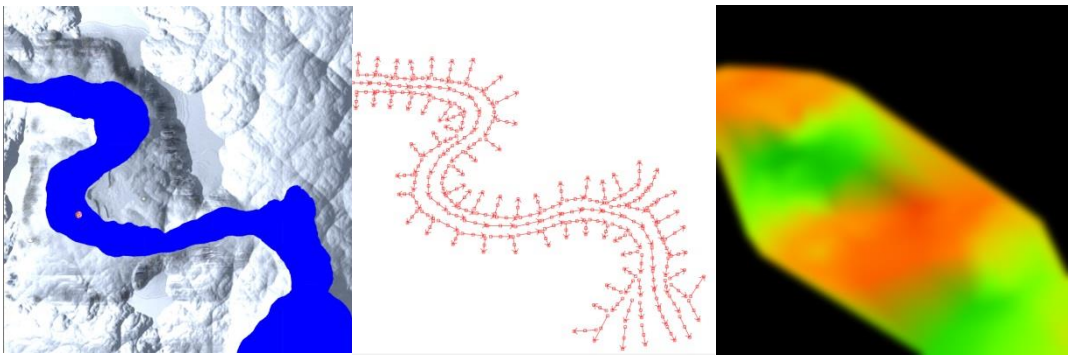


Figure 5.7: Flow map generation - The first image presents the place where the flow map will be created, while the second one illustrates the vector field on that same place. The last image shows the final result of the flow map.

Vector fields are functions that assign a vector to a point in space, considering that we our target is a texture image that will hold the flow field data, the equation of 2D vector field is given by:

$$\boldsymbol{F}(\boldsymbol{x},\boldsymbol{y}) = < \boldsymbol{v} > \quad (5.5)$$

The external tool we used, allows to predefine the direction of the vectors and does not take into consideration their magnitude, thus they are either with length equal to 1 (normalized) or 0. After all vectors are placed manually, they are being transformed into the RG color space, from where values for each individual channel, respectively R and G is calculated, knowing the exact direction of the vectors, as shown in Figure 5.8.

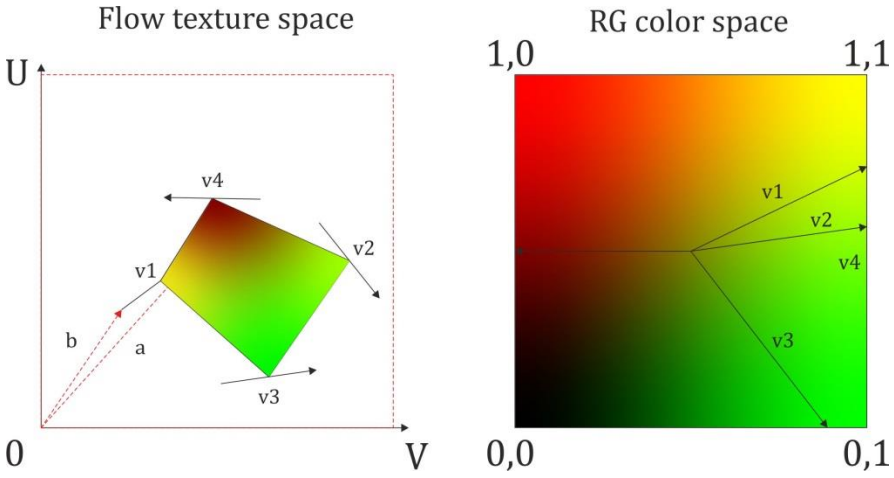


Figure 5.8: Flow field vector generation - a) Image of flow texture space on the left side. b) Image of the interpolation between the colors in RG color space.

Since the color channels of the flow texture can hold information in the interval $[0,1]$ and we use all directional flow texture, where $R = 0.5, G = 0.5$ means no movement, or velocity equal to zero, we need to map the color values into range

$[-1,1]$. If we consider vector $v_0 < r, g > \epsilon F$, each channel is scaled by 2 and shifted by -1:

$$\boldsymbol{r, g} \in [\mathbf{0} * \mathbf{2} - \mathbf{1}, \mathbf{1} * \mathbf{2} - \mathbf{1}] \rightarrow \boldsymbol{r, g} \in [-\mathbf{1}, \mathbf{1}]$$

The next stage, in using a flow texture, is to create a shifting phase, or in other words define the loop algorithm, used to translate a pixel along the vector field (see Figure 5.9):

1. Define a phase, based on linear function - $\_Time$, or how long should one loop take before restarting again. We can also add some offset to the phase as well, creating difference in the time when a pixel starts translating, for this we use a noise texture – *phase, given in Listing 5.6, code line 5.*

$$\boldsymbol{phase} = \boldsymbol{time\ in\ sec / cycles\ per\ sec} + \boldsymbol{noise\ offset}$$

2. Create two shifting/offset values, based on the vector flow texture – *offset1/offset2, given in Listing 5.6, codeline 6, 7.*

$$\boldsymbol{offset} = \boldsymbol{flow\ RG\ data} * \boldsymbol{phase}$$

3. Map the normal texture with the model's uv set and offset that uv set with the first shifting value from the previous step – *Layer1 (Listing 5.6, codeline 9)*

$$\boldsymbol{mapping}(\boldsymbol{texture}, \boldsymbol{uv} * \boldsymbol{scale} + \boldsymbol{offset})$$

4. Perform previous step, this time with the second shifting value – *Layer2 (Listing 5.6, codeline 11)*
5. Define the weight of linear interpolation between the two normal textures, and since we are using time as a linear function (which grows constantly), we can get the fraction of it. This allows us to know how much opacity should be taken into consideration since a fraction is in the interval $(0,1)$ – *opacity (Listing 5.6, codeline 13)*

```
1.  // BUMP/FLOW DISTORTION - WATER
2.          float2 flowDir = tex2D(_FlowMap, -i.uv).rg * 2 - 1;
3.          flowDir *= _Translocation;
4.          float noise = tex2D(_NoiseMap, i.uv).r;
5.          float phase = _Time.y/_Cycle + noise.r * 0.1;
6.          float2 offset1 = float2(flowDir.x,-flowDir.y) * frac(phase + 0.5f);
7.          float2 offset2 = float2(flowDir.x,-flowDir.y) * frac(phase);
8.          float interaction = tex2D(_BlobTex, i.uv).r ;
9.          float4 layer1 = (tex2D(_BumpMap,i.uv.xy * _BumpMap_ST.xy +_BumpMap_ST.wz +
10.                                                                  offset1*(1-interaction)));
11.         float4 layer2 = (tex2D(_BumpMap,i.uv.xy * _BumpMap_ST.xy +_BumpMap_ST.wz +
12.                                                                  offset2*(1-interaction)));
13.         float opacity = frac(phase);
14.         if( opacity > 0.5f)
15.                 opacity = 2 * (1 - opacity);
16.         else
17.                 opacity = 2 * opacity;
18.         ...
19.         float4 encodedNormalWater = lerp(layer1,layer2,opacity);
```

Listing 5.6: Fragment code for creating a flow.

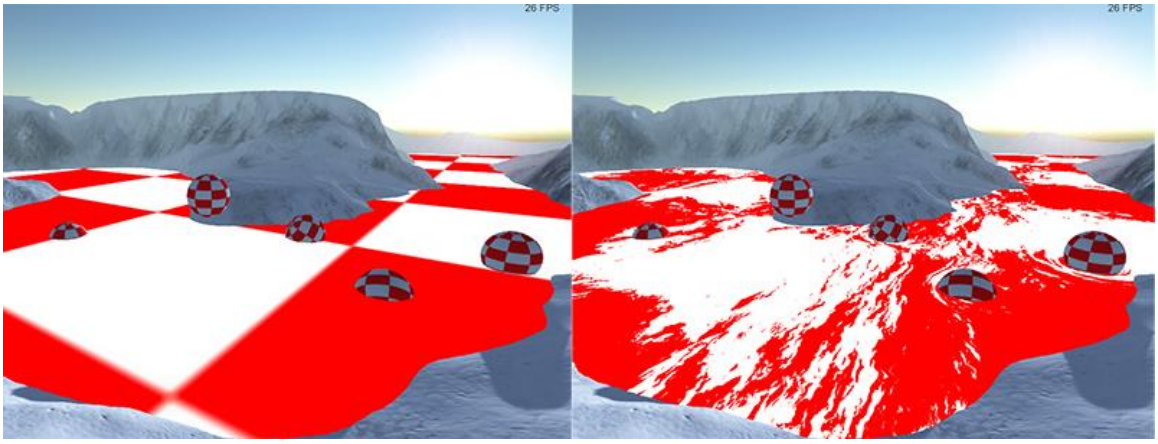


Figure 5.9: Perturbing UVs with flow map - a) Image of place of the flow, presented by checkerboard, on the left side. b) Image of the flow distortion, on the right side.

The created distortion can then be used to perturb the uv coordinates used for mapping the reflection and refraction images, allowing us to have distorted by image data formed via the animation of pixels inside the pre-defined vector field, see Figure 5.10.

Figure 5.10: Final result of the flow distortion.

5.1.5 Lighting model

Since we are creating surface water, meaning that the model's geometry is a flat plane, thus there is no shadow casting geometry. The only way to simulate lights and shadows is to use the normal map. As explained in the Section 5.1.4 Surface water rendering - Normal map, we make use of a normal map texture, from where we can extract normal directions based on pixel color values from the texture. The reconstructed normal vectors have to be transformed into world space.

$$\boldsymbol{normal\ texture} \rightarrow \boldsymbol{n_x n_y} \rightarrow \boldsymbol{n_z}$$

For each vertex we can get its normal N and tangent T from the CPU as part from the vertex input structure. Based on them, we can then calculated the binormal $B = N \times T$. All these 3 vectors are forming Unity3D specific local surface coordinate system for every surface point, which is used to interpolate the normal values when passed from the vertex shader to the fragment shader. All three vectors have to be computed in world space, since we want to relate the normal vector N to other objects inside the world space – in this case the light position. Transforming a normal from one space to another is done by multiplication with a transpose inverse matrix - $N * (M^{-1})^T$, because after transforming both the surface and its normal, should stay orthogonal to each other. Considering surface s, normal vector $n \perp s$ transformed with matrix $(M^{-1})^T$ and vector $v \in s$ transformed with M:

Inner product $\boldsymbol{A \cdot B = A^T B}$: $\quad (\boldsymbol{M^{-1}})^{\boldsymbol{T}}\boldsymbol{n} \cdot \boldsymbol{Mv} = ((\boldsymbol{M^{-1}})^{\boldsymbol{T}}\boldsymbol{n})^{\boldsymbol{T}}\boldsymbol{Mv}$

Transpose $(\boldsymbol{AB})^{\boldsymbol{T}} = \boldsymbol{B^T A^T}$: $\quad = [\boldsymbol{n^T}((\boldsymbol{M^{-1}})^{\boldsymbol{T}})^{\boldsymbol{T}}]\boldsymbol{Mv}$

Transpose $((\boldsymbol{A})^{\boldsymbol{T}})^{\boldsymbol{T}} = \boldsymbol{A}$: $\quad = [\boldsymbol{n^T M^{-1}}]\boldsymbol{Mv}$

Inverse $\boldsymbol{A^{-1} = I}$: $\quad = \boldsymbol{n^T Iv} = (\boldsymbol{n \cdot v})\boldsymbol{I}$

Identity matrix applies no transformation to a vectors:

$$\rightarrow \boldsymbol{N_{transformed} \cdot v_{transformed} \equiv n \cdot v}$$

As regards the tangent T, its describing a direction between index followed vertices, thus is transformed using the object-to-world matrix , see Listing 5.7(same one used for the vertices themselves). Using both T and N in normal space, we can then easily calculate B.

```
1.  vert()
2.  {
3.          ...
4.          o.normalWorld = normalize(mul(float4(i.normal,0),_World2Object).xyz);
5.          o.tangentWorld = normalize(mul(_Object2World, float4(i.tangent.xyz,0)).xyz);
6.          o.binormalWorld = normalize(cross(o.normalWorld, o.tangentWorld) * i.tangent.w);
7.          ...
8.  }
```

Listing 5.7: Normal space – computing the normal, tangent and binormal per vertex basis

Using the normal, tangent and binormal in world space we can then form a matrix that is used to map each extracted normal vector from the normal texture to the corresponding interpolated normal, because the three vectors N, T, B are axes of subspaces.

$$\boldsymbol{M_{N_{local} \rightarrow N_{world}}} = [\boldsymbol{T} \quad \boldsymbol{B} \quad \boldsymbol{N}] = \begin{bmatrix} \boldsymbol{T_x} & \boldsymbol{B_x} & \boldsymbol{N_x} \\ \boldsymbol{T_y} & \boldsymbol{B_y} & \boldsymbol{N_y} \\ \boldsymbol{T_z} & \boldsymbol{B_z} & \boldsymbol{N_z} \end{bmatrix}$$

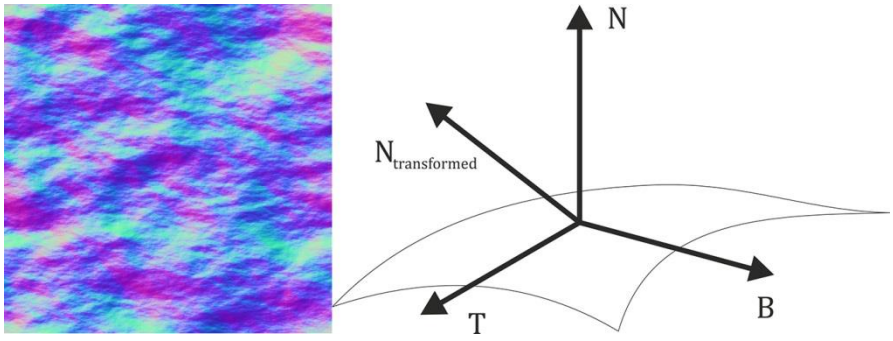


Figure 5.11: Normal extraction from texture - a) Image of normal map, on the left; b) Illustration of the Normal, Tangent, Binormal and Transformed normal

If we consider normal $n \in normal\ texture$ with values in the alpha channel $a = n_x$ and green channel $g = n_y$, we first map them from $[0,1]$ to $[-1,1]$ since normal vectors show the facing direction of a surface. Based on the values a and g, we calculate $n_z = \sqrt{1 - n_x^2 - n_y^2}$. Having all of the components of the newly formed normal vector, we can now transform it using the matrix described above.

$$\boldsymbol{nM}_{N_{local} \rightarrow N_{world}}$$

```
1.  frag()
2.  {
3.      // NORMAL MAP DECODING
4.          ...
5.          float4 encodedNormal = lerp(layer1,layer2,opacity);
6.          float3 localCoordsNormal = float3(2*encodedNormal.a - 1,2*encodedNormal.g - 1, 0.0);
7.          localCoordsNormal.z = sqrt(1 - pow(localCoordsNormal.x,2) - pow(localCoordsNormal.y,2));
8.          float3x3 local2WorldNormalMatrix = float3x3(i.tangentWorld,i.binormalWorld,i.normalWorld);
9.          float3 normalDirBump = normalize(mul(localCoordsNormal, local2WorldNormalMatrix));
10.         ...
11. }
```

Listing 5.8: Reconstructing normal vectors based on normal map

As mentioned earlier, light computation makes use of a set of parameters based on the desired light model. For example position of the light in world space, direction of the light, view direction, surface normal and etc. We will divide the light model into a few sections for convenience, namely ambient lighting, diffuse lighting, specular lighting and translucent lighting, each serving its own purpose. We also make use of only one directional light in the scene for simplicity, representing the sun.

- **Ambient light**

This is a fixed color value that affects all objects in the scene and usually is dictated inside the editor, thus we get that color value via Unity3D built in function, given in Listing 5.9.(see Figure 5.12)

```
1.  // Ambient
2.          float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb * fresnelColor.rgb;
```

Listing 5.9: - Ambient lighting.

Figure 5.12: Ambient lighting.

- **Diffuse light**

The geometry of the water will interpolate normal vectors that are parallel to each other, as the geometry is forming a flat surface. Thus, calculating the diffuse factor D will give the same value for all rendered fragments, unless we use the previously reconstructed normal vectors, extracted from the normal map and compared with the light direction. We apply both of these calculations – first one giving us a "flat" diffuse factor and the second one – "bumped" diffuse factor, given in Listing 5.10 (see Figure 5.13). We then lerp between these factors, based on how much diffuse reflectance we want to have in the water – giving more artistic control, or how intense the self-shadowing of the waves should be. We also apply the Valve's Half-Lambert diffuse reflection to further smooth out the diffuse shading.

$$\boldsymbol{D_{flat} = N \cdot L}$$

$$\boldsymbol{D_{bumped} = N_{transformed} \cdot L}$$

$$\boldsymbol{D = \left[D_{flat} + w * (D_{bumped} - D_{flat})\right] * 0.5 + 0.5}$$

```
1.  //Diffuse
2.          float diffuseFactor = max(0,dot(nDir, lightDir));
3.          float diffuseFactorBump = max(0,dot(normalDirBump, lightDir));
4.          diffuseFactor = lerp(diffuseFactor,diffuseFactorBump,_BumpShadowAmplitude);
5.          diffuseFactor = diffuseFactor * 0.5 + 0.5;
6.          float3 diffuseLighting = _LightColor0.rgb * fresnelColor.rgb * diffuseFactor;
```

Listing 5.10: Diffuse lighting.

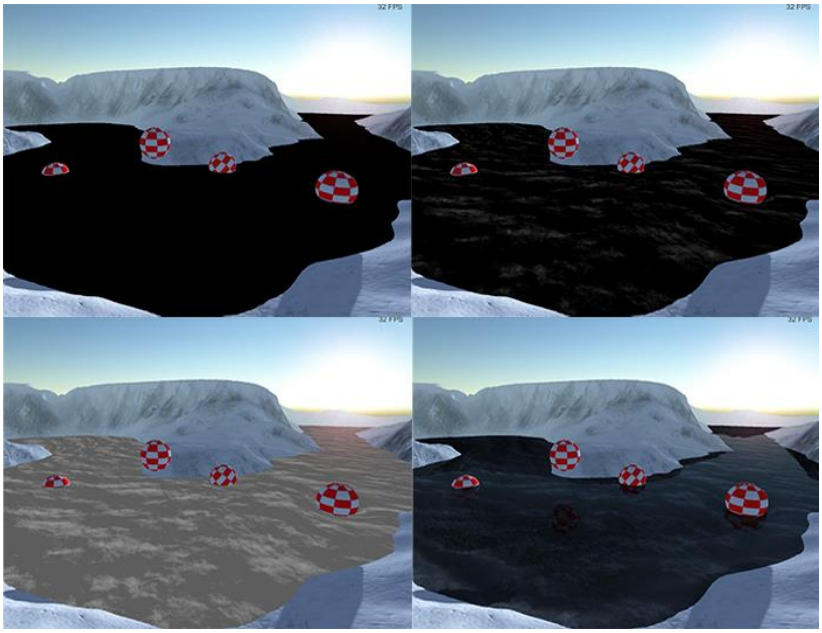


Figure 5.13: Diffuse lighting - the top two images – flat diffuse lighting, on the left and bumped diffuse lighting, on the right. Regarding the two imaged above – Half-Lambert diffuse lighting, on the left and the final image, on the right.

- **Specular light**

The brightest spots in the water are basically the tips of the waves. This means that this light calculation will rely entirely on the extracted normal vectors from the normal map, given in Listing 5.11. As described earlier, this is the focused light, reflected in the viewer's eye, thus we need the following arguments: view direction, light direction and normal vector (around which light will be reflected). Again, we make use of the flat shading of the surface geometry, against the bumped one, resulting in more smooth transition of the specular light, based on where most of reflected light rays are focused on a flat geometry, see Figure 5.14.

$$S_{flat} = \left(N \cdot L_{reflected}\right)^{Shininess}$$

$$S_{bumped} = \left(N_{bumped} \cdot L_{reflected}\right)^{Sharpness}$$

$$S = S_{flat} + w * (S_{bumped} - S_{flat})$$

```
1.  // Specular
2.          float specularBumpFactor = pow(max(0,dot(reflect(-lightDir,
3.                                                           normalDirBump),vDir)),_Sharpness);
4.          float specularInterpolatedFactor = pow(max(0,dot(reflect(-lightDir,
5.                                                           nDir),vDir)),_Shininess);
6.          float3 specularLighting = _LightColor0.rgb * _SpecColor.rgb * (fresnelColor.a) *
7.                                    (lerp(specularBumpFactor,specularInterpolatedFactor,0.3));
```

Listing 5.11: Specular lighting.

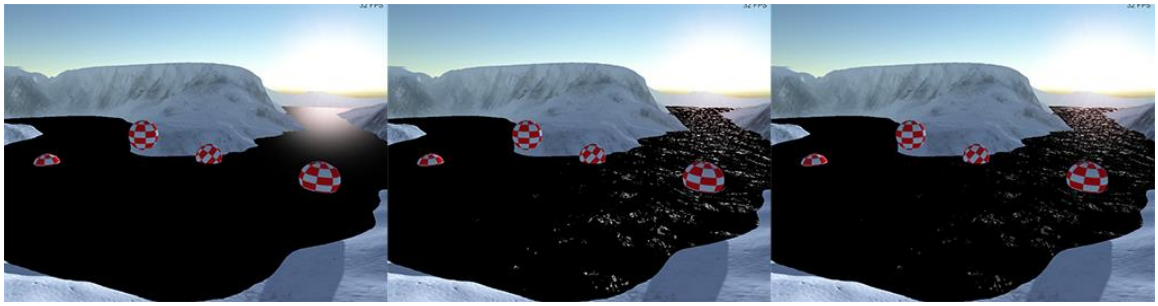


Figure 5.14: Specular lighting - the first image presents the flat specular lighting, second one- the bumped specular lighting and the last one is the final specular lighting.

5.2 Additional Visual effects

5.2.1 Translucent light

Additional lighting used to define sub-surface scattered light, or how light is refracted inside the volume of an object. Since we have flat geometry, the translucent light can help us with making the water able to render properly when the surface is not facing the light direction at all, or when the diffuse factor is equal to zero. This allows us to have rendering property for the water, in case the environment is set to be at night, thus the translucent light is given by Equation 5.6.(see Figure 5.15) The only difference is that we get the inverse normal direction, opposite to the diffuse light factor, see Listing 5.12. We add this as an extra feature with possibility of future work on it (see feature work section for further details).

$$\boldsymbol{T_{factor}} = \boldsymbol{L} \cdot (-\boldsymbol{N}) \tag{5.6}$$

```
1.  // Translucent Diffuse
2.          float translucentFactor = max(0,dot(lightDir, - nDir));
3.          float3 translucentDiffuseLighting = _LightColor0.rgb * _DiffuseTranslucentColor.rgb *
4.                                                                  translucentFactor;
```

Listing 5.12: Translucent diffuse.

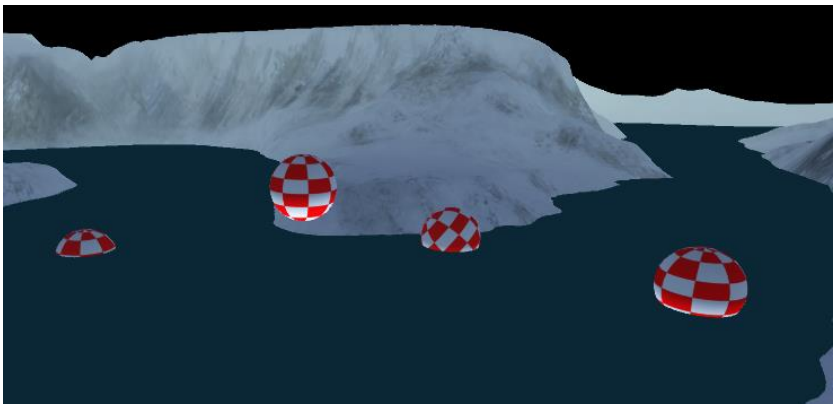


Figure 5.15: Translucent diffuse.

5.2.2 Fog

In order for the shading to include build in options inside Unity3D, like fog for instance, we need to specify how this shading is added on top of our results. Simply said, fog in computer graphics is connected with spatial decrease/interpolation in the color values towards a specific color value, based on reference value (calculated via a so called fog equation [72]. In the previous images, our scene was using the squared exponential fog equation:

$$\boldsymbol{f = 1/e^{(d*b)^2}} \quad (5.7)$$

, where e is the exponent; d distance in world space and b is the fog attenuation/density (defined through the editor), see Equation 5.7. The only unknown is the distance. We can either solve it with hardcoded value or we can make use of the fragment projection distance from the camera. The second choice is much easier to use, but sufficiently harder (explanatory). When a vertex is transformed from local/model space to projection space using the combined matrix MVP *(model/view/projection)*, it ends up being projected onto the near clipping plane, inside the clip space, given in Listing 5.13. Therefore, its forth vector component w, or the 4th dimension will contain information how was the vertex transformed from the previous space (homogeneous coordinates), thus:

$$\boldsymbol{vertex[w] = distance\ between\ near\ clip\ plane\ and\ position\ in\ view\ space = z_{eye\ space}}$$

```
1.  vert()
2.  {
3.          ...
4.          o.pos = mul(UNITY_MATRIX_MVP, i.vertex);
5.          ...
6.  }
7.  frag()
8.  {
9.          ...
10.         // FOG
11.         float eyeZ = i.pos.w;
12.         float fogFactor = 1/exp(pow(abs(eyeZ)*_FogDensity,2));
13.         finalColor = lerp(_FogColor,finalColor,fogFactor);
14.         ...
15. }
```

Listing 5.13: Fog.

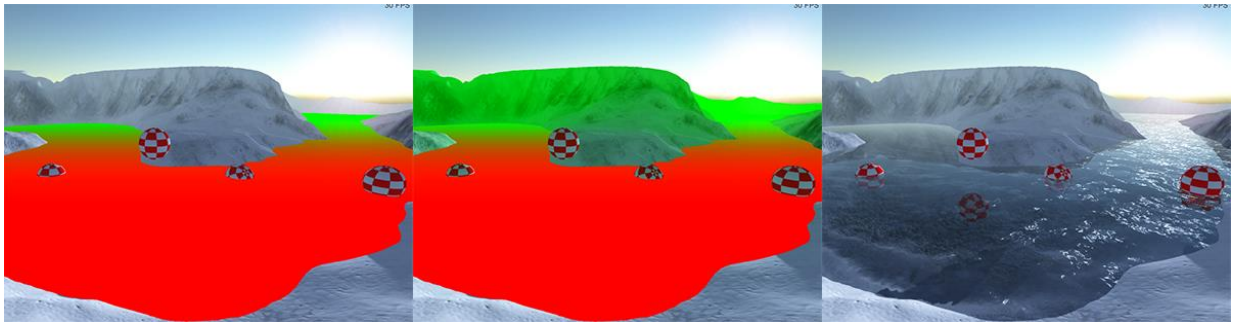


Figure 5.16: Fog debugging - the first image shows the fog effect on the water surface. In the second image, is presented the fog in the whole scene, matching with the first image. The final result of the scene is shown on the last image.

5.2.3 Edge blending

To further enhance the rendering of the surface, we also use another blending technique that lowers the alpha value of the final output color of the water surface, based on the depth of the pixels. This technique makes use of the depth texture – a rendering stored inside a depth buffer. Unity3D allows us access to this buffer, from where we can extract two types of data sets:

- Linear01Depth - providing values within the range [0, 1], based on the position of the rendered fragment inside the clipping space (between near and far clipping planes).
- LinearEyeDepth – providing values between the camera position and the rendered fragment, using world space units.

Our final goal is to mask fragments from the water surface that are significantly closer to the rendered fragment behind (the shoreline formed by the terrain) and reduce the alpha value of these fragments. In order to preserve the shoreline fragments, (which are rendered behind/under the water surface) we need to prevent the water fragments from writing to the depth buffer (used for depth testing). After that we need a difference between the world depth inside the depth buffer and the position of the current fragment. Since the result is in world space units, we can saturate the results back into range [0,1] and pass a controlling/editable argument for the saturation, given in Listing 5.14, in order to customize the amount of difference between the two compared fragments (the one from the depth buffer and the water fragment), see Figure 5.17 and Figure 5.18.

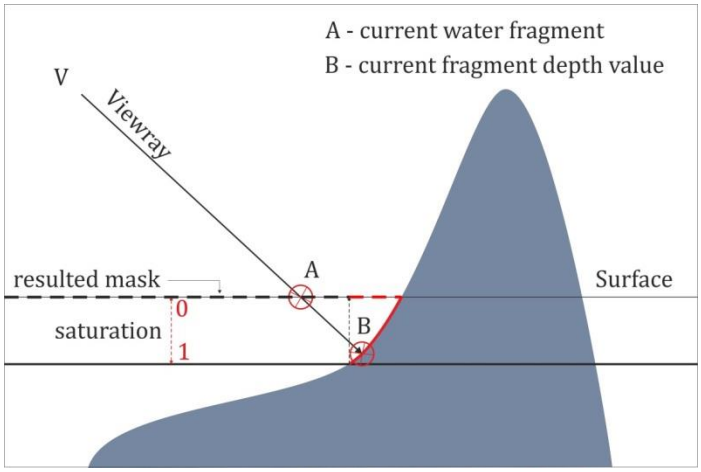


Figure 5.17: Edge blending vector scheme – fragment A is not written in the depth buffer in order to compare and blend it with B.

```
1.  // EDGE BLENDING
2.          float depth = tex2Dproj(_CameraDepthTexture, i.reflUV).r;
3.          depth = LinearEyeDepth(depth);
4.          float eyeZ = i.pos.w;
5.          float dist = abs(eyeZ - depth);
6.          float edgeBlend = saturate(_Fade*dist);
```

Listing 5.14: Edge blending.

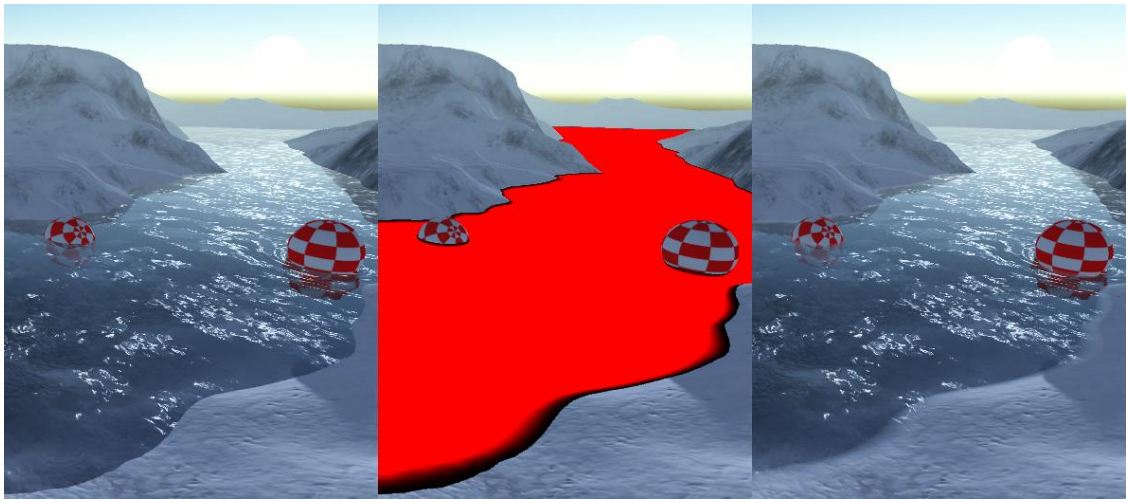


Figure 5.18: Edge blending debugging - First image – scene without edge blending, second image – only the edge blending and third one – the final result.

5.2.4 Vertex displacement

We implemented a simple vertex displacement based on adding sine waves (Fast Fourier Transform – FFT) and a displacement texture, as seen in Figure 5.19, (given in Listing 5.15). The idea is to calculate a sin value every frame and pass that value to the vertex shader,

then perform a lookup for the displacement texture channels to give variance to each sine wave direction, see Figure 5.20.

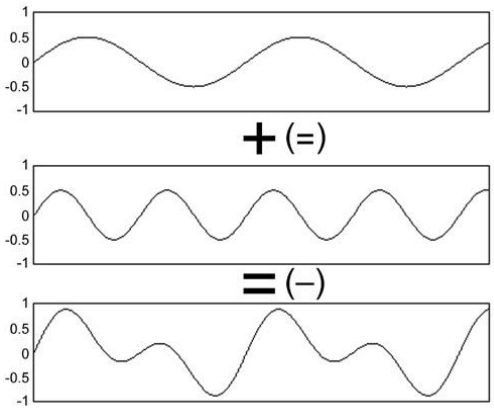


Figure 5.19: Visual representation of adding sin waves (FFT), image taken from http://music.columbia.edu/cmc/musicandcomputers/chapter3/03_03.php

$$\boldsymbol{y(t) = Amplitude * \sin(2\pi + phase)}$$

$$\mathbf{displacement} = \sin(\boldsymbol{a}) * \boldsymbol{tex_r} + \sin(\boldsymbol{b}) * \boldsymbol{tex_g} + \sin(\boldsymbol{c}) * \boldsymbol{tex_b}$$

```
1.  // Z-Vertex Displacement
2.          float3 dTex = tex2Dlod(_DispTex, float4(i.vertex.xy + _DispTex_ST.wz*_Time.y,0,0)).xyz;
3.          float d = (dTex.r * _ChannelFactor.x + dTex.g * _ChannelFactor.y + dTex.b *
4.                                                                          _ChannelFactor.z);
5.          i.vertex.xyz = i.vertex.xyz + i.normal * d * _DisplacementAmount;
```

Listing 5.15: Vertex displacement.

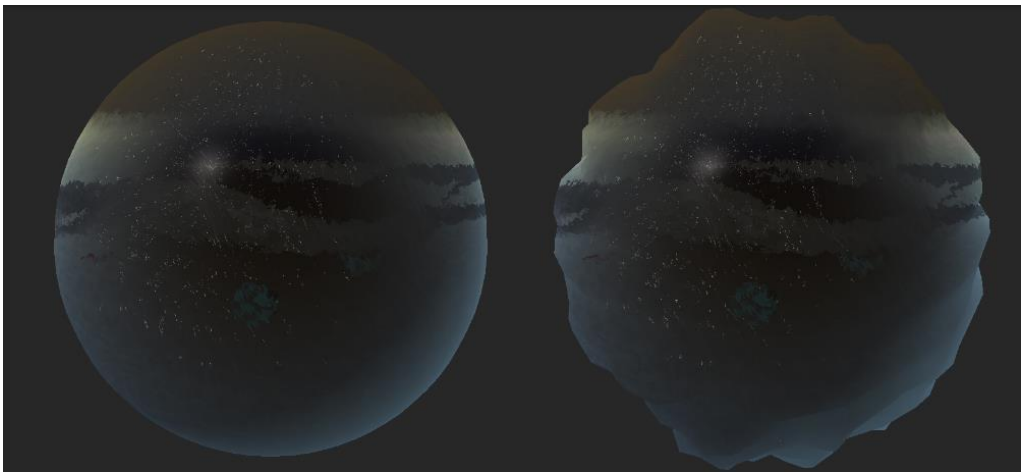


Figure 5.20: Vertex displacement - On the left : displacement $d = 0$; On the right: displacement $d = 0.5$, images taken from the Unity3D shader properties panel inside the editor mode, using spherical geometry in order to give better representation of the vertex displacement.

5.2.5 Ice transition

The last visual part is connected with the transition between the water states. We are limited with only the geometry we have, and since we are creating surface water, we are restricted to only simulating the effect, rather than the cause of it. In other words, we can only implement behavior of the water that transitions between the two states – liquid and solid. In order to achieve this effect, we need to have observation of what needs to happen. In reality, the freezing of a river is divided into a few stages [73], explained physically by changes that happen to the water liquid. The water transitions between states of crystals, through a thick layer of formed ice plates. Then a border of ice is formed along the river bank. Since the water height in that area is lower, thus the velocity is also lower. The final stage of the water is when a solid layer of ice is formed on top of the water. Another event, usually observed when water freezes, is the increase of size, but regarding rivers or lakes the effect observed is opposite, because they water flow decreases. We took all these effects into their most basic form and implemented them as well as possibility to control the event at some key points (for example -the water level decrease).

To form a water transition we first need to produce the final stage of the frozen water and afterwards interpolate between the two states of the water using a control value. The ice color is formed by an image mapped to the UV coordinates of the water plane. We then use alpha blended value used from a pre-computed texture handling the spread of the ice – from the shore, towards the inner part of the river (Figure 5.21). We apply this technique twice, where the second iteration is used with lower saturation value, resulting in more smoothed image (Figure 5.22) (Listing 5.16). Blue tint is multiplied with the second iteration, since it plays the role of a thick layer ice cover (the refractive index of the blue light is the greatest one from the RGB light).

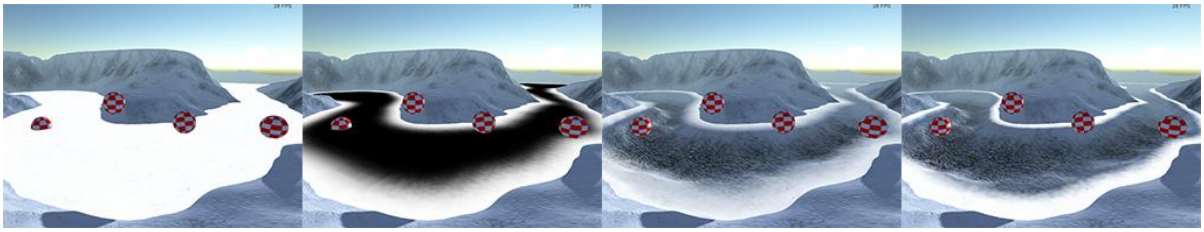


Figure 5.21: Snow blending - a) Snow color; b) Pre-defined ice region; c) Alpha blended image; d) Saturated image.

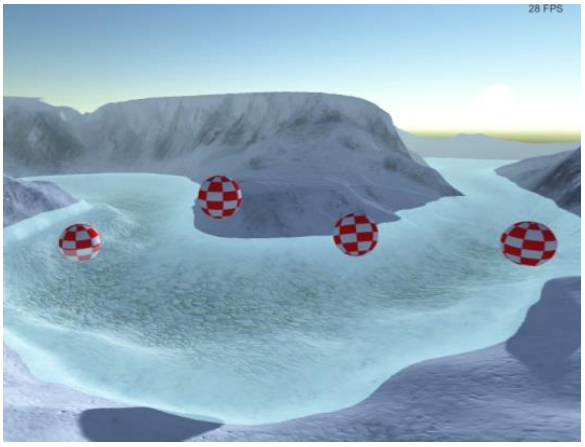


Figure 5.22: Ice forming - 20 times smaller saturation, resulting in more smoothed distribution. Final color is multiplied by a bluish value.

```
1.  // SNOW & ICE
2.  // First iteration
3.          float4 snow = tex2D(_SnowTex, i.uv.xy * _SnowTex_ST.xy + _SnowTex_ST.wz);
4.          snow.a = tex2D(_SnowBlend, -i.uv.xy).r;
5.          snow.a += (_BlendAmount * 2 - 1);
6.          snow.a = saturate((snow.a * _EdgeSharpness - (_EdgeSharpness - 1) *0.5)*_Thickness);
7.  // Second iteration
8.          // same uv, but edge sharpness is lower to apply blurriness
9.          float4 ice = tex2D(_SnowTex, i.uv.xy * _SnowTex_ST.xy + _SnowTex_ST.wz);
10.         ice.a = tex2D(_SnowBlend, -i.uv.xy).r;
11.         ice.a += (_BlendAmount * 2 - 1);
12.         ice.a = saturate((ice.a * _EdgeSharpness/20 - (_EdgeSharpness/20 - 1) *0.5)*_Thickness);
13.         ice.rgb *=_IceColor.rgb;
14. // Final texture color
15.         float4 finalIce;
16.         finalIce = lerp(snow,ice,1-snow.a);
```

Listing 5.16: Snow & Ice.

To enhance the produced color value, we also use a second normal map for the ice. The ice normal map is used for producing different shading/lighting output. (Listing 5.17)

```
1.  // NORMAL MAP DECODING
2.          float4 encodedNormalIce = tex2D(_SnowBump, i.uv.xy * _SnowBump_ST.xy + _SnowBump_ST.wz);
3.          float4 encodedNormalWater = lerp(layer1,layer2,opacity);
4.          float4 encodedNormal = lerp(encodedNormalWater, encodedNormalIce, _Thickness);
```

Listing 5.17: Normal map decoding.

The final ice color is calculated based on a controlling value in the interval [0, 1], where zero means water is in liquid state and one means water is in frozen state. This controlling value is used to interpolate between the normal maps, as well as it control a Y-axis displacement in the vertex shader, resulting a negative translation of the vertices, forming the plane along that axes (frozen water level decreases, as it freezes). (Listing 5.18)

```
1.   vert()
2.  {
3.          ...
4.          o.pos = mul(UNITY_MATRIX_MVP, i.vertex - float4(0.0, _Thickness*0.2, 0.0, 0.0));
5.          ...
6.  }
```

Listing 5.18: Vertex displacement before transforming it in clip space

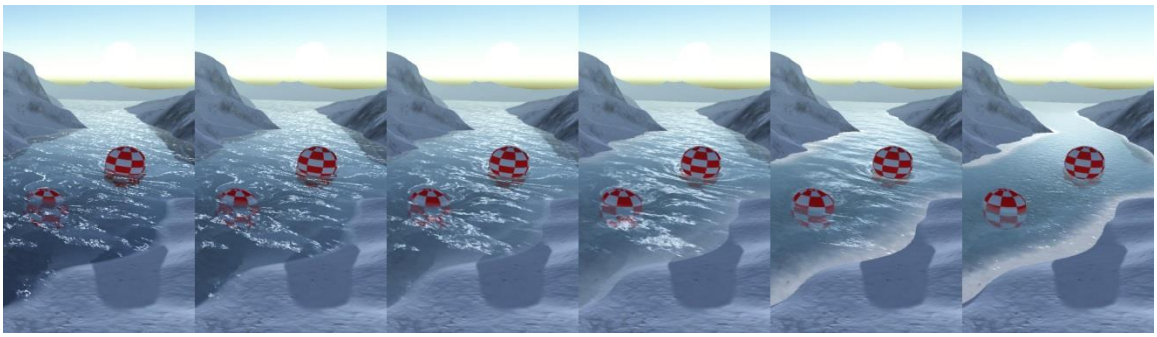


Figure 5.23: Ice thickness - in ascending order with values: thickness $= 0;\ 0.2;\ 0.4;\ 0.6;\ 0.8;\ 1.0$

Object to water interaction – as explained earlier in Section 3.11 - Water to object interaction, when objects enter the water, they apply influence to it. Since our water is defined by a water plane geometry, the distortion that objects can create inside the geometry of the water is either applied on the pixels or the vertices. We decided to implement the first type by using another render texture. The following steps were followed to produce an image, we called "interaction texture", since it holds all the information about objects interacting with the water. The interaction texture was captured by another camera placed inside the scene, as seen in Figure 5.24, via the following the set of steps:

- Interaction camera – a camera with orthogonal projection is placed right above the scene geometry, so that it renders the entire water plane.
- Interaction fragments – each object interacting with holds a quad object with texture, representing approximated geometry projection from the interaction camera onto the water plane.
- Render to texture – results in a cubic texture (width and height pixel resolution are values equal to 2^n).
- Texture-to-shader – the resulted texture is then sent inside the shader to distort the vector field, used for the flow map, thus resulting in distortion of the animation of the pixels.

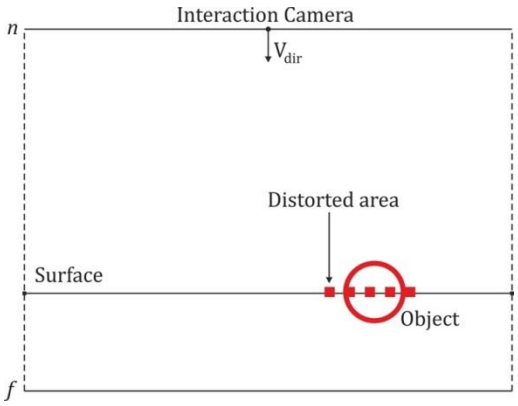


Figure 5.24: Illustration of the process of capturing the interaction texture.

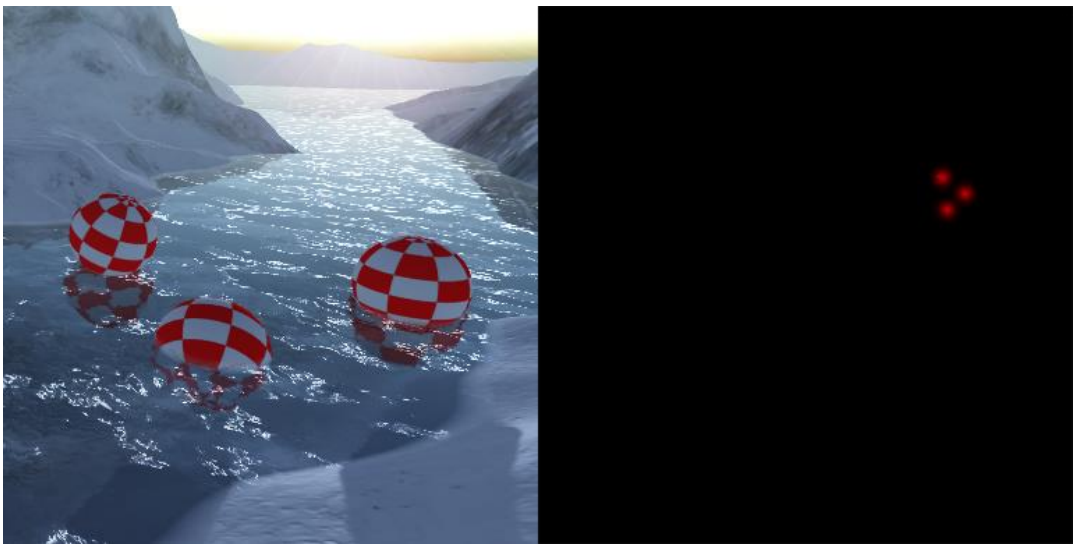


Figure 5.25: Interaction texture - only three spheres in the water, on the left,, resulting in rendering three quads holding approximated geometry projection of the spheres, on the right.

The rendered texture is send inside the fragment shader and mapped to the UV coordinates of the water plane, but only one channel is used. Consider texel $t(x, y) \in interaction\ texture$ and $offset$, being the vector from the flow vector texture, and $t_r \in [0,1]$. Then all black texels (values equal to zero) will have no impact on the flow map, while ever colorful texel (values not equal to zero) will impact the flow map linearly, transforming the corresponding vector inside the flow vector field , example can be seen in (Figure 5.26).

$$\boldsymbol{t_r = 0 \ \rightarrow offset * (1 - t_r) = offset}$$

$$\boldsymbol{t_r \neq 0 \rightarrow offset * (1 - t_r) \begin{cases} \neq 0 \\ < offset \end{cases}}$$

```
1.          float interaction = tex2D(_BlobTex, i.uv).r ;
2.          float4 layer1 = (tex2D(_BumpMap,i.uv.xy * _BumpMap_ST.xy +_BumpMap_ST.wz +
3.                                                                  offset1*(1-interaction)));
4.          float4 layer2 = (tex2D(_BumpMap,i.uv.xy * _BumpMap_ST.xy +_BumpMap_ST.wz +
5.                                                                  offset2*(1-interaction)));
```

Listing 5.19: Adding the interaction texture inside the computation of the flow map perturbance

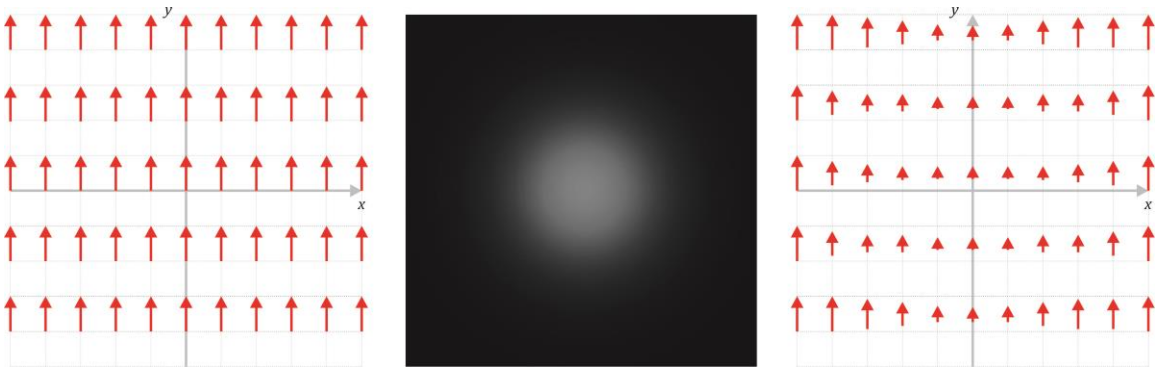


Figure 5.26: Vector field changes based on texture - a) Vector field $F(x, y) = < 0,1 >$; b) Texture image holding values in the range [0,1]; c) multiplication of the vector field with the image.

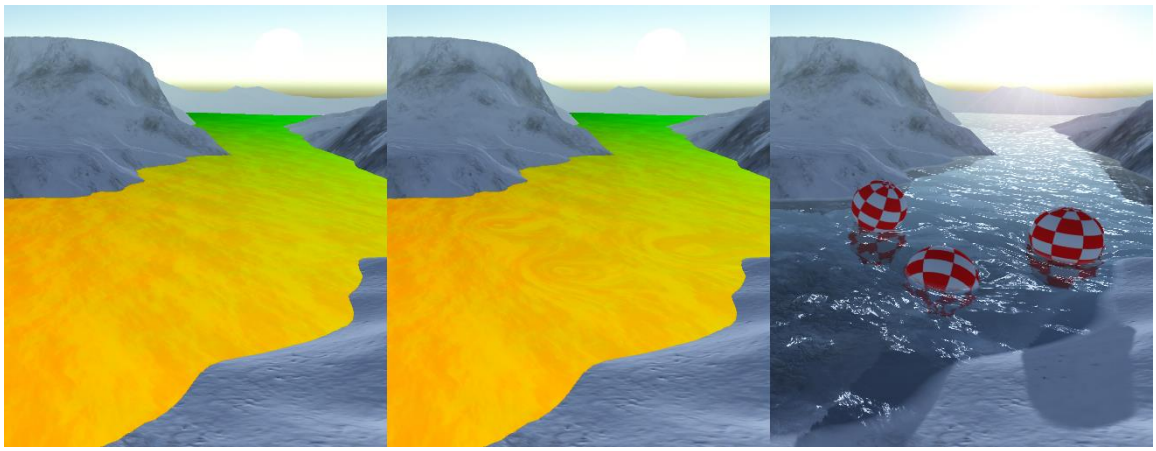


Figure 5.27: Interaction texture debugging - a) flow vector field coordinates created by the RG channels from a flow map; b) same vectors multiplied with the interaction texture; c) the final rendering of the water surface including all techniques.

5.3 Physics

For implementing all physics we use the build in physics engine of Unity3D, in order to simulate the inverse interaction – water to object. Each object gets detected by the physics engine when it has colliding and rigid body components, automatically adding properties like mass, gravity force, drag force and collision detection. The only missing ones to create close to real-life physics simulation are the buoyancy and the water stream force.

5.3.1 Buoyancy

We use the cube approximation technique described in the Section 3.9 Physical Models. Each object, affected by this type of force, needs pre-computational cube approximation, in order to create local points from the model space and then apply forces to these points. To approximate geometry, we use the boundaries of the object as a coordinate space. We then divide each axel of this space n times, resulting a total number of approximation cubes equal to n^3. To proper distribute the cubes, before any approximation cube (voxel for short notation) is created, the object is placed at world or parent object space coordinates (0, 0) and rotated to Euler angles (0, 0, 0). The final output is only the centers of these cubes, because we only need the center of the cube, where we are going to apply all forces, see Figure 5.28 and Listing 5.20.

If we consider an object with geometry bounds, respectively $x_{min}: x_{max}$ and $y_{min}: y_{max}$, and also point $A(x, y)$ is a point inside the space defined by the geometry bounds, then A can be found by Equation 5.8, where A is also a center of one of the voxels:

$$\boldsymbol{A_x = x_{min} + (B_x/n) * (0.5 + I_c)} \quad (5.8)$$

, where B is the size of the bounds and I_c is the current iteration (assuming that the cubes are being created from the center of the bounds in outward direction).

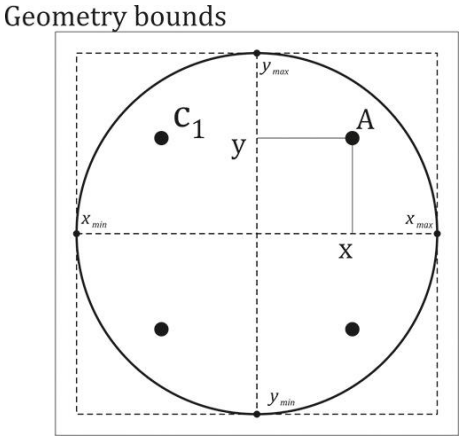


Figure 5.28: Cube approximation of a 2D circle - same algorithm will work for 3D shape as well.

```
1.  private List<Vector3> CubeApproximiation()
2.  {
3.          int totalNumberCubes = slicesPerAxis ^ 3;
4.          var cubeCenters = new List<Vector3>(totalNumberCubes);
5.          var bounds = myCollider.bounds;
6.          for (int iterationsX = 0; iterationsX < slicesPerAxis; iterationsX++)
7.          {
8.                  for (int iterationsY = 0; iterationsY < slicesPerAxis; iterationsY++)
9.                  {
10.                         for (int iterationsZ = 0; iterationsZ < slicesPerAxis; iterationsZ++)
11.                         {
12.                                 float x = bounds.min.x + (bounds.size.x / slicesPerAxis) *
13.                                                                         (0.5f + iterationsX);
14.                                 float y = bounds.min.y + (bounds.size.y / slicesPerAxis) *
15.                                                                         (0.5f + iterationsY);
16.                                 float z = bounds.min.z + (bounds.size.z / slicesPerAxis) *
17.                                                                         (0.5f + iterationsZ);
18.                                 var p = transform.InverseTransformPoint(new Vector3(x, y, z));
19.                                 cubeCenters.Add(p);
20.                         }
21.                 }
22.         }
23.         return cubeCenters;
24. }
```

Listing 5.20: Method that creates a list of voxel centers, based on cube approximation.

After we find all voxel centers, we need to compile the buoyancy. In order to do that, we need to specify object's density per instance, because the Unity3D physics engine does not provide us with that property. As regards the volume, we can calculate it from the density Equation 5.9. The buoyancy we have calculated is for the entire object, thus we need to divide the value by the number of voxels in order to get the amount of force equally distributed in local space for each voxel. Here by local we mean only that part of the buoyancy that takes into consideration the direction of the force. This stage of the calculations is pre-computed as well (implemented in the *Start()* method), given in Listing 5.21.

$$\boldsymbol{\rho} = \frac{\boldsymbol{m}}{\boldsymbol{v}} \qquad (5.9)$$

```
1.          float volume = myRigidBody.mass / objectDensity;
2.          float buoyancy0  = WATER_DENSITY * Mathf.Abs(Physics.gravity.y) * volume;
3.          localBuoyancy0 = new Vector3(0, buoyancy0 , 0) / voxels.Count;
```

Listing 5.21: Calculating additional properties per object

The actual apply of the full buoyancy force is implemented inside the looping function, called once per frame – *FixedUpdate(),* because we want this force to be true whenever the object is in contact with the water. The algorithm iterates through all the voxel centers (Figure 5.29) and calculates the full buoyancy force. Taking into consideration the distance between the water level and the position of the current iterated voxel center, provides us with a value of how much the current approximated cube is inside the water (in the case of

3D space – how much below the water surface, or based on comparison between the Y-axel position)

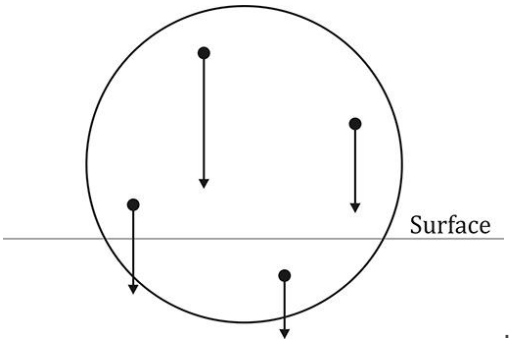


Figure 5.29: Force application points

5.3.2 Stream force

In order to apply velocity changes on the objects inside the river, we need a way to access the information that creates that river flow – in our scenario this is the flow texture. Therefore, we need a method that can extract values from the flow texture, similarly to the flow map technique used on the GPU. For each object affected by the stream force, we cast a ray from the center of it towards an invisible layer, where only one plane is rendered. Using the same size as the water plane and a shading model that excludes all lighting, but maps only the flow texture to the models UV coordinates. The *RaycastHit* struct in Unity3D can hold information of the casted ray, and more particularly the coordinates of where the ray hit a collider – in our case this is the flow texture plane. Based on these coordinates and the flow texture itself, we can extract the color values from the flow texture, just like we did in the flow map technique. Using these color values, we created a vector that holds the information of how the stream is flowing. Thus, we apply it as direction multiplied by a water force magnitude value (see Figure 5.30). All of the above statements were also implemented inside the *FixedUpdate()* function.

$$\boldsymbol{F_{stream} = V[texture(x), texture(y)] * F_{flow}}$$

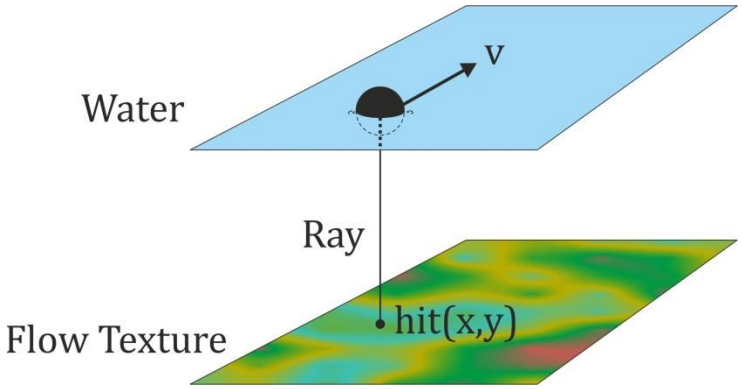


Figure 5.30: Stream force direction - extracting values from the flow map in order to apply stream force to the water flow.

5.4 Shading tree

In this section we will present the final output of what we accomplished in a simplified way – using tree structure. This type of presentation is widely used for presenting shader programs – shading tree (Figure 5.31). It will give a basic overview of what we accomplished, in respect to the visuals and interaction (since we are implementing the interaction on camera-rendering basis). As regarding the physics, we will also display the steps inside a code design structure tree that will include only the most important functions used – see Figure 5.32.

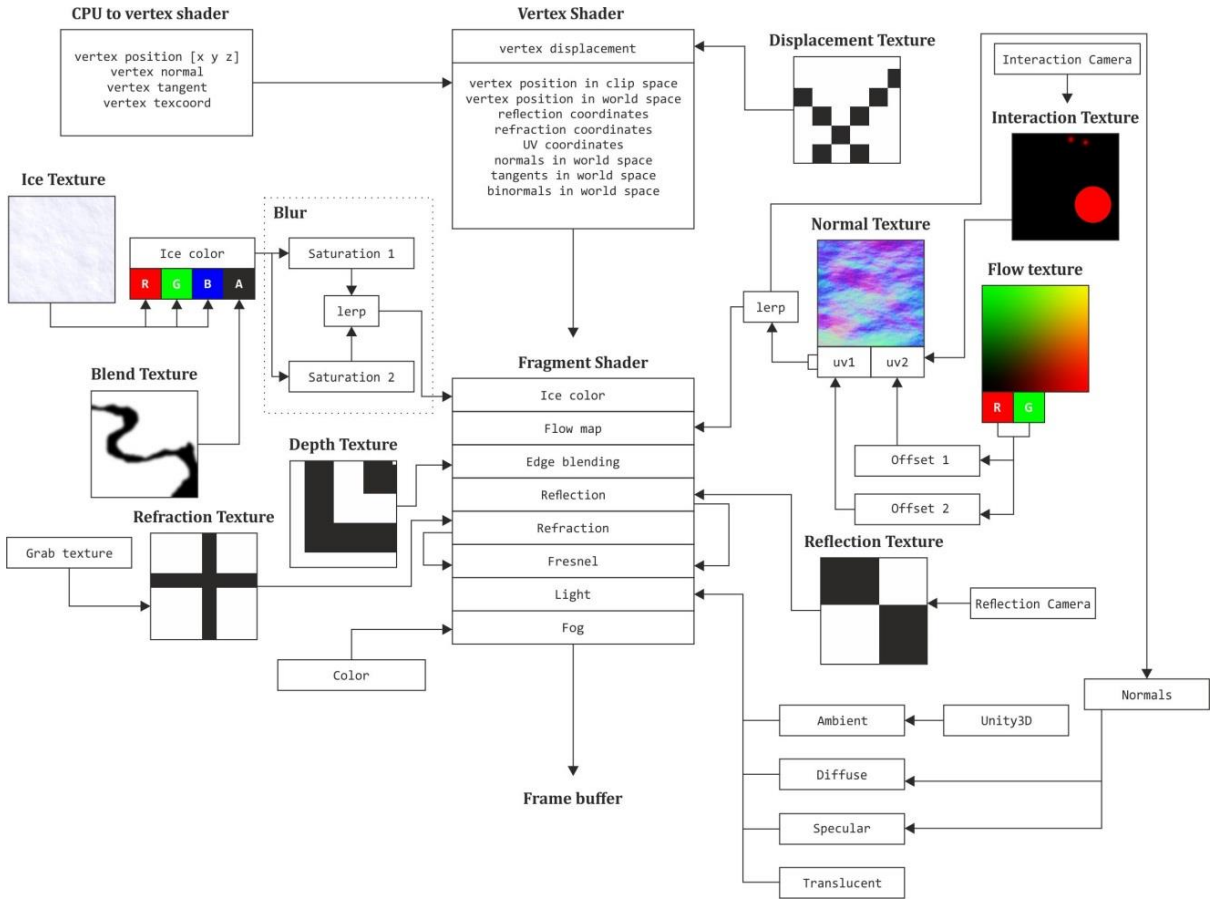


Figure 5.31: Shading tree

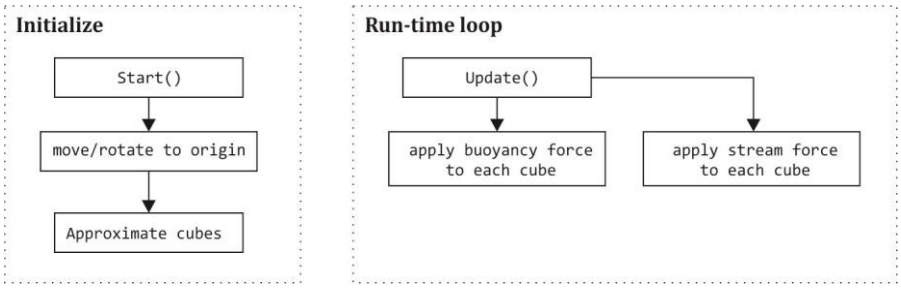


Figure 5.32: Code design structure tree

6. Testing

6.1. Test design

The origin of our work is divided into three sections, namely visuals, interaction and physics. The first two are implemented on the GPU while the last runs on the CPU. Therefore, we need to do a performance testing that will give us precise data to evaluate. Since the main criterion, used for comparison, is the computational time, we will not be using Unity3D's FPS, since this characteristic involves not only the rendering time, but also the processing of everything included in the scene. To measure the time, needed for our water to render, we will use deep profiling inside the Profiler tool of Unity3D – sections GPU and CPU. The results will be displayed in milliseconds μs. To have accurate results, we also use a number of iterations taken from each measurement and then the average is taken as input. If we consider 6 iterations with interval of 30 frames, this would result in observing 180 consecutive frames, or three seconds of animation with target FPS=60. Another importance, taken into consideration, is that Unity3D profiling tool provides labeling of the processes only for the CPU, meaning that we have no clear information about the GPU skinning – we only get results about Opaque, Transparent geometry processing and other types of graphical events, computed on the GPU. To deal with this, we discard (remove from scene) any other type of data that will be processed by the GPU, leaving only the water shading inside the scene. Also to check the retrieved data, we will perform 2 independent measurements of the data (later referred to as *First Run* and *Second Run*) and compare it in a T-test for significance of the results.

We divide the test into two stages. In the first part we will measure computational time of each individual implemented technique when it is included inside the overall computational time against when it is not. (see Chart 6.1) We expect this to provide us with knowledge of how expensive are different techniques, regarding computational time and more importantly what can be further optimized/expanded based on the results, or in other words what is the impact of each individual effect to the overall. Apart from the computational time, we also include the number of mathematical instructions inside the vertex and the fragment shader, needed to produce the current observation effect using DirectX11's 3D graphics API – d3d11.dll. This statistical measurement is provided by the compiled version of the shader program.

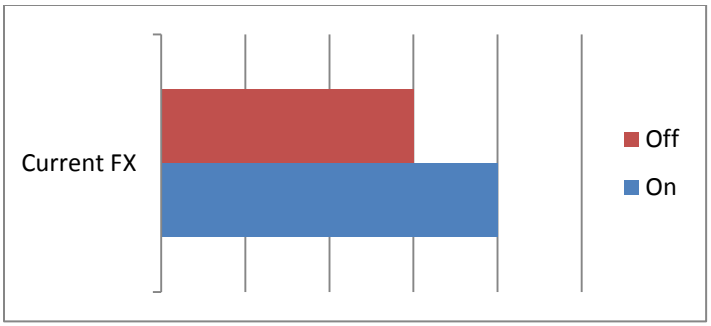
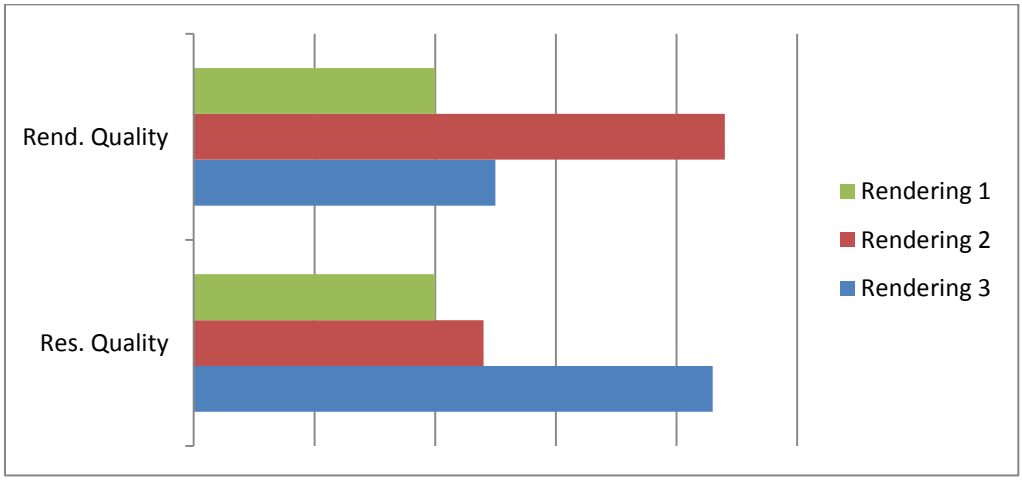


Chart 6.1: Comparison of turned on and off effects, measuring computational time of each technique individually.

The second stage of the testing takes into consideration each individual effect and compares it against the following parameters:

- Number of objects affected – will provide us with data about the impact of having large number of objects. This is extremely important for the interaction texture, as well as reflection and refraction.
- Water plane geometry grid size – the change of the amount of vertices. This is important for testing the z-displacement.



The testing is carried on mobile PC with Intel® Core™ i7-4500U CPU and GeForce GT 750M with 4GB of memory (1000MHz,128Bit), 384 Shading units, 32 texture mapping units, 7.6GPixel/s, 30GTexel/s.

6.2. Expectations

Prior to testing, our predictions are purely based on the amount of computation instructions given inside the corresponding programs. Most of the visual effects are implemented inside the fragment shader (formulas for Fresnel, view direction, light direction and etc.), which we think will produce significant computational requirements. The most computationally demanding features should be the ones that use extra rendering of the scene, namely – reflection, refraction and interaction. Regarding the CPU processes,

computing rigid bodies is not that heavy, although the process of applying forces to a number of cubes, results in a looping calculation inside an update function (also looping once per frame). Having a large number of objects interacting with the water, (interaction texture) should not have any impact on this feature computational time, as its output will always be just an extra texture sent to the shading process of the water.

6.3. Results

The following Table 6.1 shows the statistical evidence that the collected data is reliable, based on the fact that there is no difference between the two observation trials. Any evaluation can be based on that proof. All t-test are carried using a paired samples (assuming that the data collection comes from one place). We also use a standard significance probability value $\alpha = 0.05$.

T-Test Nr. ONE				
	Point of view:	We want to confirm that the measured data is reliable		
	Considering:	The case when all visual features of the water simulation are turned on		
	Goal:	Run hypothesis test to provide statistical evidence to detemine the reliability of the data collected		
Step 1: Hypothesis list				
	H_0:	The data is reliable, not affected by system changes		
	H_1:	The data is not reliable, affected by system changes		
Step 2: Significance				
	α value:	0.05		
Step 3: Test calculations		Using Paired two-sample T-test, because the data is mined under the same circumstances, twice (same computer)		
			First Run	Second Run
		Mean	5.569	5.629
		Variance	0.008	0.006
		Observations	5	5
		degree of freedom	4	
		t Stat	-1.067	
		p(T<=t) one tail	0.173	
		t Critical one-tail	2.132	
Step 4: Test conclusion		Because the p-value of 0.173 > α, we reject H_1. The statistical evidence does not suggest that there is any statistically significant difference between the two trials - First Run and Second Run		

Table 6.1: T-test 1

The rest of all statistical tables can be found in the Appendix Section. For simplicity of the writing, herein we will only include the results of each individual T-test performed, considering every VFX in the rendering of the water simulation, see Table 6.2.

T-test (α=0.05)	Excluded feature	Significance (p-value)	Significant difference
	Reflection	0.219	No
	Refraction	0.180	No
	Fresnel	0.451	No
	Flowmap	0.387	No
	Light	0.051	No
	Fog	0.139	No
	Ice	0.200	No
	Z-displacement	0.468	No
	Edge blend	0.051	No

Table 6.2: Individual T-test for every VX feature.

Based on the previous data validation, we can now assume that every group of frames we take (and moreover their average), would be statistically correct and can be used to make evaluations, based on that set of data. For this purpose, we will use the average values of both sets of data combined, regarding the computational time, measured in milliseconds (Table 6.3).

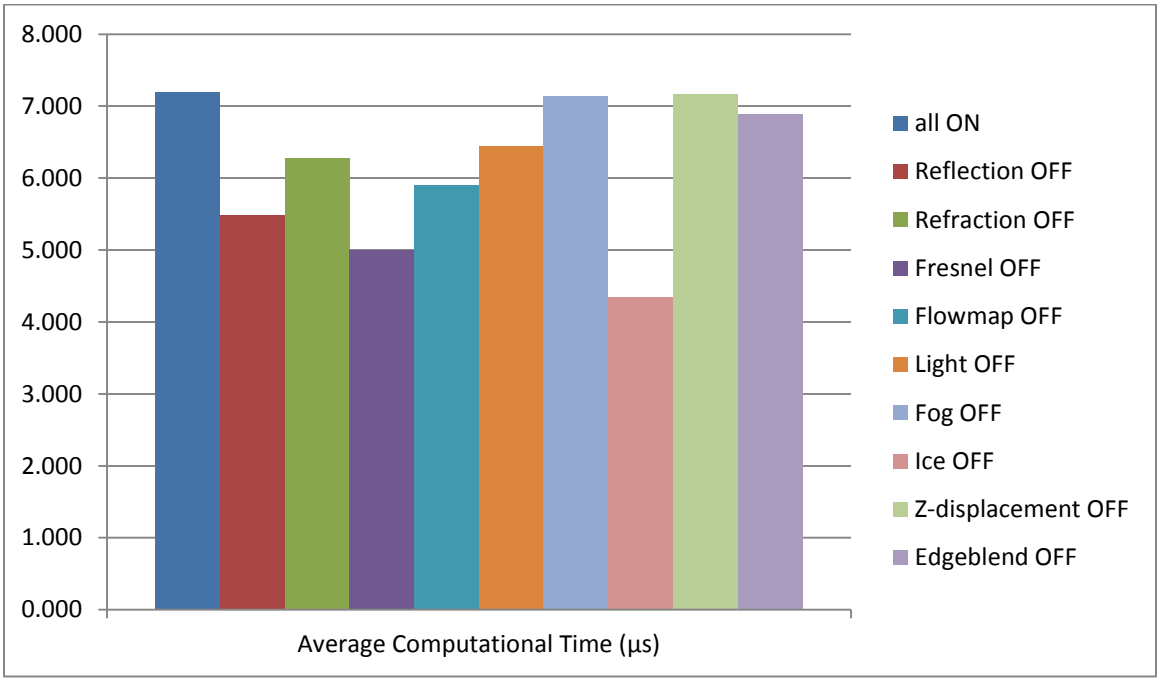


Table 6.3: Average computational time.

The next data, we have collected, was based on the compiled version of the shader program used for the rendering, namely vertex and fragment mathematical instructions count as well as number of textures used (Table 6.4).

Observation	vert instructions	frag instructions	vert tex	frag tex
ALL FX ON	39	121	2	12
No REFL	39	119	2	11
No REFR	39	119	2	11
No Fresnel	39	102	2	8
No Flowmap	39	103	2	8
No Light	39	69	2	11
No Fog	39	113	2	12
No Ice	39	77	2	5
No Z-disp	34	121	0	12
No Edge	39	107	2	11

Table 6.4: vertex and fragment instruction/texture count

The second test, described in the test design was taken throughout the same data validation procedure as in the previous/first test. Therefore, each measurement was performed twice and t-tested in the same logical approach (Table 6.5), regarding grid size (Table 6.6).

T-test (α=0.05)	Grid Size	Significance (p-value)	Significant difference
	10x10	0.173	No
	16x16	0.435	No
	32x32	0.091	No
	64x64	0.241	No
	128x128	0.063	No

Table 6.5: T-test for validating the data taken with different water grid-size

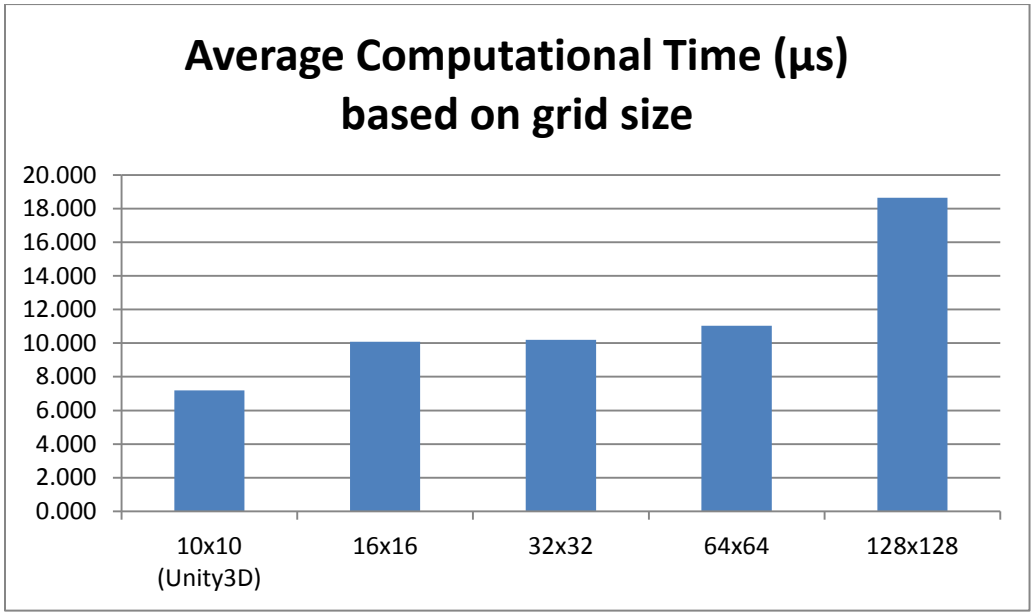


Table 6.6: Avarage computational time based on grid size.

Regarding the last measurement, namely objects in the scene (affected by the full physical model we implemented), we took the average measurement of the rendering time of the water, the objects and the reflection independently from each other and we compared them, as seen in Table 6.7.

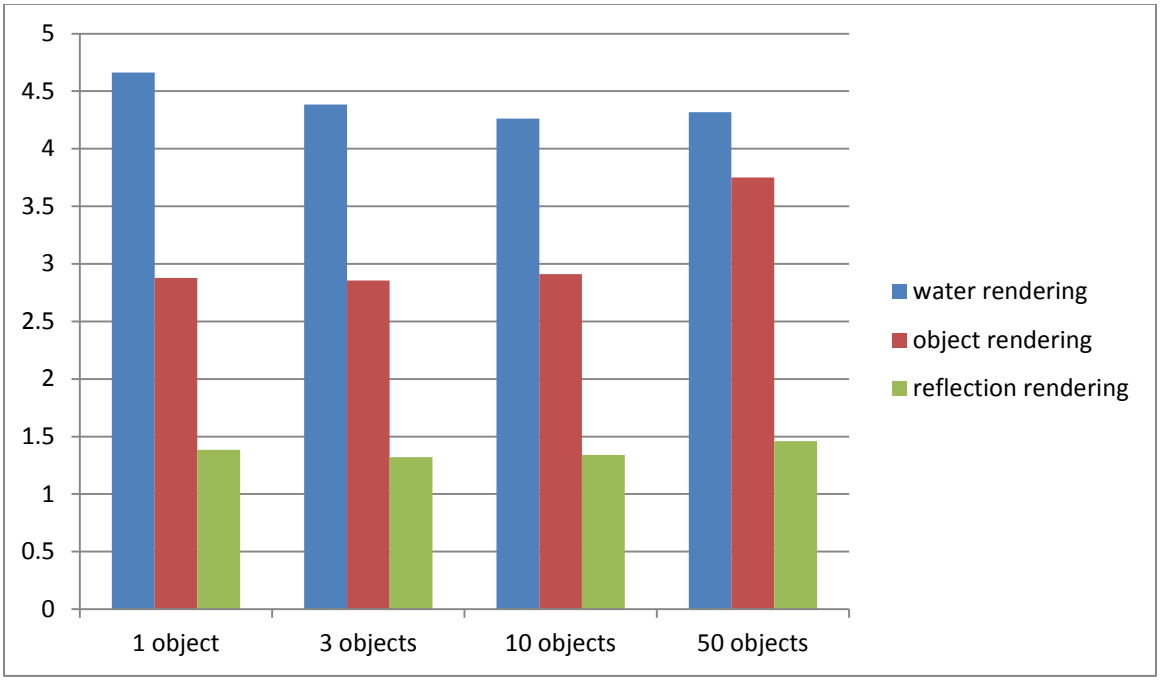


Table 6.7: Measuring separate tasks – water rendering, opaque objects rendering, reflection image rendering

7. Discussion

Throughout the development of this project, we managed to experiment with a number of techniques. We will iterate throughout all of them based on the results we gathered in the previous section.

- Reflection – this is probably one of the most complex feature when simulating water, as it takes a great number of functions to accomplish (mostly on the CPU), as well as creating a second rendering of the scene with a different point of view. Having a second camera to render, means that the RTC reflection is depended on the number of objects in the scene, more precisely – the vertex and poly count. Despite that, the reflection image (in real-time) produces a high level of realism in the water rendering at the cost of a steady computational time (Table 6.7 – reflection rendering). RTC reflection has less to do with the actual shader program, as it only sends a single 2D texture data to be included inside the fragment color, requiring a small fraction of mathematical functions (Table 6.4 – fragment instructions). Having a real-time reflection in modern applications

is not impossible, based on the fact that the computational time needed to produce that type of reflection is considerably small – around $1.5\mu s$ (Table 6.7-reflection rendering)

- Refraction – as mentioned earlier there are a few concepts, when creating refraction and the one we used is significantly expensive – requiring around $1\mu s$ to compute. It again produces just a single image data that is used inside the computation of the fragment color. We have only tested the scenario of using a grabpass texture, provided by Unity3D. Therefore, we cannot conclude which method for refraction will yield best results, regarding performance. Based on our findings, we can assume that this approach is providing significant and stable result. We have noticed one small problem, connected more with the visual cue – objects on top of the water also get captured inside the GrabPass image. For this reason, their color is also taken inside the refraction computation, leading to appearance of artifacts. What we found interesting is that using GrabPass, allows us to preserve the shadows, thrown from objects above the water surface.
- Fresnel effect – the considerable decrease of computational time seen in Table 6.3 is due to the fact that the Fresnel effect takes into consideration the view direction and interpolated normal vectors, meaning that obtaining/computing these values per pixel entry requires significant computational time – approximately $2.1\mu s$. Using different formula for computing the Fresnel effect yields different results, depending on the needs (either more realistic or more artistic). In any case, computing the Fresnel effect should be implemented per fragment basis, as the above stated variables connected with its computation, are also used for the light computation. Thus, there is no need to compute them twice (once inside the vertex shader and secondly in the fragment shader). Disabling the Fresnel computation, reduces the mathematical instructions with around 1/6 from the total count, which based on all shader implemented calculations is within normal boundaries.
- Flow map – the results from Table 6.3 shows that excluding the flow map technique from being computed, reduces the overall computational time by only $1.2\mu s$. With around the same number of mathematical instructions as the Fresnel effect, (around 20) the use of this technique shows significantly good computational requirements. It is based on the assumption that it is used for perturbing the main visual cue of the water – forming visual movement of the water waves and fluids. Our initial expectations were not met, as the algorithm of computing the flow mapping makes use of high number of textures and especially an *if-statement*, which is computationally unwise solution, when parallel processing is applied.
- Light – computing the light requires a great amount of mathematical calculations as seen from Table 6.4 – around ½ from the total amount of the count. Despite that, light computation doesn't have any significant impact on the overall computational time – less than $1\mu s$. We assume this is due to the fact that computing light has nothing to do with using external data (textures) unless bump map is used. In our case, the bump map is computed inside the flow map technique. We should also consider the fact that our model makes uses of significantly more math instructions than normally a light model would use, as we produce two light calculations – one for flat surface and one for bumped surface, producing more artistic results.

- Fog – just like light computation, fog computation does not use any external input, rather a set of simple mathematical instructions – around 1/12 from the total amount, leading to almost no difference in the overall computational time, when turned off.
- Ice – results in Table 6.3 and 6.4 indicate that this feature requires significant amount of computational time – around $3\mu s$, using almost half of the total amount of textures and instructions used inside the fragment shader. This is due to the fact that we used a blurred and smoothed transition between 2 stages of rendering the ice and snow cover, when the water freezes. Reducing the number of used textures, (texture packing using texture channels) should reduce the computational time. We should also consider the fact that the ice computation includes the use of second normal map (different one from the water normal map).
- Z-displacement – this feature has almost no impact on the GPU computational time, due to the fact that its value is only being passed from the CPU and used inside the vertex shader to translate the current model position of each vertex based on that value. The reduction in the number of textures, used in Table 6.4, is due to the use of an image data to randomize the passed between the CPU and GPU value.
- Edge blending – using edge blending shows a small reduction in the computational time in Table 6.3. As up till now, everything rendered on the screen is entirely opaque, with zero transparence (even the refraction technique does not use transparency). Having edge blending requires the assumption that the fragment behind the current one is still held somewhere in the memory, resulting in having 2 sets of information per fragment. Despite that, this technique has significant output, regarding visual cueing.

Aside from the algorithms for rendering the water plane, we also measured external factors, affecting the final output indirectly. Table 6.6 shows us results when we increased the grid size, resulting in having higher amount of vertices to process. As far as this is concerned, we believe that the amount of difference we received in the data (especially when using a 128 by 128 grid) is due to the amount of information passed from the vertex shader to the fragment shader. In other words, more vertices mean more data to be interpolated between. At current stage this cannot be optimized at all, as each set of data is used for specific purpose. However, all of the images in this writing were produced using the Unity3D built-in 3D plane (grid 10x10), which lead to significantly good results. Having higher grid is unnessecery, unless we want to have more per-vertex operations. One such use is connected with the z-displacement, since it operates by applying transformation of each vertex. Thus, higher grid is equal to more distinguishable waves.

The second “external” testing provided us with important data about the interaction of objects with the water. As mentioned earlier, this technique can end with extreme amount of computational power required. But as we see in Table 6.7, having higher amount of objects inside the scene has almost no impact on the computation of the water. The reflection rendering has just a small increase, due to the fact that is a render texture – higher count of objects rendered is equal to having higher amount of vertices to process. As regarding of the decrease of the water rendering time, this can be explained simply by the fact that less

fragments from the water surface are being computed, due to the depth testing (the opaque objects are covering portions of the water).

8. Conclusions

Our results indicated that producing a model of water simulation that relies on the flow between visuals, physics and interaction can be achieved at significantly low computational costs, allowing extra features to be added. We included one such, namely a transition of the water states. Although it is more a visual simulation that can be executed at run-time, rather than an actual life-like model, we think that even better and more optimized results can be achieved in further investigation. We reached our goals of achieving high FPS value in a scene, where water simulation was elicited, based on a number of techniques – reflection, refraction, flow mapping and etc.

While other researches are focusing solely on a specific part of water simulation, we induced a proposal for a flow between three different topics, namely visuals, physics and interaction. We produced a final output based on a combination between these three key areas and even further expanded the results by including the transition between water liquid and solid state. The simplified solution we introduced in this paper can be utilized mainly inside gaming applications or virtual environments, but we believe that a higher precision model can be accomplished and used even for scientific purposes and simulations.

9. Future work

There is a set of areas that can be potentially optimized in order to increase the performance even further. Also, some ideas were brought into consideration during the design process, but were left out due to complexity. We will include this as a list of bullet points that can be further worked onto:

- General optimization – currently we did not invest any significant time into lowering the precision of the float variables, as well as ways to reduce math instructions.
- Optimization of the ice transition by reducing the number of textures mappings used, could lead to much better results. Another algorithm for blurring the transition of the water states should be used, rather than performing the same operation twice with different arguments, as it is currently.
- Pack normal textures for ice and water into one texture, rather than using two independent – using two channels per texture will provide enough data to reconstruct the normals for each individual normal texture.
- Use real-time object data for creating the interaction texture – currently each interaction with the water object has a decal representation, rendered inside the interaction texture, rather that we can use projection of each object onto the interaction plane and then use that data as interaction texture.
- Spray and Foam – additional FX used to create even higher impression of realism when objects collide with the water surface.
- Liquid interaction – up to this point, we assumed that only solid objects can interact with the water. If we consider an object that moves through space and it also applies pixel drawing along that path, we can then use the rendered output in combination with the interaction texture, resulting in a simulation of liquid being spilled inside the water surface. We find this as quite a challenging and at the same time interesting problem.
- Tessellation – make full use of d3d11 library by including the tessellation shader to produce more geometry at run time for better z-displacement waves and proper shadow-light mapping. This would bring the results of this paper to much higher level.
- Underwater – rendering of the water when the camera position is below the water surface plane. This would include the need of another set of techniques to be included – for example different depths of field of the camera below and above the water and other more complex methods.
- Post processing the most light glancing places of the water waves using star filter technique (an extension of the bloom technique) that can produce extreme life-like visual cues.

10. References

[1] Larsson,Thomas. (2002-03-13). *"Scientific Foundations of Computer Graphics"*, Department of Computer Engineering, Mälardalen University Sweden MRTC Report, ISSN 1404-3041 ISRN MDH-MRTC-67/2002-1-SE.

[2] Kore, Dhruv; Gonzalez, Giancarlo; Sum, Jenny; Professor Forbes, Angus *"Introduction to Computer Graphics"*. Fluid simulation

[3] Golder, Dave. (2011-10-04). *"CGI Visual FX: Great Leaps Forward"*, article

[4] Wong, David (2007-12-10). *"A Gamer's Manifesto"*. Cracked.com. Retrieved 2014-11-18.

[5] Thorn, Alan (2009-06-23). *Cross Platform Game Development*. Jones & Bartlett Publishers. p. 251. ISBN 0763782815.

[6] Harris, John (2007-09-26). *"Game Design Essentials: 20 Open World Games - Air Fortress"*. Gamasutra. Retrieved 2008-08-02.

[7] Wissmath, B, Weibel, D., & Groner, R. (2009). *Dubbing or Subtitling? Effects on Spatial Presence, Transportation, Flow, and Enjoyment*. Journal of Media Psychology 21 (3), 114-125.

[8] Martin, James (1965). *Programming Real-time Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall Inc. p. 4.ISBN 013-730507-9.

[9] Jiawei Han; Micheline Kamber (2011-06-09). *Data Mining: Concepts and Techniques.* Elsevier. p. 159. ISBN 978-0-12-381480-7

[10] *"Pygmalion's Spectacles*". Project Gutenberg

[11] CGW, "*CG-Revolution*" (2012-10), Issue 6, Vol.35

[12] Caulfield, Brian. (2014-09-18). "*How Our Maxwell GPUs Debunked the Apollo 11 Conspiracy Theory*"

[13] BRIDSON, R. 2009. "*Fluid Simulation For Computer Graphics*". A.K Peters.

[14] Batchelor, G.K. (1973), "*An introduction to fluid dynamics*", Cambridge University Press, pp. 71–73.

[15] Braley, Colin. "*Fluid Simulation for Computer Graphics: A Tutorial in Grid Based and Particle Based Methods*"

[16] M. J. Gourlay. (2009). "*Fluid simulation for video games* "(part 1-15)

[17] Chentanez, N. and Muller, M. (2011). "*Real-time eulerian water simulation using a restricted tall cell grid*," in ACM Transactions on Graphics (TOG), vol. 30, p. 82, ACM.

[18] Buchholz, J. (2008). "*Matlab Particles 2.1"*

[19] Finch, M. (2004) "*GPU Gems: Chapter 1. Effective Water Simulation from Physical Models*", NVIDIA Corporation

[20] Müller-Fischer, Matthias. (2008) "*Fast* Water *Simulation for Games. Using* Height Fields." GDC

[21] Cords, H. (2009) "*Real-Time Open water environments with interacting objects*", Eurographics Workshop on Natural Phenomena

[22] Christen, Martin. (2009-01-26). *"Clockworkcoders Tutorials: Vertex Attributes"*. Khronos Group.

[23] https://www.opengl.org/wiki/Main_Page

[24] Cg programming 2012 http://en.wikibooks.org/wiki/Cg_Programming/Programmable_Graphics_Pipeline

[25] OpenGL official page 2013, "*Swap Intervals*" https://www.opengl.org/wiki/Swap_Interval

[26] Fernando, R. (2005). "*The Cg Tutorial – the definitive guide to programmable real-time graphics*", NVidia Corporation

[27] "*Kinematic Analysis*". http://en.wikipedia.org/wiki/Inverse_kinematics

[28] Why does the ocean appear blue? Is it because it reflects the color of the sky?"(1999-10-21). Scientific American.

[29] "*Water Transparency*" RMB Environmental Laboratories, Inc., article

[30] Lekner, John (1987). *Theory of Reflection, of Electromagnetic and Particle Waves.* Springer. ISBN 9789024734184.

[31] "*Shoaling, Refraction, and Diffraction of Waves*". University of Delaware Center for Applied Coastal Research. Retrieved 2009-07-23.

[32] Birn, Jeremy. "*Digital Lighting & Rendering*"

[33] Gietler, Scott. (2013). "*Underwater Lighting Fundamentals*"

[34] Young, Jim. "*Bubbles, Foam and Ocean Spray*"

[35] *What is Buoyancy?*"(2008) MIT Sea Grant College Massachusetts Institute of Technology

[36] Muldrew, Ken. (1997). "*The Freezing Process*".

[37] Voorhies, D. (1994), "*Reflection Vector Shading Hardware*", SIGGRAPH, ACM Press

[38] Blinn, Jim and Newell, Martin. (1976). "*Texture and Reflection in Computer Generated Images*". Communications of the ACM, Vol.19, No. 10, pp. 542–547.

[39] Greene, Ned. (1986-11). "*Environment Mapping and Other Applications of World Projections*". IEEE Computer Graphics and Applications, Vol. 6, No. 11, pp. 21–30.

[40] Grootjans, Riemer. "*3D Series 4: Advanced Terrain*", http://www.riemers.net/

[41] Belyaev, V. (2003). "*Real-time simulation of water surface*", GraphiCon Conference pp.131-138

[43] Sousa T., (2005)"*GPU Gems 2: Generic Refraction Simulation*" NVIDIA Corporation, pp. 295-305

[44] Wikipedia, "*Householder transformation*"

[45] A. Howard. (1994). *Elementary Linear Algebra* (7th ed.), John Wiley & Sons, p. 155

[46] Sousa T., (2005)"*GPU Gems 2: Generic Refraction Simulation*" NVIDIA Corporation, pp. 295-305

[47] Wikipedia "*Dispersion (optics)"* http://en.wikipedia.org/wiki/Dispersion_%28optics%29

[48] K. Mitchell. "*GPU Gems 3: Volumetric Light Scattering as a Post-Process*",

[49] Wikipedia *"Stencil buffer"*

[50] Unity Documentation 2015, "*ShaderLab Syntax: Stencil*", Unity Technologies, http://docs.unity3d.com/Manual/SL-Stencil.html

[51] Lengyel, E. (2005) "*Oblique View Frustrum Depth Projection and Clipping*", Journal of Game Development, Vol. 1, No. 2, Charles River Media, pp 5-16

[52] Wikibooks (2012) "*CG programing: Computing the projection Matrix"*, http://en.wikibooks.org/wiki/Cg_Programming/Vertex_Transformations

[53] Birn, J. (2013) "*Digital Lighting & Rendering (3rd Edition)*"

[54] Grootjans, R. "*3D Series 4: Advanced Terrain*", http://www.riemers.net/

[55] Schlick, C. (1994) "*An inexpensive BRDF Model for Physically-based Rendering*", Computer Graphics forum 13 : 233

[56] Wolfgang, F. "*ShaderX2: Shader Programming Tips & Trics with DirectX 9*"

[57] Gomez, Miguel. (2000)"*Interactive Simulation of Water Surfaces*" in "Game Programming Gems", ed. Mark DeLoura. Charles River Media, p 187-199

[58] Max N., Becker B. (1995). "*Flow visualization using moving textures*", Proceeding of the ICASW/LaRC Symposium on Visualizing Time-varying Data, 77-87

[59] Vlachos, A. (2010) "*Advances in Real_time Rendering in 3D Graphics and Games*", Parts I and II, SIGGRAPH

[60] Bell, John & Johnson, Andy. (2005) "*Lightning and Shading*".

[61] F.Kane (2014), "*3D Water Effects*", http://gamasutra.com/

[62] Wikipedia "*Buoyancy*" http://en.wikipedia.org/wiki/Buoyancy

[63] Lorensen, W. (1987), "*Marching Cubes: A high resolution 3D surface construction algorithm*", In Computer Graphics, Vol.21 Nr.4

[64] Baerentzen, J. "*Isosurface polygonization*" http://www2.compute.dtu.dk/~janba/

[65] Carlson M., Mucha P. and Turk G. (2004). "*Rigid fluid: animating the interplay between rigid bodies and fluid*," in ACM Transactions on Graphics (TOG), vol. 23, p. 377{384, ACM.

[66] Muller-Fisher, M. (2008), "*Fast Water Simulation for Games Using Height Fields*", Games Developers Conference 2008

[67] Elias, H. "*2D Water*", http://freespace.virgin.net/hugo.elias/graphics/x_water.htm

[68] Xiong, Y. (2013), "*Fast and stable simulation of virtual water scenes with interactions*", Virtual Reality 17, p. 77-88

[69] Laramee, R. (2006), "*Texture Advection on Stream Surfaces: A novel hybrid visualization applied to CFD Simulation Results*", Eurographics 20012

[70] Wijk, V. (2002), *"Image based flow visualization*", ACM Transactions on Grpahics 21, p. 745-754

[71] Dholakia, R. "*Interactivity and revisits to websites: a theoretical framework*"

[72] Craitoiu, S. (2014). "*Create a fog shader*", in2gpu.com

[73] New Hampshire Lakes Association (2011). "*Why do rivers take longer to freeze than lakes*?" www.nhlakes.org

11. Appendix

11.1. Water shader code

- Inspector controlling parameters

```
Shader "Svego/IcyWater01"
{
    Properties
    {
        [HideInInspector]_Checkers ("test", 2D) = "white" {}
        [HideInInspector]_ReflectionTex("Reflection Texture", 2D) = "white" {}
        _FresnelCorrection("Fresnel Correction", Range(0,2)) = 1.3
        _BumpMap("Bump Map", 2D) = "bump" {}
        _BumpAmplitude("Bump Amplitude", Range(0,3)) = 0.5
        _FlowMap("Flow Map", 2D) = "white" {}
        _NoiseMap("Flow noise", 2D) = "white" {}
        _Cycle("Time cycle", float) = 1.0
        _Translocation("Pixel Translocation", float) = 0.05
        _Fade("Edge fading", float) = 0.15
        _Foam("Foam Texture", 2D) = "white" {}
        _FoamAmplitude("Foam Amplitude", Range(0,1)) = 1
        _SpecColor("Specular Color", Color) = (1,1,1,1)
        _Shininess("Shininess", Float) = 10
        _Sharpness("Specular Sharpness", Float) = 10
        _BumpShadowAmplitude("Wave Shadow Intensity", Range(0,0.5)) = 0.2
        _DiffuseTranslucentColor("Translucent Color", Color) = (1,1,1,1)
        _ChannelFactor("Channel Factor", Vector) = (1,1,1)
        _DispTex("Displacement Texture", 2D) = "white" {}
        _DisplacementAmount("Displacement Amount", Range(0,0.5)) = 0.01
        _FogColor("Fog Color", Color) = (1,1,1,1)
        _FogDensity("Fog Density", Range(0,0.1)) = 0.03
        _SnowTex("Snow Texture", 2D) = "white" {}
        _SnowBump("Snow Normal", 2D) = "bump" {}
        _SnowBlend("Snow Blend Texture", 2D) = "white" {}
        _EdgeSharpness("Edge Sharpness", Range(0,10)) = 0.5
        _BlendAmount("Blend Amount", Range(0,1)) = 0.5
        _IceColor("Ice Color", Color) = (1,1,1,1)
        _Thickness("Ice thickness", Range(0,1)) = 0.0
        _BlobTex("Blob" , 2D) = "white" {}
    }
```

- Vertex input/output structure declaration

```
    SubShader
    {
        GrabPass
        {
            Tags{"LightMode" = "Always"}
        }

        Pass
        {
            Tags {"Queue" = "Transparent" "LightMode" = "ForwardBase"}
            Zwrite Off
            Blend SrcAlpha OneMinusSrcAlpha

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.0
            #include "UnityCG.cginc"
            .
            .
            .
            // variable declaration
            .
            .
            .
            struct vertexInput
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 tangent : TANGENT;
                float4 uv : TEXCOORD0;
            };
            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float4 reflUV : TEXCOORD0;  // used for reflection
                float4 refrUV : TEXCOORD1;  // used for refraction
                float3 normalWorld : TEXCOORD2; // used for normal vector direction
                float4 posWorld : TEXCOORD3;    // used for view vector direction
                float4 uv : TEXCOORD4; // used for uv texture coordinates
                float3 tangentWorld : TEXCOORD5; // used for bumpy light
                float3 binormalWorld : TEXCOORD6; // used for bumpy light
            };
```

- Vertex program

```
vertexOutput vert(vertexInput i)
{
    vertexOutput o;
// Z-Vertex Displacement
    float3 dTex = tex2Dlod(_DispTex, float4(i.vertex.xy + _DispTex_ST.wz*_Time.y,0,0)).xyz;
    float d = (dTex.r * _ChannelFactor.x + dTex.g * _ChannelFactor.y + dTex.b * _ChannelFactor.z);
    i.vertex.xyz = i.vertex.xyz + i.normal * d * _DisplacementAmount * (1-_Thickness);

// Data to fragment
    o.pos = mul(UNITY_MATRIX_MVP, i.vertex - float4(0,_Thickness*0.2,0,0));
    o.reflUV = ComputeScreenPos(o.pos);
    o.refrUV = ComputeGrabScreenPos(o.pos);
    o.posWorld = mul(_Object2World, i.vertex);
    o.uv = i.uv;
    o.normalWorld = normalize(mul(float4(i.normal,0),_World2Object).xyz);
    o.tangentWorld = normalize(mul(_Object2World, float4(i.tangent.xyz,0)).xyz);
    o.binormalWorld = normalize(cross(o.normalWorld, o.tangentWorld) * i.tangent.w);

    return o;
}
```

- Fragment program

```
float4 frag(vertexOutput i) : COLOR
{
// SNOW & ICE
    float4 snow = tex2D(_SnowTex, i.uv.xy * _SnowTex_ST.xy + _SnowTex_ST.wz);
    snow.a = tex2D(_SnowBlend, -i.uv.xy).r;
    snow.a += (_BlendAmount * 2 - 1);
    snow.a = saturate((snow.a * _EdgeSharpness - (_EdgeSharpness - 1) *0.5)*_Thickness);

    // same uv, but edge sharpness is lower to apply blurriness
    float4 ice = tex2D(_SnowTex, i.uv.xy * _SnowTex_ST.xy + _SnowTex_ST.wz);
    ice.a = tex2D(_SnowBlend, -i.uv.xy).r;
    ice.a += (_BlendAmount * 2 - 1);
    ice.a = saturate((ice.a * _EdgeSharpness/20 - (_EdgeSharpness/20 - 1) *0.5)*_Thickness);
    ice.rgb *=_IceColor.rgb;

    float4 finalIce;
    finalIce = lerp(snow,ice,1-snow.a);

// BUMP/FLOW DISTORTION - WATER
    float2 flowDir = tex2D(_FlowMap, -i.uv).rg * 2 - 1;
    flowDir *= _Translocation;
    float noise = tex2D(_NoiseMap, i.uv).r;
    float phase = _Time.y/_Cycle + noise.r * 0.1;

    float opacity = frac(phase);

    float2 offset1 = float2(flowDir.x,-flowDir.y) * frac(phase + 0.5f);
    float2 offset2 = float2(flowDir.x,-flowDir.y) * frac(phase);

    float interaction = tex2D(_BlobTex, i.uv).r;

    float4 layer1 = (tex2D(_BumpMap,i.uv.xy * _BumpMap_ST.xy +_BumpMap_ST.wz + offset1*(1-interaction)));
    float4 layer2 = (tex2D(_BumpMap,i.uv.xy * _BumpMap_ST.xy +_BumpMap_ST.wz + offset2*(1-interaction)));

    if( opacity > 0.5f)
        opacity = 2 * (1 - opacity);
    else
        opacity = 2 * opacity;


// NORMAL MAP DECODING
    float4 encodedNormalIce = tex2D(_SnowBump, i.uv.xy * _SnowBump_ST.xy + _SnowBump_ST.wz);
    float4 encodedNormalWater = lerp(layer1,layer2,opacity);
    float4 encodedNormal = lerp(encodedNormalWater, encodedNormalIce, _Thickness);

    float3 localCoordsNormal = float3(2*encodedNormal.a - 1,2*encodedNormal.g - 1, 0.0); // N(x,
    localCoordsNormal.z = sqrt(1 - pow(localCoordsNormal.x,2) - pow(localCoordsNormal.y,2)); //

    float3x3 local2WorldNormalMatrix = float3x3(i.tangentWorld,i.binormalWorld,i.normalWorld);
    float3 normalDirBump = normalize(mul(localCoordsNormal, local2WorldNormalMatrix));

    float3 finalBump = UnpackNormal(encodedNormalWater);

// EDGE BLENDING
    float depth = tex2Dproj(_CameraDepthTexture, i.reflUV).r;
    depth = LinearEyeDepth(depth);
    float eyeZ = i.pos.w;
    float distance = abs(eyeZ - depth);
    float edgeBlend = saturate(_Fade*distance);
```

```
217             // REFLECTION
218                 float4 uv1 = i.refluv;
219                 uv1.xy += finalBump.xy * _BumpAmplitude;
220                 float4 refl = tex2Dproj(_ReflectionTex, UNITY_PROJ_COORD(uv1));
221                 //float4 refl = float4(1,0,0,1); //DEBUG
222 
223             // REFRACTION
224                 float4 uv2 = i.refrUV;
225                 uv2.xy += finalBump.xy * _BumpAmplitude;
226                 float4 refr = tex2Dproj(_GrabTexture, uv2);
227                 //float4 refr = float4(0,1,0,1); //DEBUG
228 
229             // FOAM
230                 float4 uvFoam = i.uv;
231                 uvFoam.xy +=finalBump.xy;
232                 float4 foam = tex2D(_Foam, uvFoam * _Foam_ST.xy + _Foam_ST.wz);
233                 foam.rgb *= (1-edgeBlend);
234 
235             // FRESNEL - Simplified
236                 float3 nDir = normalize(i.normalWorld);
237                 float3 vDir = normalize(_WorldSpaceCameraPos - i.posWorld.xyz);
238                 float fresnelFactor = dot(vDir,nDir)*_FresnelCorrection;
239                 float4 fresnelColor = lerp(refl, refr, fresnelFactor);
240                 fresnelColor += foam*_FoamAmplitude;
241                 fresnelColor.a = edgeBlend;
242 
243                 fresnelColor = lerp(fresnelColor, finalIce, finalIce.a);
244 
245             // FRESNEL - Schlick's approximation
246                 //float fresnelFactor1 = 0.02f + 0.98f*pow(1-(dot(nDir,vDir)),5);
247                 //float4 fresnelColor1 = lerp(refr, refl,fresnelFactor1);
248 
249             // LIGHT
250                 float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
251                 // attenuation for directional light = 1, so we bypass it
252 
253             // Ambient
254                 float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb * fresnelColor.rgb;
255 
256             // Diffuse
257                 float diffuseFactor = max(0,dot(nDir, lightDir));
258                 float diffuseFactorBump = max(0,dot(normalDirBump, lightDir));
259                 diffuseFactor = lerp(diffuseFactor,diffuseFactorBump,_BumpShadowAmplitude);     // add bump sl
260                 diffuseFactor = diffuseFactor * 0.5 + 0.5;
261                 float3 diffuseLighting = _LightColor0.rgb * fresnelColor.rgb * diffuseFactor;
262 
263             // Specular
264                 float specularBumpFactor = pow(max(0,dot(reflect(-lightDir, normalDirBump),vDir)),_Sharpness),
265                 float specularInterpolatedFactor = pow(max(0,dot(reflect(-lightDir, nDir),vDir)),_Shininess);
266                 float3 specularLighting = _LightColor0.rgb * _SpecColor.rgb * (fresnelColor.a) *
267                                           (lerp(specularBumpFactor,specularInterpolatedFactor,0.3));
268 
269             // Translucent Diffuse
270                 float translucentFactor = max(0,dot(lightDir, - nDir));
271                 float3 translucentDiffuseLighting = _LightColor0.rgb * _DiffuseTranslucentColor.rgb * translucentFactor,
272 
273                 float4 finalColor = float4( ambientLighting +
274                                             diffuseLighting +
275                                             specularLighting +
276                                             translucentDiffuseLighting,fresnelColor.a);
277 
278             // FOG
279                 float fogFactor = 1/exp(pow(abs(eyeZ)*_FogDensity,2));
280                 finalColor = lerp(_FogColor,finalColor,fogFactor);
281 
282             // Testing
283                 //float4 uvTest = i.uv;
284                 //uvTest.xy +=finalBump.xy* _BumpAmplitude;
285                 //float4 testColor = tex2Dproj(_Checkers,uvTest);
286 
287                 return finalColor;
288 
289             }
290 
291             ENDCG
```

11.2. Water reflection code

```
using System;
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
[RequireComponent(typeof(Renderer))]
public class ReflectiveWater : MonoBehaviour {

    public float clipPlaneOffset = 0.07f;
    public LayerMask reflLayers = -1;
    public RenderTexture reflTexture;
    Hashtable mCameras = new Hashtable();

    //  This is called when it's known that the object will be rendered by some

    /// This is a build-in function in Unity3D that...
    private bool reflectionIsRendering = false;
    public void OnWillRenderObject()
    {
        if (reflectionIsRendering)
        {
            return; // prevents repetitive reflection
        }
        // REFLECTION.....................................
        reflectionIsRendering = true;
        Camera cam = Camera.current;
        Camera reflectionCamera = CreateReflectionCamera(cam);
        Vector3 pos = transform.position;
        Vector3 normal = transform.up;
        CopyCameraProperties(cam, reflectionCamera);
        float d = -Vector3.Dot(normal, pos) - clipPlaneOffset;
        Vector4 reflectionPlane = new Vector4(normal.x, normal.y, normal.z, d);
        Matrix4x4 reflection = Matrix4x4.zero;
        CalculateReflectionMatrix(ref reflection, reflectionPlane);
        Vector3 oldpos = cam.transform.position;
        Vector3 newpos = reflection.MultiplyPoint(oldpos);
        reflectionCamera.worldToCameraMatrix = cam.worldToCameraMatrix * reflection;
        Vector4 clipPlaneBelow = CameraSpacePlane(reflectionCamera, pos, normal, 1.0f);
        reflectionCamera.projectionMatrix = cam.CalculateObliqueMatrix(clipPlaneBelow);
        reflectionCamera.cullingMask = ~(1 << 4) & reflLayers.value;
        reflectionCamera.targetTexture = reflTexture;
        GL.invertCulling = true;
        reflectionCamera.transform.position = newpos;
        Vector3 euler = cam.transform.eulerAngles;
        reflectionCamera.transform.eulerAngles = new Vector3(-euler.x, euler.y, euler.z);
        reflectionCamera.Render();
        reflectionCamera.transform.position = oldpos;
        GL.invertCulling = false;
        GetComponent<Renderer>().sharedMaterial.SetTexture("_ReflectionTex", reflTexture);
        reflectionIsRendering = false;
    }
```

```
    Camera CreateReflectionCamera(Camera main)
    {
        Camera reflCamera = mCameras[main] as Camera;
        if (!reflCamera)
        {
            GameObject go = new GameObject("Water Reflecion Camera", typeof(Camera), typeof(Skybox), typeof(FlareLayer));
            reflCamera = go.GetComponent<Camera>();
            reflCamera.enabled = false;
            reflCamera.transform.position = main.transform.position;
            reflCamera.transform.rotation = main.transform.rotation;
            go.hideFlags = HideFlags.HideAndDontSave;
            mCameras[main] = reflCamera;
        }
        return reflCamera;
    }

    void OnDisable()
    {
        foreach (DictionaryEntry kvp in mCameras)
        {
            DestroyImmediate(((Camera)kvp.Value).gameObject);
        }
        mCameras.Clear();
    }
    void CopyCameraProperties(Camera source, Camera dest)
    {
        if (dest == null)
        {
            return;
        }
        if (source.clearFlags == CameraClearFlags.Skybox)
        {
            Skybox mainSky = source.GetComponent<Skybox>();
            Skybox reflectedSky = dest.GetComponent<Skybox>();
            if (!mainSky || !mainSky.material)
            {
                reflectedSky.enabled = false;
            }
            else
            {
                reflectedSky.enabled = true;
                reflectedSky.material = mainSky.material;
            }
        }
        dest.clearFlags = source.clearFlags;
        dest.backgroundColor = source.backgroundColor;
        dest.farClipPlane = source.farClipPlane;
        dest.nearClipPlane = source.nearClipPlane;
        dest.orthographic = source.orthographic;
        dest.fieldOfView = source.fieldOfView;
        dest.aspect = source.aspect;
        dest.orthographicSize = source.orthographicSize;
    }
```

```
    static void CalculateReflectionMatrix(ref Matrix4x4 reflectionMat, Vector4 plane)
    {
        float a = plane[0];
        float b = plane[1];
        float c = plane[2];
        float d = plane[3];

        // row 0
        reflectionMat.m00 = (1 - 2 * a * a);
        reflectionMat.m01 = (0 - 2 * a * b);
        reflectionMat.m02 = (0 - 2 * a * c);
        reflectionMat.m03 = (0 - 2 * a * d);

        // row 1
        reflectionMat.m10 = (0 - 2 * b * a);
        reflectionMat.m11 = (1 - 2 * b * b);
        reflectionMat.m12 = (0 - 2 * b * c);
        reflectionMat.m13 = (0 - 2 * b * d);

        // row 2
        reflectionMat.m20 = (0 - 2 * c * a);
        reflectionMat.m21 = (0 - 2 * c * b);
        reflectionMat.m22 = (1 - 2 * c * c);
        reflectionMat.m23 = (0 - 2 * c * d);

        // row 3
        reflectionMat.m30 = 0F;
        reflectionMat.m31 = 0F;
        reflectionMat.m32 = 0F;
        reflectionMat.m33 = 1F;
    }

    /// <summary>
    /// Given position/normal of the plane, calculates plane in camera space.
    /// </summary>
    Vector4 CameraSpacePlane(Camera cam, Vector3 pos, Vector3 normal, float sideSign)
    {
        Vector3 offsetPos = pos + normal * clipPlaneOffset;
        Matrix4x4 currentViewMatrix = cam.worldToCameraMatrix;
        Vector3 clipPlanePos = currentViewMatrix.MultiplyPoint(offsetPos);
        Vector3 clipPlaneNormal = currentViewMatrix.MultiplyVector(normal).normalized * sideSign;
        return new Vector4(clipPlaneNormal.x, clipPlaneNormal.y, clipPlaneNormal.z, -Vector3.Dot(clipPlanePos, clipPlaneNormal))
    }
}
```

11.3. Water physics code

```
    private List<Vector3> CubeApproximiation()
    {
        int totalNumberCubes = slicesPerAxis ^ 3;
        var cubeCenters = new List<Vector3>(totalNumberCubes);

        var bounds = myCollider.bounds;
        for (int iterationsX = 0; iterationsX < slicesPerAxis; iterationsX++)
        {
            for (int iterationsY = 0; iterationsY < slicesPerAxis; iterationsY++)
            {
                for (int iterationsZ = 0; iterationsZ < slicesPerAxis; iterationsZ++)
                {
                    float x = bounds.min.x + (bounds.size.x / slicesPerAxis) * (0.5f + iterationsX);
                    float y = bounds.min.y + (bounds.size.y / slicesPerAxis) * (0.5f + iterationsY);
                    float z = bounds.min.z + (bounds.size.z / slicesPerAxis) * (0.5f + iterationsZ);
                    var p = transform.InverseTransformPoint(new Vector3(x, y, z));

                    cubeCenters.Add(p);
                }
            }
        }
        return cubeCenters;
    }

```

```
    private void FixedUpdate()
    {
        // Buoyancy Force
        foreach (var point in voxels)
        {
            var wp = transform.TransformPoint(point);
            float waterLevel = ZDisplacement.ZPosition;

            if (wp.y - voxelHalfHeight < waterLevel)
            {
                float k = (waterLevel - wp.y) / (2 * voxelHalfHeight) + 0.5f;
                if (k > 1)
                {
                    k = 1f;
                }
                else if (k < 0)
                {
                    k = 0f;
                }

                var velocity = myRigidBody.GetPointVelocity(wp);
                var localDampingForce = -velocity * DAMPFER * myRigidBody.mass;
                var fullBuoyancy = localDampingForce + Mathf.Sqrt(k) * localBuoyancy0;
                myRigidBody.AddForceAtPosition(fullBuoyancy, wp);
            }
        }

        if(StreamForce == true)
        {
            // Stream force
            RaycastHit hit;
            if(Physics.Raycast(transform.position, Vector3.down, out hit, 1000f, myMask))
            {
                Debug.DrawRay(transform.position,Vector3.down,Color.red);
                Texture2D hitTex = hit.transform.gameObject.GetComponent<Renderer>().material.mainTexture as Texture2D;
                Vector2 pixelUV = hit.textureCoord;
                pixelUV.x *= hitTex.width;
                pixelUV.y *= hitTex.height;

                Color pixelColor = hitTex.GetPixel((int)pixelUV.x, (int)pixelUV.y);

                flowDir = new Vector3(pixelColor.r, 0, pixelColor.g);
                flowDir = flowDir*2;
                flowDir -= new Vector3(1,0,1);
                flowDir.z *= -1;
            }
            myRigidBody.AddForce(flowDir * WATER_FORCE * Time.deltaTime);
        }
    }
    Get values for info
}
```

11.4. Water Interaction code

```
    void Start()
    {
        waterYPos = ZDisplacement.ZPosition;
        interactionBlob = transform.GetChild(0);

        // min/max boundaries with reference to 0,0,0 center (local space)
        objectHeight = GetComponent<Renderer>().bounds.size.y;
        min = -objectHeight/2;
        max = objectHeight/2;

        // interaction blob
        startOffset = transform.position - interactionBlob.position;
        startRot = interactionBlob.rotation;

        // sphere
        shpereStartPos = transform.position;
        sphereStartRot = transform.rotation;
    }

    void Update()
    {
        waterYPos = ZDisplacement.ZPosition; // y is up (z-displacement)
        float dist = transform.position.y - waterYPos;
        if(dist >= min && dist <= max)
        {
            //Debug.Log("Inside range");
            interactionBlob.gameObject.layer = 8; // Interaction layer
        }
        else
        {
            //Debug.Log("Outside range");
            interactionBlob.gameObject.layer = 9; // Hidden layer
        }
        interactionBlob.position = transform.position - startOffset; // prevent movement of the blob
        interactionBlob.rotation = startRot;                         // prevent rotation of the blob
    }

    public void ResetPos()
    {
        transform.position = shpereStartPos;
        transform.rotation = sphereStartRot;
    }

    void OnTriggerEnter(Collider col)
    {
        if(col.gameObject.name.ToString() == "RiverEnd")
        {
            ResetPos();
        }
    }
}
```

11.5. Collected performance data

- **Turning on/off individual features**

Inclusion	Iteration					
	1	2	3	4	5	6
All FX ON						
1st run RTC (µs)	5.624	5.496	5.601	5.543	5.491	5.713
2nd run RTC (µs)	5.818	5.671	5.571	5.519	5.708	5.674
No REFL						
1st run RTC (µs)	5.533	5.447	5.447	5.544	5.615	5.503
2nd run RTC (µs)	5.412	5.53	5.369	5.616	5.359	5.414
No REFR						
1st run RTC (µs)	4.905	4.639	4.724	4.703	4.702	4.693
2nd run RTC (µs)	4.519	4.654	4.727	4.727	4.671	4.798
No Fresnel						
1st run RTC (µs)	3.393	3.313	3.403	3.382	3.391	3.618
2nd run RTC (µs)	3.39	3.718	3.417	3.399	3.316	3.33
No Flowmap						
1st run RTC (µs)	4.312	4.352	4.302	4.319	4.298	4.391
2nd run RTC (µs)	4.382	4.456	4.32	4.226	4.341	4.25
No Light						
1st run RTC (µs)	4.952	4.852	4.936	4.858	4.91	4.877
2nd run RTC (µs)	4.857	4.806	4.784	4.832	4.864	4.875
No Fog						
1st run RTC (µs)	5.514	5.588	5.58	5.486	5.493	5.588
2nd run RTC (µs)	5.555	5.56	5.548	5.643	5.708	5.596
No Ice						
1st run RTC (µs)	2.723	2.812	2.796	2.814	2.779	2.73
2nd run RTC (µs)	2.752	2.757	2.768	2.772	2.813	2.742
No Z-move						
1st run RTC (µs)	5.604	5.502	5.604	5.513	5.782	5.606
2nd run RTC (µs)	5.58	5.59	5.697	5.566	5.544	5.584
NO Edge blend						
1st run RTC (µs)	5.346	5.389	5.504	5.303	5.342	5.308
2nd run RTC (µs)	5.332	5.169	5.335	5.279	5.263	5.328

- **Changing grid-size of the water surface**

Grid Size							average	
10x10 (Unity3D)	7.195	7.067	7.172	7.114	7.062	7.284	7.190	1.571
	7.389	7.242	7.142	7.09	7.279	7.245		
16x16	9.937	9.930	10.170	9.951	9.936	10.127	10.077	4.603
	10.515	10.767	10.754	9.613	9.514	9.705		
32x32	9.945	9.922	10.425	10.224	10.738	10.392	10.197	4.756
	10.063	10.043	9.968	10.117	10.169	10.353		
64x64	11.377	10.876	10.975	10.745	10.968	10.960	11.035	4.959
	11.376	11.289	10.830	11.352	10.859	10.816		
128x128	18.700	18.559	18.387	18.427	18.709	18.241	18.652	7.436
	18.721	18.905	18.822	18.983	18.381	18.990		

- **T-tests for validating information**

ALL FX ON - no signif diff		
	5.624	*5.818*
Mean	5.569	5.629
Variance	0.008	0.006
Observations	5	5
Pearson Correlation	-0.059	
Hypothesized Mean Difference	0	
df	4	
t Stat	-1.067	
P(T<=t) one-tail	0.173	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.346	
t Critical two-tail	2.776	

NO REFLECTION - no signif diff		
	5.533	*5.412*
Mean	5.511	5.458
Variance	0.005	0.012
Observations	5	5
Pearson Correlation	-0.115	
Hypothesized Mean Difference	0	
df	4	
t Stat	0.862	
P(T<=t) one-tail	0.219	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.437	
t Critical two-tail	2.776	

NO REFRACTION - no signif diff		
	4.905	*4.519*
Mean	4.692	4.715
Variance	0.001	0.003
Observations	5	5
Pearson Correlation	0.470	
Hypothesized Mean Difference	0	
df	4	
t Stat	-1.032	
P(T<=t) one-tail	0.180	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.360	
t Critical two-tail	2.776	

Fresnel - no signif diff		
	3.393	*3.39*
Mean	3.421	3.436
Variance	0.013	0.027
Observations	5	5
Pearson Correlation	-0.609	
Hypothesized Mean Difference	0	
df	4	
t Stat	-0.130	
P(T<=t) one-tail	0.451	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.903	
t Critical two-tail	2.776	

Flow map - no signif diff		
	4.312	*4.382*
Mean	4.332	4.319
Variance	0.002	0.008
Observations	5	5
Pearson Correlation	-0.064	
Hypothesized Mean Difference	0	
df	4	
t Stat	0.306	
P(T<=t) one-tail	0.387	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.775	
t Critical two-tail	2.776	

Light - no signif diff		
	4.952	*4.857*
Mean	4.887	4.832
Variance	0.001	0.001
Observations	5	5
Pearson Correlation	-0.208	
Hypothesized Mean Difference	0	
df	4	
t Stat	2.116	
P(T<=t) one-tail	0.051	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.102	
t Critical two-tail	2.776	

Fog - no signif diff		
	5.514	*5.555*
Mean	5.547	5.611
Variance	0.003	0.004
Observations	5	5
Pearson Correlation	-0.866	
Hypothesized Mean Difference	0	
df	4	
t Stat	-1.252	
P(T<=t) one-tail	0.139	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.279	
t Critical two-tail	2.776	

Ice - no signif diff		
	2.723	*2.752*
Mean	2.786	2.77
Variance	0.001	7E-04
Observations	5	5
Pearson Correlation	0.264	
Hypothesized Mean Difference	0	
df	4	
t Stat	0.942	
P(T<=t) one-tail	0.2	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.399	
t Critical two-tail	2.776	

Z-displacement - no signif diff		
	5.604	*5.58*
Mean	5.601	5.596
Variance	0.013	0.003
Observations	5	5
Pearson Correlation	-0.224	
Hypothesized Mean Difference	0	
df	4	
t Stat	0.084	
P(T<=t) one-tail	0.468	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.937	
t Critical two-tail	2.776	

Edge blend - no signif diff		
	5.346	*5.332*
Mean	5.369	5.275
Variance	0.007	0.004
Observations	5	5
Pearson Correlation	0.127	
Hypothesized Mean Difference	0	
df	4	
t Stat	2.121	
P(T<=t) one-tail	0.051	
t Critical one-tail	2.132	
P(T<=t) two-tail	0.101	
t Critical two-tail	2.776	

- **Measuring separate renderings**

	Rendering		
Sphere num.	water	spheres	reflection
1	4.597	2.825	1.379
	4.606	2.811	1.353
	4.568	3.074	1.499
	4.666	2.823	1.351
	4.769	2.922	1.338
	4.768	2.803	1.397
average	4.662	2.876	1.386
3	4.35	2.933	1.314
	4.328	2.771	1.322
	4.393	2.934	1.354
	4.367	2.787	1.294
	4.367	2.778	1.378
	4.504	2.935	1.271
average	4.385	2.856	1.322
10	4.347	2.923	1.357
	4.231	2.909	1.318
	4.174	2.81	1.346
	4.266	2.873	1.331
	4.2	2.871	1.324
	4.348	3.081	1.374
average	4.261	2.911	1.342
50	4.306	3.505	1.639
	4.505	4.252	1.493
	4.106	3.546	1.503
	4.286	3.619	1.491
	4.227	4.042	1.267
	4.481	3.538	1.37
average	4.319	3.750	1.461