KAUNAS UNIVERSITY OF TECHNOLOGY

Department of Multimedia Engineering

Master Thesis

3D visualization of sailing regatta in real time

Student: Benito Ponappan, SunithaAnilkumar
(Group, Name, Surname, Signature)

Supervisor: *assoc.prof. Armantas Ostreika*
(Position, Degree, Name, Surname, Signature)

Business consultant: *Ričardas Novošinskas*
(Position, Degree, Name, Surname, Signature)

Reviewer: *Lekt. Dr. Šarūnas Packevičius*
(Position, Degree, Name, Surname, Signature)

Kaunas, 2015

Table of Contents

1 Introduction 6
2 Analysis of existing approaches 7
2.1 Literature review 7
2.2 GIS data processing 8
2.3 3D Wave Generation 8
2.4 Modelling and Rendering Ocean Waves 10
2.5 Comparison with Existing Ocean models 11
2.6 Terrain generation 12
2.7 Level of Detail 13
2.8 Related Work 15
3 Methodology 16
3.1 Unity3D 16
3.2 Sail navigation 16
3.3 Ocean Models 18
3.4 Simulating & Animating Ocean Waves 23
3.5 Terrain generation 23
4 Implementation and Result 26
4.1 Boat Controller 26
4.2 Boat Information 28
4.3 Ocean Waves 29
4.4 Generating Wave 33
4.5 Generating Wake 34
4.6 Heightmap and Normal Maps 34
4.7 Creation of Water Shader 35
4.8 Landscape generation 38
4.9 Performance Test 43
5 Future Scope 51
5.1 Web Player Streaming 51
5.2 Tuning for Portals 51
5.3 Live Streaming 51
5.4 Publishing Streaming Web Players 53
6 Conclusion 54

List of Figures

Figure 2.1 Procedural terrain generation methods 12
Figure 3.1 Sail regatta course with markings to waypoints and routes 16
Figure 3.2 Crossing points of start and finish line 17
Figure 3.3 Sailboat heading to mark for rounding 18
Figure 3.4 Frequency curvature spectra 22
Figure 3.5 Arrangement of terrain tiles around a point of interest 24
Figure 3.6 Re-arrangement of terrain tiles system 25
Figure 4.1 Architecture overview of the 3D visualization system 26
Figure 4.2 Block diagram of the sail system 27
Figure 4.3 Boats position from routeline 29
Figure 4.4 Class Diagram of Ocean Simulator 30
Figure 4.5 Sequence Diagram of setup stage 31
Figure 4.6 Main Loop 32
Figure 4.7 Size of different IFFT domains. 33
Figure 4.8 Wake created when boat is moving 34
Figure 4.10 Height maps 34
Figure 4.9 Normal Maps 34
Figure 4.11 Elevated mesh 35
Figure 4.12 Gradient Texture that controls front wave and foam intensity 38
Figure 4.13 Static terrain representing a shoreline 39
Figure 4.14 Terrain generated using TIFF files 40
Figure 4.15 Terrain generated from RAW image 41
Figure 4.16 Terrain generated from RAW image and textured using splatting 41
Figure 4.17 (a),(b) and (c) are Terrains generated after processing data received from Goole services and viewed at various zoom levels 43
Figure 4.18 FPS of Current ocean model and Philips spectra in 8 * 8 Grid 44
Figure 4.19 FPS of Current ocean model and Philips spectra in 16 * 16 Grid 44
Figure 4.20 FPS of Current ocean model and Philips spectra in 32 * 32 Grid 45
Figure 4.21 FPS of Current ocean model and Philips spectra in 64 * 64 Grid 45
Figure 4.22 FPS of Current ocean model and Philips spectra in 128 * 128 Grid 45
Figure 4.23 FPS of Current ocean model and Philips spectra in 256 * 256 Grid 46
Figure 4.24 FPS of Current ocean model and Philips spectra in 512 * 512 Grid 46
Figure 4.25 Comparison of load time using the implemented terrain generation techniques 47
Figure 4.26 Time comparison of dataset types for generation of 9 terrains 48
Figure 4.27 Time comparison of dataset types for generation of 25 terrains 48
Figure 4.28 FPS from Core i3 processor 49
Figure 4.29 FPS from AMD A10 Processor 49
Figure 4.30 FPS from Core i5 processor 49
Figure 4.31 FPS from Core i7 processor 50
Figure 5.1 Live streaming Process 52

List of Tables

Table 3-1 Relations between oceanographic terms. The subscript ∞ indicates a deep water term....... 19
Table 4-1 Physical size of domains.. 33
Table 4-2 FPS comparison with Current ocean model and Philips spectra 44
Table 4-3 FPS comparison with Different System... 48

ABBREVIATIONS AND NOTATIONS

GPS	Global Positioning System - a space-based satellite navigation system that provides location and time information in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.
LOD	Level Of Detail - In computer graphics, accounting for level of detail involves decreasing the complexity of a 3D object representation as it moves away from the viewer
2D	Two-dimensional_space
3D	Three-dimensional_space
iOS	iPhone Operating System - is a mobile operating system developed by Apple Inc
CPU	Central processing unit
GPU	Graphics processing unit - a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display
BRDF	Bidirectional reflectance distribution function
FPS	Frame Rate per Second
GUI	Graphical User Interface
CFD	Computational Fluid Dynamics
IFFT	Inverse Fast Fourier Transform

1 Introduction

In Sail racing, while most sailors say that sailing is the most involved and exciting sport they participate in, it can be difficult to watch, especially for untrained spectators. This is because racing occurs half a mile from the shore and as sails drift away from shore, they become invisible and in-transparent from the water, making it hard for spectators to identify their friend or teammates. Even when races take place close to land, the view that spectators have is highly skewed due to the perspective, making very difficult to identify what boat is leading and which are trailing. The experience of participants is limited to real-time subjective information and makes it unpredictable to analyze the race and compare with competitors. Spectators, Coaches and Media have a hard time to understand the competition and it is impossible to visualize key moments.

Using the Global Positioning System (GPS), which are installed in the sails and buoyant (marks), will record the two numbers that pin-point the precision of unlocking a huge potential in global tracking – latitude and longitude, are transmitted to the shore‘s base station. The information is processed and graphically presented along with the current environmental conditions on the sea to the viewers online. The system will store the tracks of individual boats participating in sailing regattas; users will be able to review these tracks at their convenience for post race analysis and other educational purposes. The intention of this project is to develop a complete dynamic and competitive 3D Visualization system for sailing regattas in real time and lay emphasis on graphical effects. In order to attain this realistic view, a system is designed by taking into account the physics of sailing and reciprocates an output and thereby overlay it with graphical effect providing a visually appealing experience for viewers globally.

The Goal of this project is to develop an online 3D viewing system for sail racing events.

The main objectives of the project are

1. Track sail races using GPS data and efficiently process the data for use within the 3D game engine.
2. Render a real-time ocean water surface for the sail-boats to sail.
3. Dynamically integrate and test real-time shorelines using geospatial data using different data types-TIFF and RAW.
4. Create a prototype for live video streaming in webplayer.

The real-time data can also be used by sailors to make tactical decisions during a race. For example, the system can alert dock staff when boats stray too close to shore, or when sails are drifting away from the course line. At the outcome of this project, we also aim to enhance our knowledge on application development and learn new aspects of graphic design.

2 Analysis of existing approaches

2.1 Literature review

Stan Melax in [4] discusses his method of achieving polygon reduction algorithm is to take a high detail model with many polygons and generate a version using fewer polygons that looks reasonably similar to the original. Having initially implemented real-time Boolean operations, it was noticed that the repeated use of Boolean operations performed on polygonal objects generated lots of additional triangles. Many of these additional faces were small or splinter triangles that didn't contribute to the visual quality of the game — which in turn just slowed it down. Most of the better techniques were variations of the progressive meshes; these techniques reduce a model's complexity by repeated use of the simple edge collapse operation. The trick Stan Melax used to produce good low-polygon models, was to select the edge that when collapsed, will cause the smallest visual change to the model and this was done at cost of collapsing an edge as the length of the edge multiplied by a curvature term.

This technique was good enough to generate the level of detail (LOD) models for the game engine. An improved version of this basic algorithm has since been incorporated into Bioware's 3D graphics engine, Omen. One problem with swapping models is that players often notice when this occurs (the phenomenon known as "popping"). Another technique for doing LODs in a game is to represent an object's geometry using parametric surface patches, which are tessellated on the fly to the desired detail.

An important component in many games is the LOD of models. Frank Losasso and HuguesHoppe[5] describe Geomerty clipmaps for Terrain rendering using Nested regular grids - which caches the terrain in a set of nested regular grids centered about the viewer. As the viewer approaches the surface, finer grid levels are produced than the stored terrain using fractal noise displacement. In this paper they mention texture clipmap which lead to geometry clipmaps, the former based on LOD per-pixel. Geometry clipmaps on the other hand use LOD in worldspace based on viewer distance, using a set of nested rectangular regions about the viewpoint and does not require special hardware. Each level contains an$n \times n$ array of vertices, stored as a vertex buffer in video memory. For each level of the clipmap, a set of rectangular regions called clip region is rendered and updated per frame depending on viewers motion. Each clipmap level also contains associated texture image(s) for use in rasterization but this will require more memory when letting the hardware control mipmapping. To overcome this, mipmapping is disabled altogether, and LOD is performed on the texture using the same spatial transition regions applied to the geometry i.e based on viewer distance rather than on screen-space derivatives as in hardware mipmapping. In order to optimize the rendering, view frustum culling was used to avoid rendering terrain that lies outside the viewport.

Filip Strugar[6]in his paper presents a technique for GPU-based rendering of heightmap terrains, which is a refinement of several existing methods which draws the terrain directly from the source heightmap data using a quadtree of regular grids and let the GPU to handle the creation of meshes instead of having the CPU do it. LOD layer selection is based on the actual three-dimensional distance from the observer. He introduces Continuous Distance-Dependent LOD, which organizes the heightmap into a quadtree, which is used to select appropriate quadtree nodes from different LOD levels at run time. The selection algorithm is performed in such a way as to provide approximately the same amount of on-screen triangle complexity regardless of the distance from the observer. The quadtree structure is generated from the input heightmap. It is of constant depth, predetermined by memory and granularity requirements. Once created, the quadtree does not change unless the source heightmap changes. Every node has four

child nodes and contains minimum and maximum height values for the rectangular heightmap area that it covers.

2.2 GIS data processing

Geospatial data or geographic information is the data or information that identifies the geographic location of features and boundaries on Earth, such as natural or constructed features, oceans, and more. Spatial data is usually stored as coordinates and topology, and is data that can be mapped. Spatial data is often accessed, manipulated or analyzed through Geographic Information Systems (GIS).

Data acquisition of 3D objects to build highly accurate models of real 3D objects are rapidly becoming more affordable. LIDAR or laser scanning technology determines the range to an objecct by measuring the time delay between transmission of a pulse and detection of the reflected signal and registers distrubted data spatially. This tchonology is highly reliable as it has the ability to penetrate the vertical profile of a forest canopy and quantify its structure. Knut Stolze[26], describes of the standard to store, retrieve and process spatial data in a relational database system where a set of types and methods are defined for reprsenation of multi-dimentional geographic features.

Alastair Aitchison[24], uses the approach of converting DEM images into useable format as ASCII text file and import them to SQL serevr through an excel template. Once the excel file is imported into SQL Server, a new table is created and the table is populated with a point in each row correspoinding to its geographic tile location. Inorder to create a terrain map, the database is queried for a subset of data corresponding to the map tile or quadkey (Bing maps tile). The data subset (which is a set of distinct elevation recordings, recorded on a grid spaced at regular intervals) is determined by cetrain parameters corresponding to the zoom level. Bing map tile system[25] describes the projection, coordinate systems, and addressing scheme of the map tiles.

Delauney Triangulation is applied to the set of distinct points to create a continuous smooth surface in WPF. The rendered terrain showed hard edges between triangles faces because many of the vertices in the mesh list were duplicates. To overcome the triangles from being rendered individually, the mesh list was updated to store only unique points and then use the Triangle Indices property to list the index positions of each vertex that forms a triangle. The aftereffect was a smooth looking terrain. The final render uses a map overlay as texture for the terrain.

Haala, Brenner and Anders [Hierarchical Storage and Visualization of Real-Time 3D Data] built a structure for the dynamic acquisition, insertion, and use of global geospatial data. Data acquisition can be of any geospatial form and follow an acquisition process which includes some pre-analysis steps that functions both for subsequent efficient data access and detail management and for improved understanding of the data thereafter apply a fast clustering algorithm to identify the overall acquisition pattern and make the acquired data available in a more efficient form. This method utilizes time as a parameter to tracks space and time patterns in data distributions. The approach is efficient in rendering terrain based on view dependence as data is organized in a hierarchical clustering format.

2.3 3D Wave Generation

There are a lot of approaches available for 3D wave generation; basically three approaches are taken into account for creating 3D water waves. Sum-of-sines, Computation Fluid Dynamics (CFD) and Inverse Fast-Fourier Transform (IFFT).

2.3.1 Sum-of-Sines

The sum-of-sines approach, where some number of waves are added together and used to displace a water mesh. But, computing individual waves like this and adding them all together each frame can be expensive and burn up GPU resources, so it end up being very limited with the number of waves you can simulate at once. Not having enough waves will result in a lack of detail on the water – Need thousands of them before it looks like a real ocean. This problem can be mitigated somewhat through the use of normal and displacement maps to add high-frequency detail over the lower-frequency procedural waves.

2.3.2 Computational Fluid Dynamics

Computational Fluid Dynamics is the branch of fluid dynamics providing a cost effective means of simulating real flows by numerical method of governing equations. The governing equations for Newtonian fluid dynamics, namely the Navier-Stokes equations, have been known for over 150 years. However the development of reduced forms of these equations is still an active area of research, in particular, the turbulent closure problem of the Reynolds-averaged Navier-Stokes equations. For non-Newtonian fluid dynamics, chemically reacting flows and two phase flows, the theoretical development is at less advanced stage.

Experimental methods has played an important role in validating and exploring the limits of the various approximations to the governing equations, particularly wind tunnel and rig tests that provide a cost-effective alternative to full-scale testing. The flow governing equations are extremely complicated such that analytic solutions cannot be obtained for most practical applications.

Computational techniques replace the governing partial differential equations with system of algebraic equations that are much easier to solve using computers. The steady improvement in computing power, since the 1950's, thus has led to emergence of CFD. This branch of fluid dynamics complements experimental and theoretical fluid dynamics by providing alternative potentially cheaper means of testing fluid flow systems. It also can allow for the testing of conditions which are not possible of extremely difficult to measure experimentally and are not amenable to analytic solutions [9].

This method is Real-time since this approach simulate pretty much any intersection water can have with its environment, it is complex and generally limited to very small areas for real-time usage on today's hardware and research improves. Hybrid approaches that blend CFD for effects near the camera with other larger- scale techniques could prove interesting.

2.3.3 Inverse Fast Fourier Transform

The best approach today is using a mathematical technique called an inverse Fast-Fourier Transform(IFFT).Basically pre-compute a 2D array of individual waves, defined by their amplitudes, directions, and wavelengths. Each frame, the FFT transform is applied to this array to convert it into animated 3D wave displacements. An array of 256x256 waves will result in 65,536 individual waves being simulated at once. FFT's may be accelerated using GPGPU technologies such as CUDA, OpenCL, or DirectX11 compute shaders. Waves created in this manner can cost less than one millisecond per frame - but this depends a lot on proper implementation. Need to displace the final waves to make them look pointy and "choppy", and this involves further FFT computations. Another complication is that fast FFT transforms may not be available on every platform user plan to support, or different platforms may require different techniques. Can assume DirectX11 graphics hardware, DX11 has very fast FFT

functions built into it. But to target DX9-level hardware, or platforms such as iOS, user need to implement and support multiple approaches to generating these waves depending on the platform they are running on. However still found FFT to be the best tradeoff between performance and visual quality available today, and Rendering infinite oceans in 1-4 milliseconds with this technique. IFFT is explained in detail under section 4.3.3.

2.4 Modelling and Rendering Ocean Waves

A simple model for the surface of the ocean, suitable for the modelling and rendering of most common waves where the disturbing force is from the wind and the restoring force from gravity. It is based on the Gerstner, or Rankine, model where particles of water describe circular or elliptical stationary orbits. The model can easily produce realistic wave's shapes which are varied according to the parameters of the orbits [10]. The surface of the ocean floor affects the refraction and the breaking of waves on the shore; the model can also determine the position, direction, and speed of breakers. The ocean surface is modelled as a parametric surface, permitting the use of traditional rendering methods, including ray-tracing and adaptive subdivision. Animation is easy, since time is built into the model. The foam generated by the breakers is modelled by particle systems whose direction, speed and life expectancy is given by the surface model. To give designers control over the shape of the ocean, the model of the overall surface includes multiple waves, each with its own set of parameters and optional stochastic elements. The overall "randomness" and "short crestedness" of the ocean is achieved by a combination of small variations within a train and large variations between trains.

2.4.1 Real time synthesis and Rendering Ocean water

A multi-band Fourier domain approach used for synthesizing and rendering deep-water ocean waves entirely on the graphics processor. The technique begins by using graphics hardware to generate and animate a Fourier domain spectrum of ocean water. Subsequently use the graphics hardware to apply an IFFT to transform the spectrum into a realistic height map of ocean water in the spatial domain. As a result, this technique can be used to efficiently synthesize and render height maps and normal maps which are spatially periodic. GPU-synthesized low frequency height maps are used to displace geometry while broad spectrum GPU-synthesized waveforms are used to generate a normal map for shading. The model also allows for compositing of other waveforms such as wakes or eddies caused by objects interacting with the water [11]. The importance is the GPU to perform all synthesis and rendering steps as well as the multi-band approach which enables efficient simulation of the natural filtering of high frequencies at different points on the water surface due to depth variation or the presence of plant matter.

2.4.2 Second Order Spectral Simulation of Directional Wave Generation and Propagation

The second order spectral simulation of directional wave generation and propagation is based on time accuracy. The flow is described in the framework of potential theory, with second order Stokes expansion of nonlinear boundary conditions on the free surface and on the wave maker. The resulting initial boundary value problem is solved using a recently developed spectral formulation. The basic principles of the numerical scheme are first presented. Then, some illustrative results on the generation of oblique waves in the new ECN offshore wave tank are given, comparing the standard snake's principle, and Dalrymple's method [12].The objective of this method is to investigate both in time and in space the

usable test area in our multi-directional wave tank, for required sea states. The drawback of this method is the damping conditions applied at far end of the tank

2.5 Comparison with Existing Ocean models

2.5.1 Multi resolution reflectance model

Multi resolution reflectance models are difficult to design in the general case. In this case the dynamic surface complicates the problem because it forbids pre-computations. The geometry and spectrum of the surface is easy to filter, by removing frequencies above the Nyquist limit. In addition, the surface of the ocean has Gaussian statistic properties, at almost all scales. This is the starting hypothesis of many BRDF models. This section presents method to transition from geometry to normals and then to shading based on statistical surface properties, recalling some physical facts about deep water waves.For the BRDF the variance of the wave slopes needs to be pre-computed and stored in a 3d texture. This only needs to be done once on start up. To render into a 3d texture in Unity it's advisable to use a compute shader. Previous experiments with this method lead to Crash the graphics card. The problem is the calculations require a rather large loop in the compute shader and it causes the crash. But by reducing number of threads, crashes can be avoided. Consider moving the calculations to the CPU but it could be rather slow to compute [13].

2.5.2 Small and large wave Replication model.

In case of Model which is designed to replicate ocean waves on both small and large waves, Phillips spectrum is considered. It has quite a different effect than the previous model and in some ways looks much more 'watery'. Jerry Tessendorf released a paper called 'Simulating Ocean Water' [14]. In this paper the math for using Phillips spectrum was outlined and Keith Lantzconverted this into code [18]. The Code was written in C++ so it was just a matter of converting that to a C# script for Unity. The issue is class of complex numbers with operator overloading. This is fine in C++ but in a C# script operator overloading does not perform as well as in C++. By performing the complex number math by just using floats and storing the results in Unity's Vector2 class, the frame rate from 10fps to 60fps. The slope and displacement both consist of two complex numbers which could be stored in array of Vector4's and have the Fourier transformed applied at the same time instead of separately. The frame rate changes from 60fps to 100 fps. The Fourier transform is calculated on the CPU and in some of the previous ocean projects, the Fourier transform is calculated on the GPU.

This method uses the wave heights to directly modify a meshes vertices position. User can interact directly with the mesh for physics effects. The mesh created is tileable and have added the ability to tile as many meshes. There are some problems with this approach however. For a start since the results are stored in a mesh instead of a texture there is no way to remove the repeating pattern produce from tiling multiple meshes. Secondly since larger scenes are made from tiling the same mesh there is no natural way to create a LOD scheme which would be essential for large scenes. And lastly since the normals are only per vertex they do not contain enough detail for good lighting calculations.

2.6 Terrain generation

A broad range of applications use artificial terrain generation approaches. Few of the applications where terrain generation is common are 3D architecture, VR applications, computer games and real-time rendering 3D engines. Terrain generation can be categorized by their various approaches such as generation of terrains using measuring techniques, modelling approaches and procedural terrain generation (PTG) algorithms.

The measuring approach uses a dataset containing real-world information, which includes elevation data, geographic position...Etc. This dataset can be used to create 3D representation of a terrain's surface and are referred as digital elevation model (DEM). These DEM‘s can be developed in a number of ways but they are commonly acquired through satellite remote sensing or land survey data. The terrain generated using DEM produce highly realistic looking models with less intervention in the process of generation. However these are more or less static datasets and cannot be manipulated easily but at the expense of time, we can have them transformed to achieve a terrain with added features or produce a terrain with the specific expectations of a designer.

Modelling is a method where terrains are sculpted using 3D modelling tools such as Blender, 3DS Max, World Machine, Grome etc. the design pipeline varies as per the tool selection. This process helps deliver a model which is an exact replica of designers thought to evolve terrains accordingly to their aesthetic feelings. While the final outcome is considered excellent as per design requirements, this technique is not as time efficient compared to its peers, as the process depends on the skill and expertise of a designer.

PTG is achieved programmatically rather than manually. Inside this category comes a variety of sub categories based on their generation techniques. Representations of these fields are shown in Figure 2.1.

PTG use voxels, which represent a value on a regular grid in three-dimensional space. Voxels are useful for storing 3D data; they are also ideal candidates for the application of procedural noise shaping functions. Voxel datasets can be quite memory hungry since they define forms as density clouds which record data for empty space as much as for solid objects, however they are very flexible and structurally formal.[23]

Generation of terrain using mesh is traditional in PTG. The mesh is used to create a topology of the terrain using an input function such as height, thereby laying out vertices according to the corresponding height values. When creating large terrains, we can have a single mesh with a uniform resolution but this would require large memory buffer. Using a dynamic quad tree [17], it would be better to draw a mesh with more details near a point of interest and decrease the mesh resolution around areas away from the point of interest to reduce poly-count or rendering load.

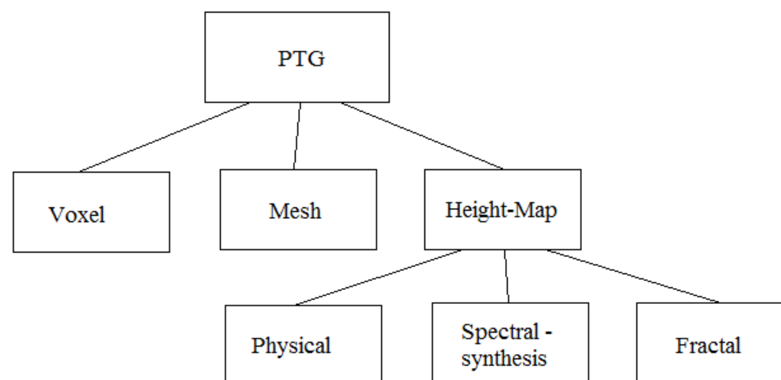


Figure 2.1 Procedural terrain generation methods

Since a large number of virtual environments, games are built around terrains or maps, a heightmap specifies elevations of various parts of the maps. In particular, many strategy games are heavily dependent on maps and the character of the map can strongly influence the game play. Height-map is a raster image used to store values, such as surface elevation data. Most terrains generation methodologies translate data stored in a heightmap into 3D structures. Heightmaps can be generated using a variety of approaches, and some of the commonly used approaches are physical, fractal and spectral synthesis.

Olsen J [19], presents a terrain evolution algorithm for physically based terrains which cater to real-time situations like thermal, hydraulic erosion using a set of desirable traitsto measure erosion rate for a height map. He also touches on the pink noise, which is characterized by the spectral energy density being proportional to the reciprocal of the frequency

$$P(f) = 1/ f_a \qquad (2.6.1)$$

Where P(f) is the power function of the frequency and a is close to 1.This kind of noise approximates real-world uneroded mountainous terrain well.

Spectral synthesis simulates 1/f noise by adding several octaves (layers) together, each octave consisting of noise with all its spectral energy concentrated on a single frequency. For each octave, the noise frequency is doubled and the amplitude A is calculated by [19].

$$A = p_i \qquad (2.6.2)$$

Where *i* is the octave number starting with 0 at the lowest frequency and p is called the persistence. Using the inverse Fast Fourier Transform (FFT), the frequencies can be converted to altitudes.

The fractional Brownian motion(fBm), developed by Mandelbrot was implemented by A. Fournier[20] to compute the surface to arbitrary levels of details without increasing the database, thereby allowing objects with complex appearances to be displayed from a very small database. Furthermore, with the modification of few parameters, the character of the terrains surface could be controlled. Using the fractal approach, the resolution of a terrain is recursively increased and the height of each new point on the terrain is adjusted to provide a blended randomness. A fractal generation algorithm generates unique terrain at each execution, thereby making it interesting for strategy genre layout.

Ashlock et al. [21] proposed that a midpoint L-system or Linden-Mayer system can be effective at generating fractal style terrains using the principles of midpoint displacement fractals. The authors uses an evolutionary algorithm (EA) to generate the right set of rules to render them into the desired type of object and use this approach to evolve fractal landscapes to fit specific shapes.

2.7 Level of Detail

In computer graphics, geometric datasets can be too complex to render at interactive rates, therefore the solution is to simplify the polygonal geometry of small and distant objects. This theory is besides being known as Level of Detail (LOD), also known as polygonal simplification, geometric simplification, mesh reduction, decimation, and multi-resolution modeling.

Luebke et al. (2003) [7] state that LOD is used to improve the performance and quality of three-dimensional (3D) visualization in computer graphics. It follows a simple fundamental rationale: when 3D scene is rendered, it is optically sufficient and computationally efficient to use a less detailed representation for small, distant, or unimportant portions of the scene. Their definition is that LOD is “the real-time 3D computer graphics technique in which a complex object is represented at various resolutions and the most appropriate representation chosen in real time in order to create a trade-off between image fidelity and frame rate. This term is often used interchangeably to refer to both the graphics technique and a single representation of an object.”

LOD can be categorized as follows: discrete LOD, continuous LOD, and view-dependent LOD.

The discrete LOD move is conventional: a fixed number of LODs are created for each object separately and preprocessed, and at game play, LOD is determined by its distance or size for point of view. They are called discrete LOD since the LODs are pre-determined at set distances. This approach has the advantages that it is simple – decouples simplification and rendering, and it does not have to address real-time rendering constraints. Run-time rendering needs only to choose The LOD for an object is only chosen during runtime. However, the drawback to this approach is its adaptation to extreme details where huge objects need to be subdivided and combined with small object (e. g. massive Computer-Aided Design (CAD) models) due to the popping effect which can occur when the graphics system switches between different levels of detail.

The continuous LOD approach provides an advantage over the discrete LOD that creates a data structure from which a desired level of detail can be extracted at run time. It provides better fidelity (LOD is specified exactly, not chosen from a few pre-created options), and smoother transitions (continuous LOD can adjust detail gradually and incrementally, reducing visual pops).

In the end, the continuous LOD leads to view-dependent LOD which uses current view parameters to select best representation for the current view. Single objects may thus span several levels of detail. For instance, nearby portions of an object can be shown at higher resolution than distant portions. Such approach provides even better granularity (allocates polygons where they are most needed, within as well as among objects), and enables drastic simplification of very large objects.

2.8 Related Work

2.8.1 GeoRacing

GeoRacing, is a product developed by R&D team of TRIMARAN, a Broadcast Graphics Specialist for Sport on TV. They provide GPS tracking with Timing, Ranking and 2D & 3D Views for various outdoor sport events which include sailing as well. During SVG Europe's interview with Trimaran CEO Olivier Emery about the development and key feature-set of the system [1], Olivier Emery stated the innovation solution includes individual trackers (e.g., GPRS, SAT, and RF), a cloud platform to manage live data and to serve customers with Mac and PC-based 2D players, apps for iOS and Android-based devices, and a 3D rendering engine for broadcast and webcast.

2.8.2 Virtual Eye Sailing

The Virtual Eye sailing system [2] (developed since 1992) has been another solution for sailing events worldwide. Virtual Eye, often only associated with the Americas Cup and the Volvo Ocean race, has evolved into an easily deployable system at any event. Some of these include the Audi MedCup and world tour events like the Monsoon Cup, the Korea Cup and the Louis Vuitton Trophy Series. Virtual Eye can show an entire race course, including marks, laylines, advantage lines and distances between the boats. Virtual Eye also displays timing information from starts, mark rounding and finishes. All of this information is available in real-time for immediate review and post-race analysis. They rely on Igtimi.Ltd for GPS and data network to develop an accurate and reliable system. Tracking data is sent at a rate of 1-10 times a second, and the tracking system is capable of transmitting this data over a dual network. Eventually this data is used to create the necessary graphics with higher quality.

2.8.3 RaceQs

RaceQs is a sailing system designed to improve sailboat racing performance. The system includes a free smart phone app that functions as a race computer and GPS tracker. The phone app records sailing data to produce 3D animated replays for post-race analysis. After sailing, the phone app automatically uploads race data to raceQs.com to create a 3D animated replay of the entire race. The shareable 3D replays are viewed on raceQs sailing social network [3]. This is one other application to view races online and monitor performance each time.

3 Methodology

3.1 Unity3D

Unity3D, by Unity Technologies, is a real time 3D engine which can handle huge databases, stream geometric asset bundles, pre-calculate lighting or perform real-time lighting, provide interactions and animations capabilities, attach behaviour scripts to objects, while preserving a fluent visualization of the final product.

Unity represents y-axis to be in the upward direction whereas in real-world, the z-axis point up. The effect of this transition causes objects imported in Unity to be off orientation. The fix is easier when placing static models because the rotation can be brought back to its normal orientation using the editor but when models are imported dynamically, they need to be redrawn to transform their axis assuming end pixels to be start pixels and vice-versa.

The indie version of Unity does not support LOD for objects but has a built-in process to which does imply LOD and batching. Setting up multiple terrains to form a large landscape leads to seams along the border of terrains, making it seem partially broken rather than continuous. Unity has a fixed method – SetNeighbours() to overcome this break but this does not really resolve the problem, the terrains still have seams which are obvious. To have seamless terrains, it is necessary to implement terrain stitching by redrawing pixel on the outer edges of a terrain tile match the outer edge of its neighbours.

3.2 Sail navigation

The need to establish a connection between the skipper and the sail was necessary to generate a path using the race waypoints or marks. Using the position of boat during the race, it was required to set up a

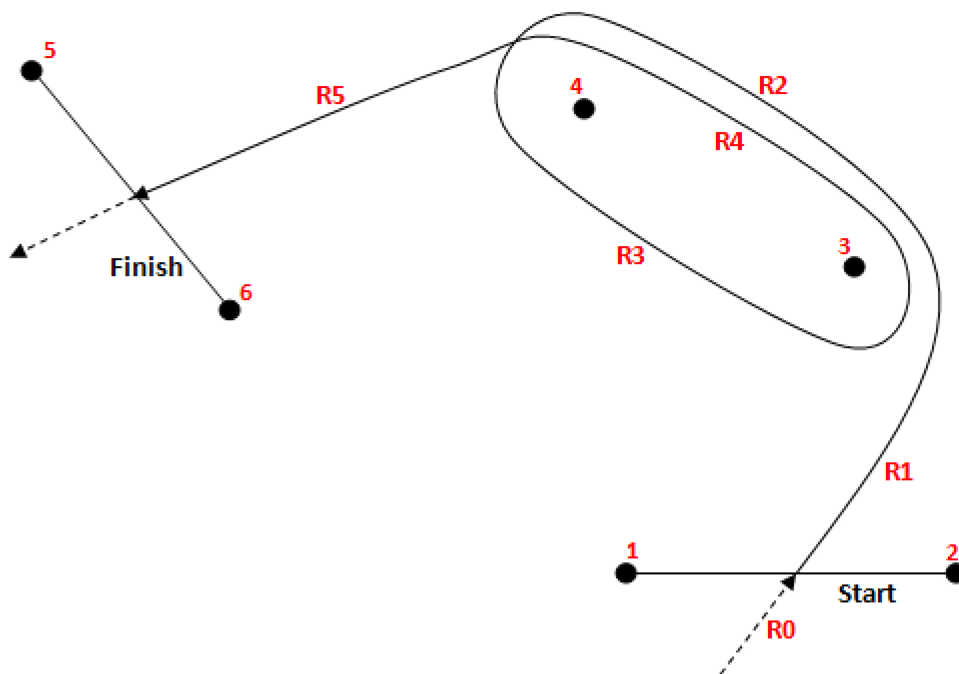


Figure 3.1 Sail regatta course with markings to waypoints and routes

state machine which can transition the state of the sailboat with respect to an approaching waypoint. The states are based on the waypoint tags - start, finish, gates and marks namely. Start and finish denote the line drawn between waypoints marked as start line and finish line; these are also considered as start and final target nodes. Similarly, the gates denote a line or crossing between two waypoints. The marks are termed as single waypoints.

To determine the state of a waypoint, the route path is required. Route information is recorded based on the waypoint tags. For example, during the race event, the sailboat are set to sail behind the start line and once the race starts, the distance travelled by the sails to reach or cross the start line is consider to be the initial course of the route. Similarly, the route taken from start line to the preceding waypoint is taken as the next successive route. The route information is stored in the route array using a route index an identifier. Figure 3.1 illustrates a sail regatta course with markings to waypoints and the route index used to identify routing path.

Waypoint states play an important role in calculating the time take by the sail to cover the set route. Waypoint states have two transition states - crossing and rounding. If the waypoint is tagged start, then the sailboat select the crossing state whereas when sailboats approach a mark, they are set to select the rounding state.

Crossing is a state which takes two waypoints into consideration while making calculations to find the crossing point between these waypoints. Considering an imaginary line drawn between the waypoints to make the process explanatory, the process checks the position of a sailboat and calculates the distance between the imaginary axis and the skipper. As the distance nears zero, it will compare the x, z coordinates of sailboat with the x, z coordinates of the referenced line. If the coordinates match each other, then the sailboat is on the line and when the comparison states a difference as positive i.e. sailboat coordinates greater to the line coordinates. Figure 3.2 indicates crossing points of start and finish line.

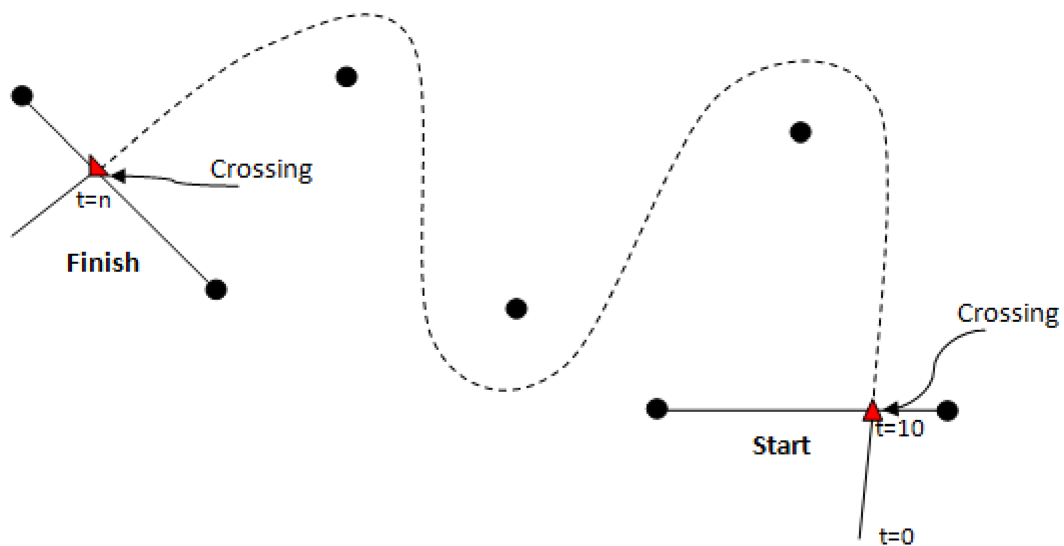


Figure 3.2 Crossing points of start and finish line

Rounding is the second state in the waypoint state machine which takes a combination of three reference waypoint. The mark that a sailboat is set to head towards is considered as the first waypoint. Second and third reference waypoints are taken with respect to the current waypoint, the waypoint immediately after the current waypoint is considered as waypoint next and the waypoint prior to the current waypoint is marked previous. However, the next and previous are referred to second and third waypoints in the rounding state.

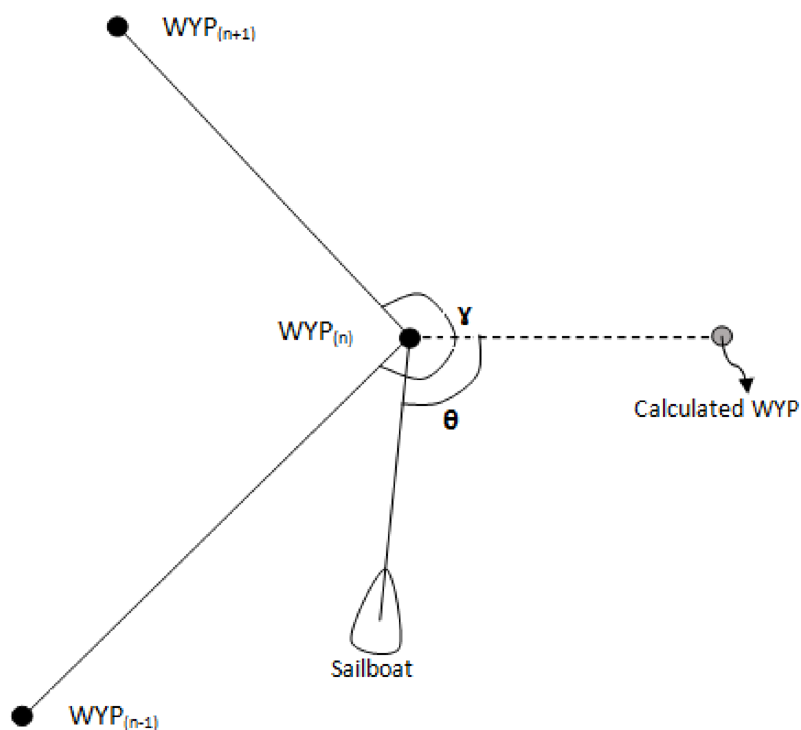


Figure 3.3 Sailboat heading to mark for rounding

A model of the rounding algorithm is represented in Figure 3.3. WYP $_{(n-1)}$, WYP $_{(n)}$ and WYP $_{(n-1)}$ stand for previous, current and next waypoints of the system respectively. The arc tangent is measured at each waypoint with respect to the current waypoint and the difference in angle between next and previous is used to measure angle ɣ. The angle ɣ is bisected to form an imaginary axis which acts like the crossing line or endpoint of a route path.

The calculated waypoint as shown in Figure 3.3 forms an imaginary line segment (with fixed endpoint) and if the sailboats were to sail far away from the current waypoints and beyond the calculated waypoint axis, it is hard to determine if the sailboat crossed this axis and it requires more time to rewrite an algorithm to enclose such scenarios. Hence, the introduction of an angle θ is used to measure the angle made by the sailboat with the approaching/current waypoint and the calculated waypoint. Angle θ is derived by keeping current waypoint stationary. Further, this will help in identifying which boat sailed first or was the first to make a round across a mark.

3.3 Ocean Models

3.3.1 Definitions and relationships

The surface elevation, η, is the vertical displacement from the mean water level of the oceanic surface at given point and time. The wave amplitude, a, is the maximum surface elevation due to this wave. The wave height, H, is the vertical distance between a trough and an adjacent crest. For a single wave, $H = 2a$. The wavelength, λ, is the distance between two successive crests. The period, T, is the time interval between the passages of successive crests through a fixed point. The frequency, $f = 1/T$, is the number of crests that pass a fixed point in 1 second. The wave speed or phase speed, $c = \lambda/T$, is the speed of wave crests or troughs. It can also be expressed as $c = \omega/\kappa$, where $\omega = 2\pi f$ is the angular velocity or angular

frequency, and $\kappa = 2\pi/\lambda$ is the wave number. The water depth, d, is the vertical distance between the floor and the mean water level.

The concept of deep water is wave dependent and is related to the ratio of the water depth to the wavelength of the wave. This comes from the transcendental expression

$$\lambda = \frac{gT^2}{2\pi}\tanh\left(\frac{2\pi d}{\lambda}\right) \qquad (4.3.1)$$

Where $g \approx 9.807\ m\cdot s^{-2}$ is the standard acceleration of gravity. Since $\tanh(x) \approx 1$ when $x > \pi$, and $\tanh(x) \approx x$ when $x < \pi/10$, water is considered deep if $d/\lambda > 1/2$ and shallow if $d/\lambda > 1/20$. This leads to some simplifications in the expressions of the terms given above (Table 3-1).

Table 3-1 Relations between oceanographic terms. The subscript ∞ indicates a deep water term

Transitional depth: $1/20 < d/\lambda < 1/2$	Deep water: $d/\lambda > 1/2$
$c = \frac{\lambda}{T} = \frac{\omega}{\kappa}$ $c = \frac{gT}{2\pi}\tanh(kd)$ $c^2 = \frac{g}{k}\tanh(kd)$ $k = \frac{2\pi}{\lambda}$ $k_\infty = k\tanh(kd)$ $\lambda = \frac{2\pi}{k} = cT$ $\lambda = \frac{gT^2}{2\pi}\tanh(\frac{2\pi d}{\lambda})$ $\lambda_\infty = \lambda_\infty\tanh(kd)$ $\omega = \frac{2\pi}{T} = kc$ $\omega^2 = gk\tanh(kd) = gk_\infty$	$c_\infty = \frac{gT}{2\pi}$ $c^2_\infty = \frac{g}{k}(\frac{g}{\omega})^2$ $\lambda_\infty = \frac{gT^2}{2\pi}$ $\omega^2_\infty = gk$

3.3.2 Parametric equations

The base model for ocean waves simulation comes from the linear (or small-amplitude) wave theory in [30] ,widely used in oceanic engineering, and in Computer Graphics in [31]. It describes waves with a sinusoidal shape, which corresponds to calm weather conditions. Considering a location $x_0$lying on a 1D surface at rest, the elevation $z = \eta$ at time t due to a wave with wavenumber κ, amplitude a and angular velocity $\omega = \sqrt{g\kappa}$ is

$$z(x_0,t) = a\cos(\kappa x_0 - \omega t) \qquad (4.3.2)$$

When the steepness of the wave increases, its crests become sharper and its troughs flatter. Therefore, document [10] used a more realistic description based on trochoids. The surface equation is now

$$\begin{cases} x(x0,t) = x0 + a\sin(\kappa x0 - \omega t) \\ z(x0,t) = z0 - a\cos(\kappa x0 - \omega t) \end{cases} \qquad (4.3.3)$$

Where z_0 is the surface elevation at rest (typically 0). Note that the important point here is the phase term difference of $-\pi/2$ between $x(x_0,t)$ and $z(x_0,t)$, not the use of a particular sine or cosine function. This Lagrangian model describes the trajectory in a vertical plane of a particle (x, z) around its position at rest (x_0, z_0) [32].Summing several waves and extending equation 3 to a 2D surface gives

$$\begin{cases} \vec{x}(\overrightarrow{x0}, t) = \\ \overrightarrow{x0} + \sum_{\vec{k}} \hat{k}\, a(\vec{k})\sin(\vec{k}.\overrightarrow{x0} - \omega(\kappa)t + \varphi(\overrightarrow{k})) \\ \vec{x}(\overrightarrow{x0}, t) = \\ Z0 - \sum_{\vec{k}} a(\vec{k})\cos(\vec{k}.\overrightarrow{x0} - \omega(\kappa)t + \varphi(\overrightarrow{k})) \end{cases} \quad (4.3.4)$$

where$\vec{x}$ = (x, y) is the horizontal particle position at time t, $\overrightarrow{x0} = (x_0, y_0)$ its position at rest, $\vec{k} = (k_x, k_y)$ a wave vector, i.e. a vector with magnitude$\|\vec{k}\| = \vec{k} / \|\vec{k}\|$and direction of the considered wave propagation and is the unit vector of$\hat{k} =$ Because of the presence of uniformly distributed random phase terms, $\varphi(\overrightarrow{k}) \in [0, 2\pi]$, this model is also known as random waves. Any set of $\overrightarrow{x0}$ can be taken, which allows the use of adaptive mesh, for optimal surface sampling [33].

3.3.3 Ocean Wave Generation

Another method for computing equation 4 is the 2D inverse Fourier transform, used in [34][35][14].

The set of $N \times M$ complex numbers $Fn,m$is transformed into a set of complex numbers fp,q by

$$fp,q = \sum_{n=1}^{N} \sum_{m=1}^{M} Fn,m \ \exp\left(i2\pi\left(\frac{np}{N} + \frac{mq}{M}\right)\right) \quad (4.3.5)$$

Where $p \in \{1, .., N\}$ and $q \in \{1, .., M\}$. This is of particular interest because the fast Fourier transform (FFT) algorithm is an efficient way to compute these sums. However, in comparison with the previous method, its usage is more restrictive. The set of $\overrightarrow{x0}$ is defined as a regular grid of $N \times M$ points with dimensions $Lx \times Ly$ and origin at the center, such that $\overrightarrow{x0} = (nx_0Lx/N, my_0Ly/M)$, where $nx_0 \in [-N/2, N/2[$ and$mx_0 \in [-M/2, M/2[$ are integers. Equation 4 rewritten as

$$\begin{cases} \vec{x}(\overrightarrow{x0}, t) = \\ \overrightarrow{x0} + I \sum_{\vec{k}} \hat{k}\, A(\vec{k}, t) \exp(i\,\vec{k}.\overrightarrow{x0})) \\ \vec{x}(\overrightarrow{x0}, t) = \\ Z0 - R \sum_{\vec{k}} A(\vec{k}, t) \exp(i\,\vec{k}.\overrightarrow{x0})) \end{cases} \quad (4.3.6)$$

Where $I()$ and $R()$correspond, respectively, to the imaginary part and real part of the expression, and

$$A(\vec{k}, t) = \mathrm{a}(\vec{k})\exp(i(\overrightarrow{x0} - \omega(\kappa)t + \varphi(\overrightarrow{k})) \quad (4.3.7)$$

Rather than the random phase termφ, authors commonly use normally distributed random amplitude. To comply with equation 5, the set of wave vectors must be

$$\vec{k} = (2\pi \frac{nk}{Lx}, 2\pi \frac{mk}{Ly}) \quad (4.3.8)$$

Where n_x and m_y are defined as nx0 and mx0 above. The resulting surface is periodic in all directions and can be used as a tile [14].To decrease the FFT computation time by a factor two, real numbers can be used instead of complex ones. This is achieved by adding two waves with same amplitude and travelling in opposite directions [14]. Equation 4.3.7 becomes

$$A(\vec{k},t) = a(\vec{k})\exp(\ i(-\ \omega(\kappa)t + \varphi(\vec{k}))) + a(\vec{k})\exp(\ -i(-\ \omega(\kappa)t + \varphi(\vec{k}))) \quad (4.3.9)$$
$$= 2a(\vec{k})\cos(-\ \omega(\kappa)t + \varphi(\vec{k})).$$

This results in a single stationary wave with time dependent amplitude.

3.3.4 Ocean Wave Spectrum

The shape of wave spectra is still under investigation due to absence of a unified universal form. Significant difference exists between these spectra and unfortunately, there are further examples that exhibit similar divergence in [36][37][38].The analytical spectrum that is constructed, attempts to take advantage of spectra developments. Apel's objective of building an analytical spectrum that can be easily used in modelling electromagnetic interactions with the sea surface. Indeed, the primary need of EM models is the autocorrelation of the displacement field or its Fourier transform, i.e., the elevation spectrum [39].

Omnidirectional spectrum expressed as a sum of two spectra regimes:

$$S(k) = k^{-3}[B_l + B_h] \quad (4.4.1)$$

Where subscripts l and h indicate low and high frequencies, respectively, and B stands for the curvature spectrum $(B= K^3S)$.

3.3.4.1 *Long-wave curvature spectrum*

Here B_l is written in general form as

$$B_l = \frac{1}{2}\ \propto_p C_p/C\ F_p \quad (4.4.2)$$

Where $\propto_p$ is the generalized Philips-Kitaigorodskii equilibrium range parameter for long waves dependent on the dimensionless inverse-wave-age parameter $\Omega = U_{10}/C_p$($U_{10}$is the wind speed at a height of 10 m from the water surface) , C(k) is the wave phase speed and $C_p = C(k_p)$ is the phase speed at the spectral peak, and $F_p$is the long-wave side effect function dependent on dimensionless paramters $k/k_p$and $\Omega = U_{10}/C_p$.

The long- wave side effect function $F_p$is given by

$$F_p = L_{PM}\, J_P exp\left\{-\frac{\Omega}{\sqrt{10}}\left[\sqrt{\frac{k}{kp}} - 1\right]\right\} \quad (4.4.3)$$

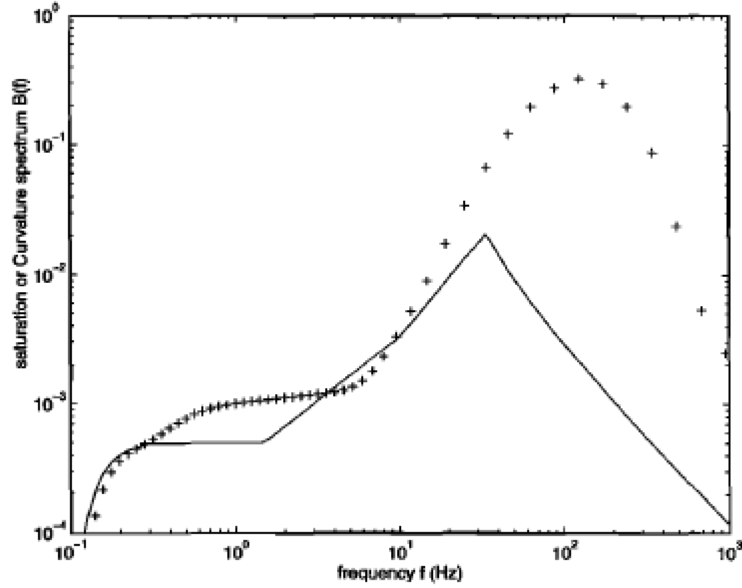


Figure 3.4 Frequency curvature spectra

3.3.4.2 *Short-wave curvature spectrum*

The high-frequency curvature spectrum, B_h ,is expressed as

$$B_h = \frac{1}{2} \propto_m C_m / C\, F_m \qquad (4.4.4)$$

where $\propto_m$ is the generalized Phillips-Kitaigorodskii equilibrium range parameter for short waves dependent on the dimensionless parameter (u^*/c_m), where u^* is the friction velocity at the water surface, c is the short-wave phase speed and C_m is the minimum phase speed at the wave number (k_m) associated with a supposed gravity-capillary peak in the curvature spectrum, and F_m is the short-wave side effect function dependent on dimensionless parameters k/k_m and $\Omega_m = u^*/C_m$.

The short-wave side effect function in (40) is taken here as

$$F_m = exp - \frac{1}{4}\left\{\left[\sqrt{\frac{k}{km}} - 1\right]\right\}^2 \qquad (4.4.5)$$

Directional wave spectrum satisfactorily models surface waves from near the main spectral peak up to the gravity capillary peak. Non developed seas were also modeled by using the modified JONSWAP formulation for long waves together with a new fetch versus wave age relationship that extends its validity to large fetch values in Open Ocean. The secondary gravity-capillary peak increases with the wind friction velocity as a two-regime logarithmic function as derived from *Jiihneand Riemer*and *Hara et al* tank data. The omnidirectional spectral model reproduces significant wave height for developing seas and measured mean square slopes (mss) for both clean and oil-slick covered water surfaces (Figure 1).Moreover, when combined with simple spreading function, upwind-crosswind asymmetry also verifies for both clean and slick mss to within experimental errors.

$$\psi(k,\varphi) = \frac{1}{2\pi} k^{-4} [B_l + B_h][1 + \Delta(k) \cos 2(\varphi)] \qquad (4.4.6)$$

Where*B1*, *Bh'* and Â are subsequent formulas from paper Unified directional spectrum [39] With this modification, the curvature secondary peak moves over to 1.7 cm wave1ength (Figure 3.4).Wind speed, Wind Amplitude, Inverse wave age are three important parameters which can be adjusted accordingly. These setting can be used to control the look of the waves from rough seas to calm lakes. A higher wind speed gives greater swell to the waves, Wind amplitude Scales the height of the waves and the value of

Inverse wave age is lower number means the waves last longer and will build up larger waves. This approach is very flexible and can be used in conjunction with future in situation data to refine these initial findings.

3.4 Simulating & Animating Ocean Waves

Procedural model for breaking ocean waves that is intended to be used for interactive visualization. The movement as well as the appearance of the waves is modelled by a set of functions in dependence of time and space. This continuous surface description allows it to calculate all properties of every point (including foam) on the ocean surface at every time without any information from previous time steps. By using an adaptive sampling scheme for rendering, the frame rate of the animation only depends on the screen resolution rather than on the model size. The model is quite simple, easy to implement, fast to compute and provides a visual appealing interactive animation of infinite large ocean coast scenes. On the other hand it provides only limited flexibility due to its procedural character. For achieving more realistic scene appearance, it may also easily be combined with models for deep-water waves presented in the past.

The challenge of this work is a procedural method to be used for simple interactive ocean animating. The appearance of the ocean scene can be computed every time without the need for transferring information over time. Furthermore, in combination with a rendering method that uses adaptive sampling, the output frame rates are decoupled from the size of the ocean scene. On the other hand, in a procedural model everything has to be modelled by hand so that much work has to be done to obtain a more realistic model[40]. This includes the support of different wave forms (for instance spilled breaking waves) and allowing different wave sizes and a more complex wave behaviour. Furthermore, assuming a given ocean scene, a mapping operation that automatically adapts the parameterization for the breaking waves (for instance for wave refraction) would be highly desirable. Many more components have to be included to enhance the realism. The model can easily be combined with models for deep-water waves (including sinusoids, trochoids or FFT based approaches and small wave ripples modelled as animated bump maps .Aside from the waves, environment reflection maps can be used to model sky reflections on the ocean surface by using graphics hardware.

3.5 Terrain generation

Unity holds a built-in terrain system which comes in handy while developing landscape based appplications. The system allows for editing terrains, storing heightmaps, detail mesh positions, tree instances, and terrain texture alpha maps. Unity‘s manual [27] provides a detailed overview explaining the various options available for terrains and how to make use of them.

Heightmaps are 2-D scalars which represent an elevation value and these values play a vital role in interpreting terrain in 3D. Heightmaps can be generated using various tools or software’s. Unity is featured to automatically read heightmaps both internally and externally, provided that an external height map is of the required format - .raw. However, the implementation of sourcing height maps using other formats such as .TIFF – required manual effort of creating a .TIFF reader to extract elevation data.

A RAW file contains a set of information recorded by a sensor. They are referred as digital negatives as they play similar role as negatives used in film photography. Bruce Fraser [29] clearly explains how RAW data is captured using photosensitive detectors which are arranged in a 2D grid, each element of the array representing one pixel. The values attained through these sensors are directly proportional to the

amount of light captured by them and correspond to grayscale values. The steps post retrieval of raw data are used in conversion of grayscale to color using color filter array.

Unity does not have any constrains set on terrain's size. This means that we can have huge terrains but from our experience with unity3D during the initial phase of the project, we realized that having large terrains to fulfill the idea of constructing the real-world dynamically would slow the generation process and cause memory issues. To overcome these issues, the approach of breaking terrains into smaller regions to cover a larger area arose but this again just helped improve the load time of terrains. The issue pertaining to memory remained and this led us to generate terrain tiles based on a sailboats location.

We implement a tilling system by tacking the sails geographic location in terms of latitude, longitude coordinates and map them to real-world map data. Using the sails position, a region is draw around the sail marking its boundary for terrains to be generated. At this stage, we divide the bounded region into n*n grids for the generated terrain tiles to be placed in the right order corresponding to real world map coordinates. Considering the terrains dimension - width and length, a scale factor is introduced to determine the maximum number of terrains in a grid.

$$n = (2 * (scale_factor+1)) - 1; \quad (3.6.1)$$

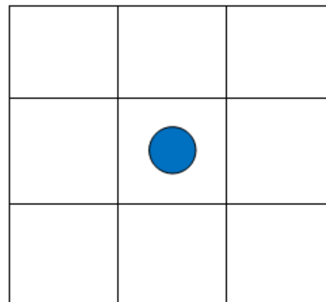


Figure 3.5 Arrangement of terrain tiles around a point of interest

In Figure 3.5, the blue dot represents the sail boat and the grid tiles denote the terrain region around the sail.

The position of a sailboat is dynamic at all times through the race and this eventually means that terrain tiles have to adapt to such change in position of sails. For dynamic terrain generation, we implemented a boundary limit within the terrain grid. By tacking the position of sails every frame, we can retrieve the current cartesian coordinates(x,z) of the sail. The terrain tile index on which the sail is located can be computed by dividing the sailboats x coordinate by terrain width and z coordinate by terrain length.

The boundary limit is determined by the difference in the number of terrain tiles crossed with respect to the sart position within a given grid. The boundary limit was a matter of choice in our case, the cutoff limit, could be m(x,z) where m – difference in position along x znd z respectively, such that $m < n$ (from equation).

$$m = \text{Current position} - \text{Start position} \quad (3.6.2)$$

When the sailboat moves within the same terrain tile, there is no change in the grid system containing the terrain set but when the sailboat crosses a terrain tile, the grid system has to makes the necessary changes to make sure the sailboat is centred to the grid at all times. A pictorial representation of the system is presented in Figure 3.6.The blue dot is moved to the left and when this happens, a new set on tiles (green boxes) are generated around the blue dot. The terrain tiles outside the grid area (shaded in red) are removed from the system.

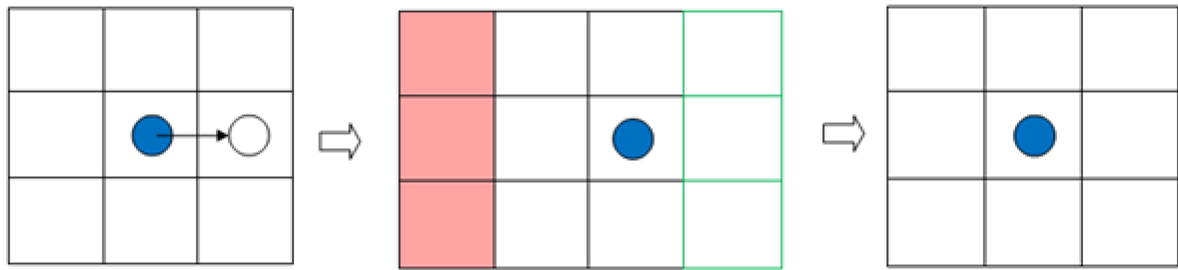


Figure 3.6 Re-arrangement of terrain tiles system

When terrains are re-positioned to maintain the grid structure, it is not necessary to re-generate all terrains within the grid. Only the missing terrains are re-drawn and using this approach, the memory consumption of the terrain system remains constant and is directly proportional to the grids size. There is a problem when the change is position is not smooth or consistent, it leads to dropping wrong terrain tiles around a target boat. To overcome such situations, a boolean is introduced to check if the terrain load is in progress and how big is the change in difference with respect to the current position of the sailboat. If the change in position is large and there is an existing terrain load in progress, the terrain load is stopped and all existing tiles are removed from their original position and a new set of terrains are generated using the new position as target point.

4 Implementation and Result

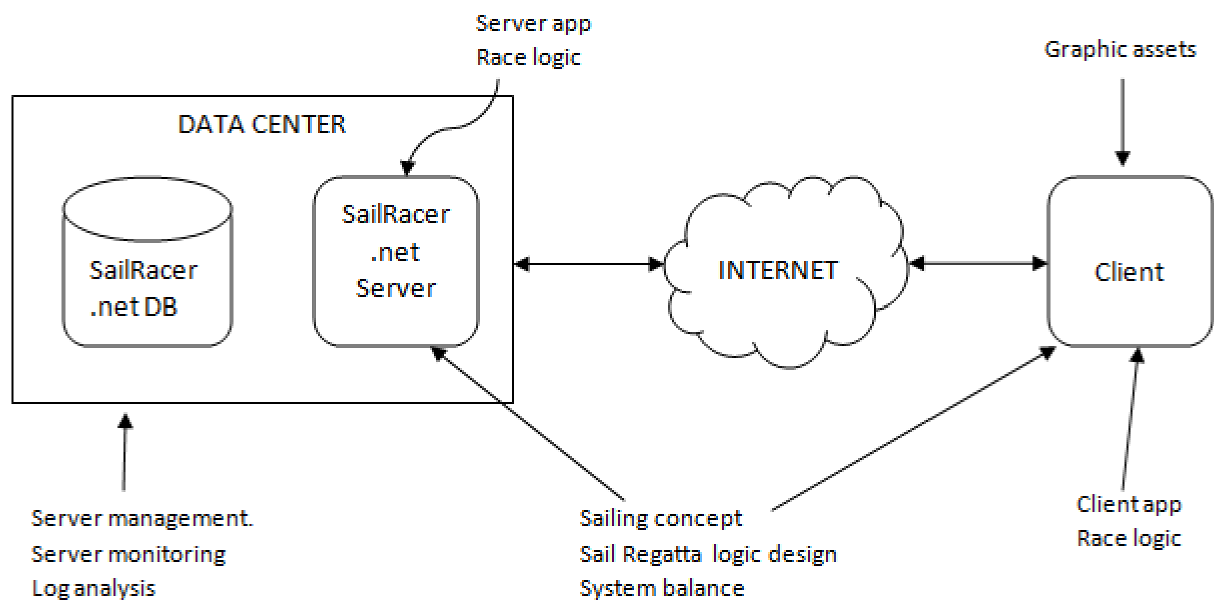


Figure 4.1 Architecture overview of the 3D visualization system

A datacenter was initially put in place to test the data processing methodologies and the access to SailRacer server and SailRacer DB were granted. The server application and race logics were written and processed on the server-end. The scripts which carried out logic processing, directly communicated with the MySql DB to fetch race information. The DB stored vital race information which included GPS coordinates, time, mark data, track information etc.

Similar to most software development kits, Unity allows local or standalone development and build to publish the files onto a server. The focus on graphic assets and performance were implemented using the Unity game engine and the implementation of processed data from server is used to complete the race logic.

An overview of the 3D visualization system is presented in Figure 4.1. The sailing concepts and regatta design were sourced from SailRacer‘s expertise and it was used to implement a bilateral processing using their data center. The workflow is such that when a client request a connection to the SaiRacer system, the page is available for access to clients can access and view the 3D race system.

The main components used in the race system are the Boat Controller and Boat Information blocks. Figure 4.2 shows the layout and relation between the various classes implemented within the sail race system.

4.1 Boat Controller

The boat controller is set to be the manager, controlling all the boats that participate in the sailing regatta. The Boat Controller uses an event‘s ID to set out and retrieve data from the server, this is done by sending

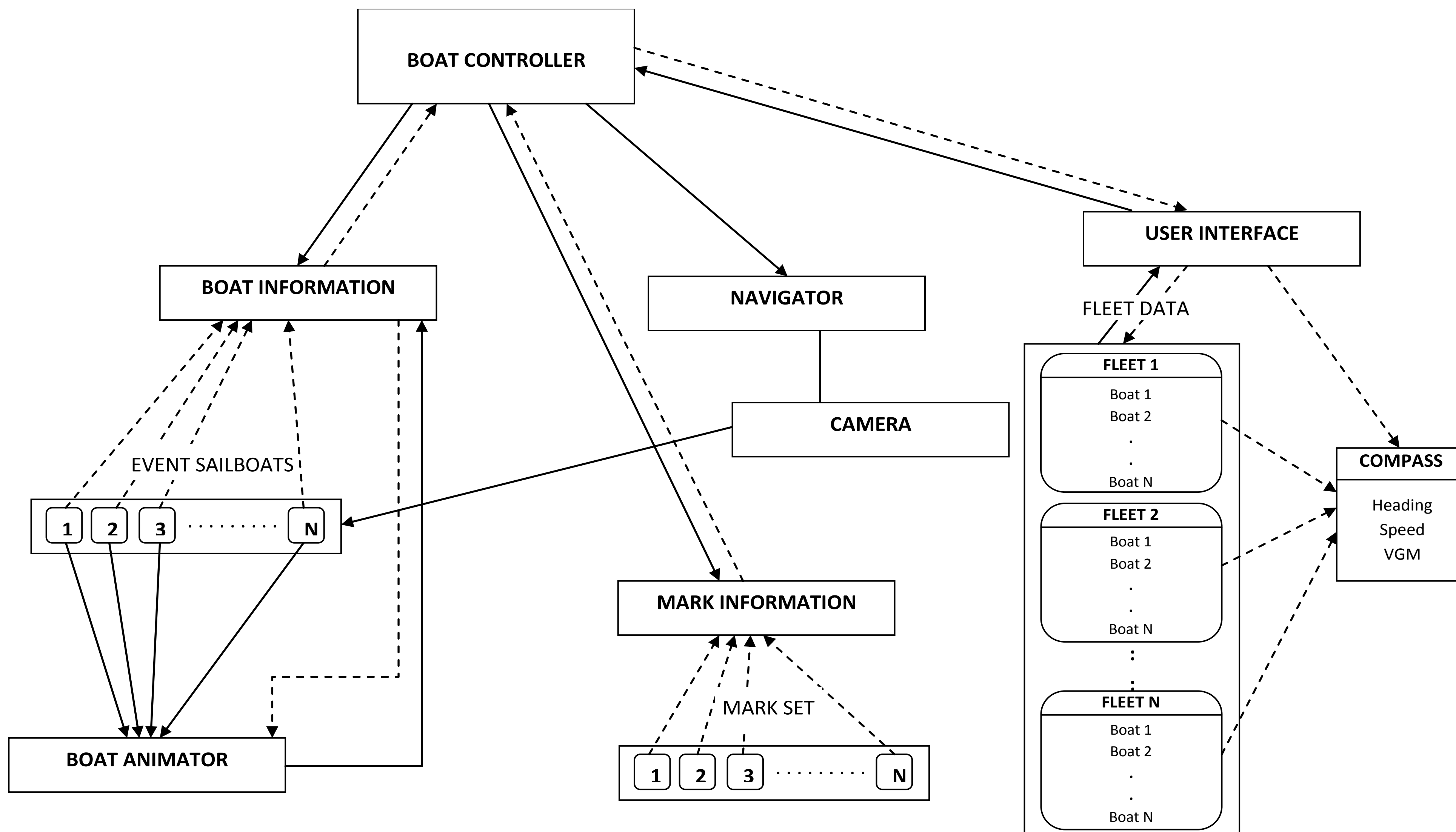


Figure 4.2 Block diagram of the sail system

a HTTP request to the server to get information of event data which include registered sailors and races tagged to an event.

The event data is send to the user interface to give the viewer the option of selecting any particular race of interest. The selected race triggers a Loadrace() call which does the same as the name describes, loads the race. Loadrace(), sets the stage for sailboat to start sailing but before that it send a request to the Sailracer server asking for race specific details. Since an event has many races, the intent is to start with a selected race and progressively load the subsequent races that start later to the selected race.

When race specific data, such as position of marks, course data, skipper information, start and end time of race are retrieved from the server, they are stored in separate arrays. The marks array uses the stored information and pushes a game object to the scene. Course and skipper information is sent to user boat. A user boat is a game object, in this case it is a boat structure which was modeled in 3dsMax and imported into Unity, representing a skipper. Looping through all the skippers available in a race will help setting individual attributes for each skipper; this data is useful at the later stage of processing within the boat information block. This data will also include data corresponding to the individuals track start and end time.

The second stage of the user interface is revealed after the selection screen. This stage displays data relevant to a race and overlays a boat selection list on the upper-left corner of the screen. The list is segregated by fleets containing the boats sailing in each fleet and this is retrieved by grouping the data available race using their fleetID. A seek bar is displayed towards the center bottom of the screen to show progress of the race in terms of time and calculated using the difference between the event‘s end time and start time.

The interface displays the heading, speed, VMG specific to the selected boat. These data are derived from the Boat information block which is described in the following section. A camera selection control is placed for swapping between two views, Orthogonal and perspective namely. The camera perspective mode is set to activate on race start but can be switched to orthogonal at anytime of the race. The camera is controlled by the navigator, which is tagged to the ocean water in this project. By transforming the position and adjusting the quaternion rotation angles, it is easy to swap between the perspective and orthographic views.

Boat selection is enabled by using the boats order in an array and setting it as an index value, the camera is transitioned between boats depending on the viewer choice of selection. Further, each boat is made to display some information about itself. In this case, the boat name, speed and vmg are chosen to be displayed on moving the mouse pointer over a sailboat. This information is drawn on screen, pulling data from the boat information class and using the camera index match the boats index.

4.2 Boat Information

Boat information is sailboat specific and sends data to the boat controller to keep track of it position in a race. The boat information class requests the server to retrieve a set of coordinates corresponding to the boats position in real time. The coordinates received in unity are pre-processed on the server by converting latitude and longitudes quires from the DB to Cartesian coordinates. This is done to reduce application overhead. Apart from just receiving the x,y coordinates of a boat, the data is tagged with a timestamp measured in seconds, which again is a server side processing. The data set is stored in an array and iterating though this array helps interpret the path taken by the boat in real-time.

Two points from the data set are considered each having coordinates x1, y1 and time t1. Similarly, point two forms x2, y2, t2. Using these parameters the change in distance is known and speed can be

achieved using the formula distance = speed * time. Next, the position of the boat for a given route is calculate and this is done by taking the route index 0, 0 being the first in the route table and get the start and end marks of that route index. As described in Figure 4.3, using the position of boat, the distance to the route line is calculated, boat to mark and route point to mark are determined.

Completed distance = Route distance - Route point to mark distance.

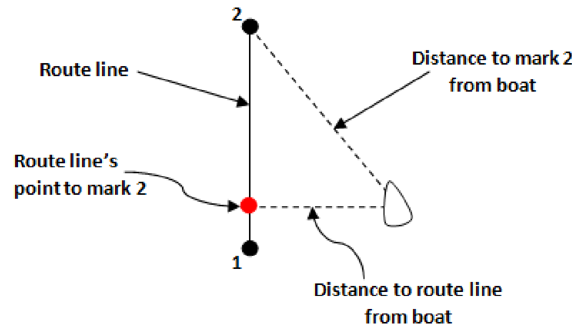


Figure 4.3 Boats position from routeline

The heading to mark is calculated using the arc tangent the mark makes with the boat, the direction of wind is not taken into consideration at this stage of development. To track the position of a boat in the race, it sent a request to the boat controller checking for the leading boat for the result table and validates the difference in distance. If the difference is greater than zero, then the current boat gets registered as the first boat in the results table.

4.3 Ocean Waves

The main goal is to make a realistic representation of the ocean wave pattern, capable of running on a laptop with a decent graphics card. There are several ways to implement a program like this regarding workload balance, each with their own strengths and weaknesses. With the hardware available today there are roughly two main strategies to choose amongst. The first strategy is to let the CPU do all the calculations regarding the wave field, and the GPU take care of the rendering with the requested effects like lightning and reflections. This requires communication and data transfers between the CPU and the graphics device. In Current days GPU's, it is also possible to do all the calculations on the device itself. The strength of this approach is that there is no need to transfer any data between the CPU and the GPU, thus removing the transfer rate bottleneck. Compared to the CPU, the GPU has an enormous capacity.

As a result, computing the water elevation on the GPU will be less time consuming. But the algorithms are more complicated to implement. With a correct setup of the first strategy it doesn‘t require latest graphics card. In addition, the domain size of the FFT calculations is relatively small so the computational time of this domain is small compared to the total time needed to draw one frame. Testing on two computers, the computation time for FFT was 4% and 7% of total computation time. Therefore not much performance is gained by doing the computations on the GPU. The cpu is therefore chosen to perform this work.

Figure 4.4 gives an overview over the most important classes, attributes and methods. The class Main takes care of setting up the windows environment. It also holds the ocean related classes needed for the simulation and visualization.

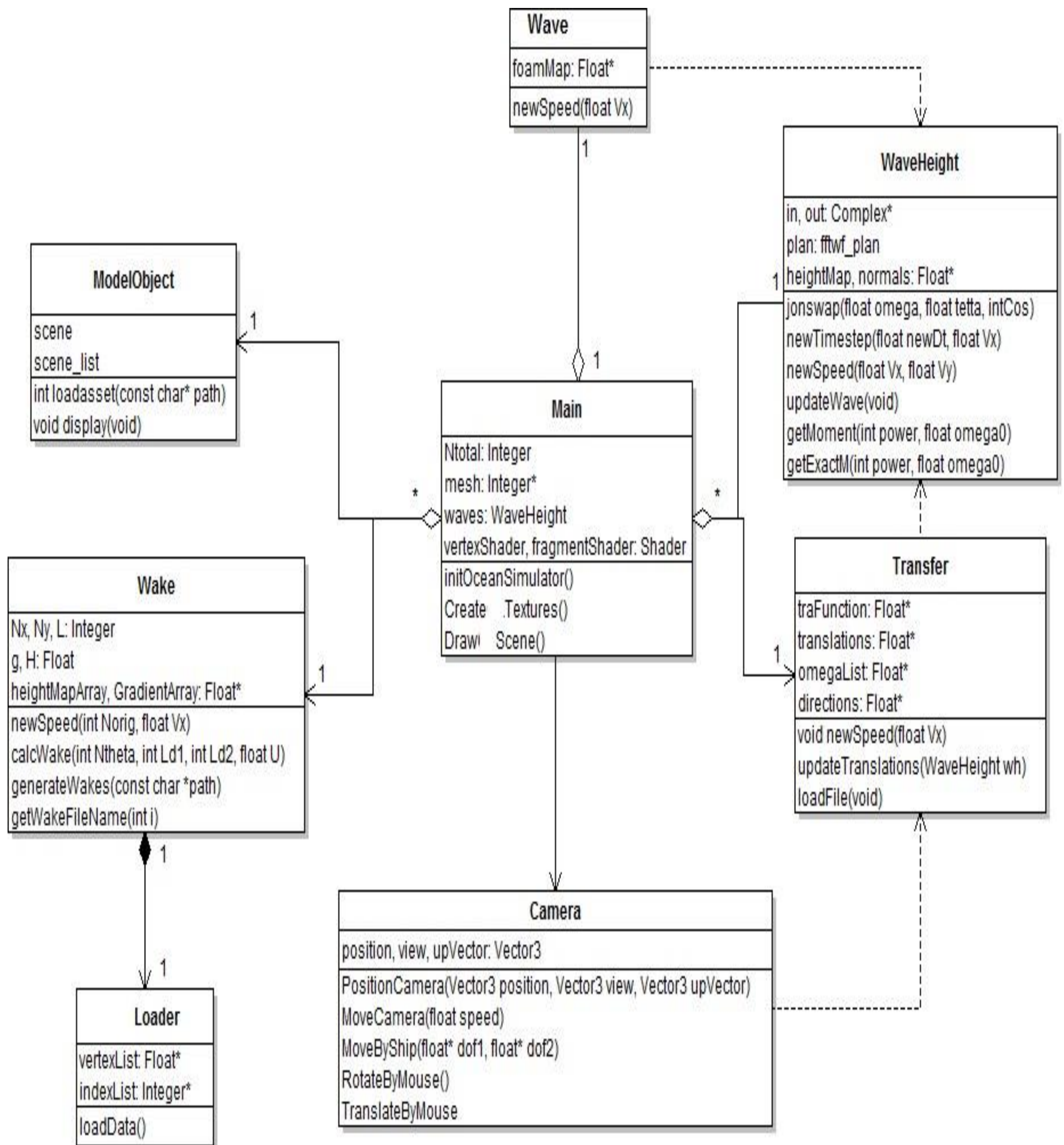


Figure 4.4 Class Diagram of Ocean Simulator

The Camera class uses mouse and keyboard input in order for the user to move around in the 3D environment. Two different views for the users were included orthogonal view and perspective view of the ocean. The wave calculations, which give the heightmap and normal maps are implemented in the class WaveHeight. ModelObject is a class used to load 3D models. The Wake class calculates the wake with different vessel velocities based on the hull of the boat. To add reality to the scene, Foam is added when certain criteria are met.

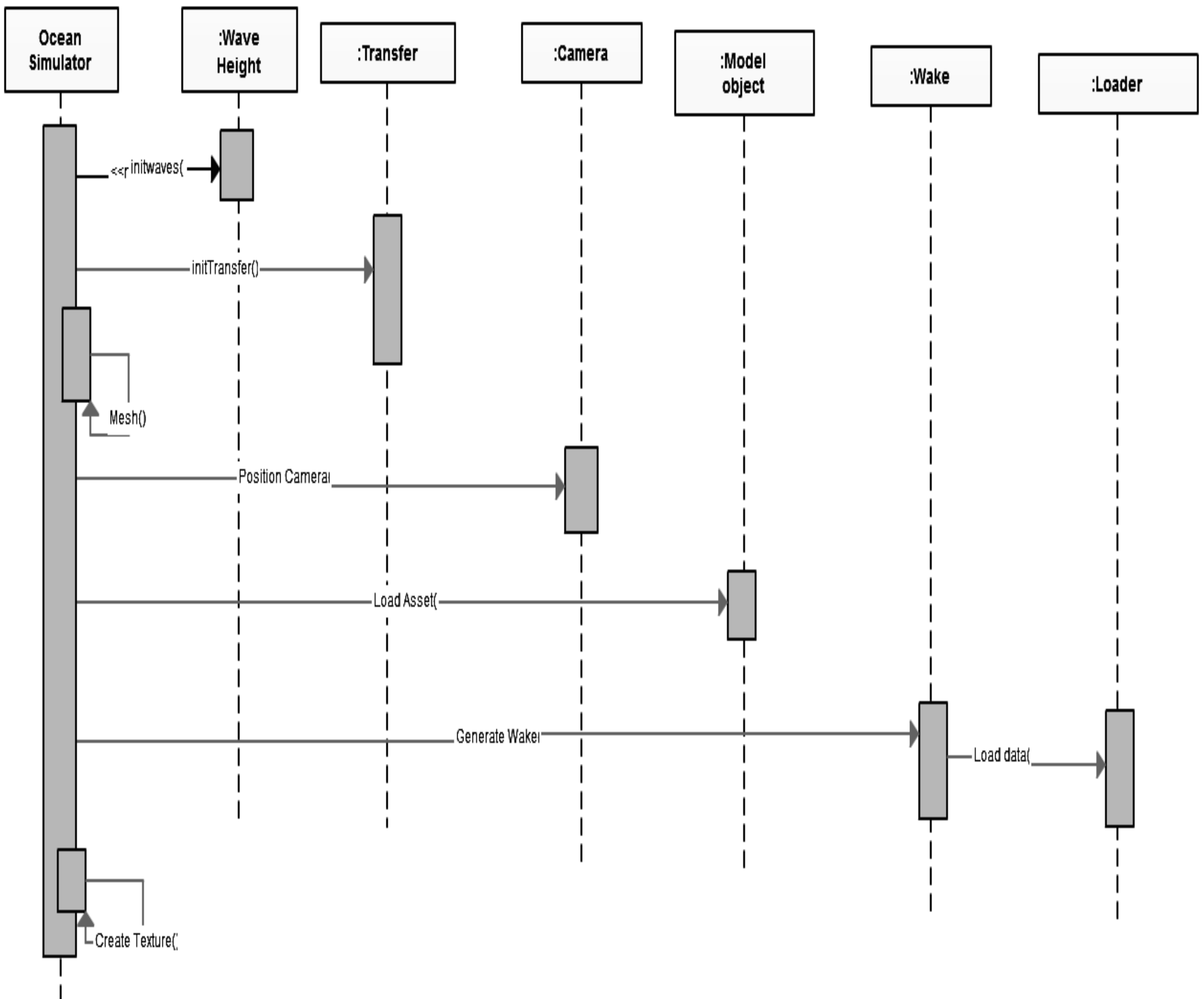


Figure 4.5 Sequence Diagram of setup stage

Figure 4.5 shows the initialization stage of the program. All classes are instantiated with correct parameters and are ready for further usage while running in the main loop. Three Wave Height objects are instantiated with parameters regarding domain, height, spread and period. The 3D model of the boat gets loaded and placed in the center of the scene. Setting up buffers for data transfer between the CPU and GPU is the last stage before the program is ready to run.

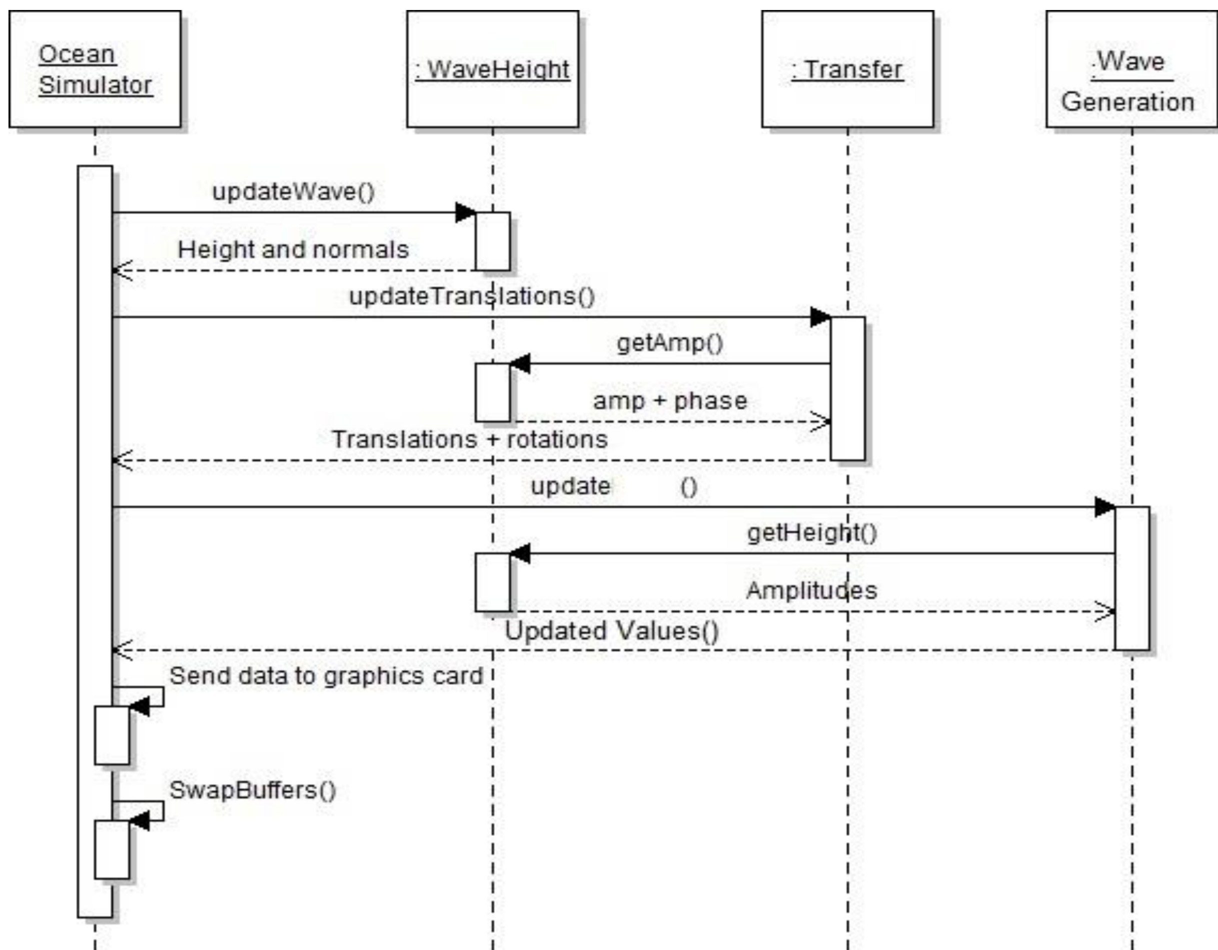


Figure 4.6 Main Loop

Figure 4.6 is a simplified diagram for the main loop. The loop starts by calculating the next step of the wave pattern in the Wave Height objects. Once this is done, the translations and rotations for the boats are calculated based on the data output from the wave patterns. The calculations are also based on the WaveHeight objects. The wave object requests the amplitudes and uses these to calculate where the wake should appear, and returns this information in a mesh. The new data is then sent to the GPU and finally drawn to the screen with SwapBuffers.

In addition to the events in the main loop there are two triggered events that may occur, time scaling and keyboard inputs. The program is scaled with time in order to get a constant speed regardless of the frame rate achieved on a given computer. The frame rate on one specific computer can also vary quite much as it depends on the current field of view. For instance, turning the camera away from the boat model only looking at the ocean can double the frame rate. The time scaling is implemented by measuring the time spent on doing all the calculations and the drawing of the scene, which essentially is the time spent for a single iteration in the main loop.

When a change in frame rate is detected the WaveHeight objects are updated with the new time and scaled properly. This ensures that the waves move with constant velocity without any glitches. Keyboard and mouse inputs can trigger a whole set of function calls. These inputs are mainly used to change wave parameters, and controlling the camera.

4.4 Generating Wave

Generating a realistic wave elevation is done by combining Jonswap spectrum with inverse fast fourier transformation (IFFT). Explanation of the method is described in section 3.3.3 IFFT is used due to its running time. Not using a frequency transformation for calculating the wave height would lead to a vast increase in computation time. Ocean Simulator uses two dimensional IFFT. One dimensional IFFT could be used for wave generation. Even though it gives a slower solution, computation time will still scale as $N^2 \log(N)$. N different one dimensional IFFT has to be computed. Then only the wave length will be restricted, not the direction. Since the current 2D IFFT version is giving satisfying results, the 1D IFFT solution has not been further investigated. The basis for the waves is three 64 x 64 IFFT transformation on different domain sizes. For an IFFT algorithm the physical domain size and the size of the matrix limit both the longest and shortest wave possible to represent. By introducing three different domain sizes, waves up to 512 meter are calculated, without compromising the details of the smallest waves. The results from the different IFFT transformations add together to the final elevation. The computation time for IFFT is $O(N^2 \log(N))$. Having three 64 by 64 domain requires less computation than computing one more detailed IFFT. To avoid duplication of singular waves, each domain is given an upper and a lower limit for the wave length.

Table 4-1 Physical size of domains

Name	Domain size	Shortest wave	Largest wave
small waves	32m	-m	15m
Medium Waves	128m	15m	74m
Big Waves	512m	74m	-m

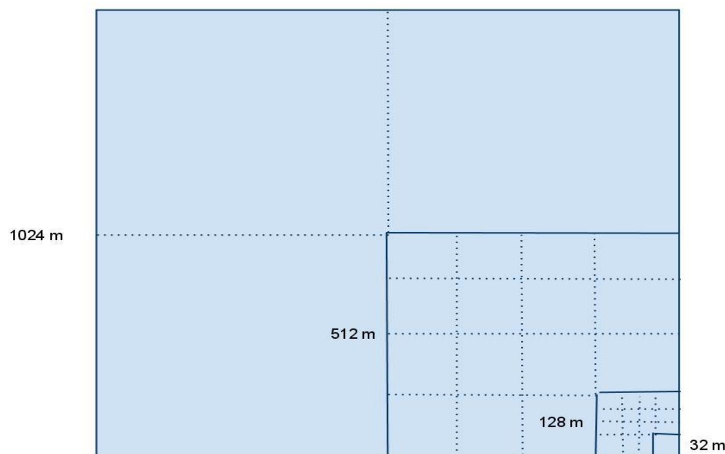


Figure 4.7 Size of different IFFT domains.

The three domains are added together to calculate the final wave elevation

4.4.1 Updating wave

For the each iteration the wave pattern has to be updated. The FPS rate is assumed to be constant, the wave pattern can be updated by multiplying each wave component with a complex value each iteration. A

wave component is the same as the input value for the IFFT. This update factor is a complex number of length 1; with an angle equal to dt.ω the ω value can be found in section 4.3, the updating factor also includes the boat speed. By adding the boat speed to the updating factor, the boat position will be constant relative to the wave domain.

4.5 Generating Wake

The algorithm for generating wake is based on the theory in [15]. A more detailed explanation of the mathematics is found in [16]. The implementation of the wake function has only been tested visually. No systematic testing or error estimating has been done. Visually the results seem promising. It is therefore to believe that the algorithm has no major fault. Figure 4.8 shows the result for wake.

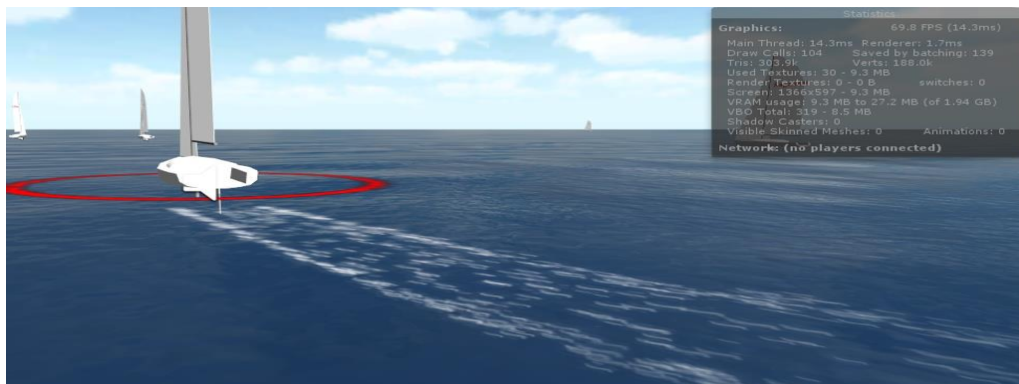


Figure 4.8 Wake created when boat is moving

4.6 Heightmap and Normal Maps

Before anything can be animated or rendered on the screen, the graphics device needs its data. As mentioned earlier all the calculations are done on the CPU, and the results need to be transferred to the GPU. The best way of doing this is by structuring the data in a heightmap, also called heightfield. A heightmap is represented by an image where each pixel, instead of representing a RGB color, represents a

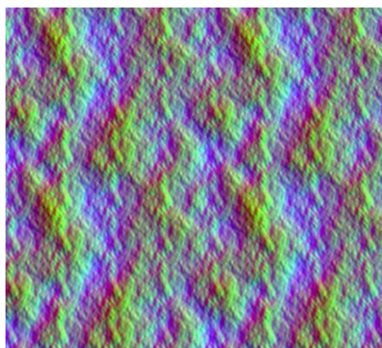


Figure 4.9 Normal Maps

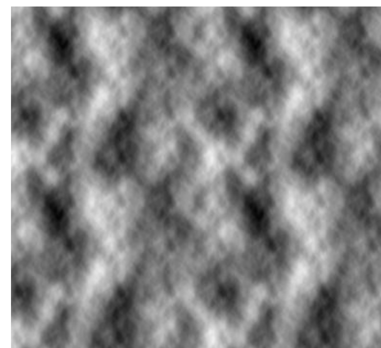


Figure 4.10 Height maps

single displacement value. Heightmaps are often used in geographic information systems to describe terrain. It is also heavily used in the rendering of terrain, and it is a perfect way to store the elevation of an ocean surface.

When visualizing the data in heightmap directly the result is a greyscale image, as shown in Figure 4.10. Black represents the minimum height, and white represents the maximum height. In Figure 4.11 the data from the heightmap have been used to elevate the nodes in a 3D mesh.

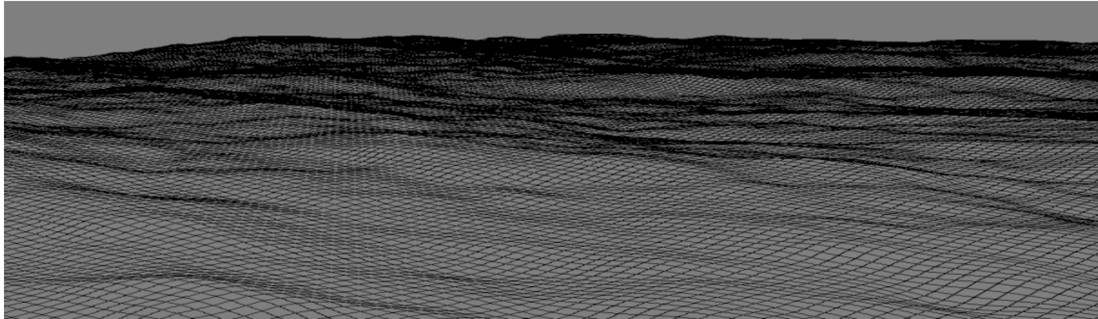


Figure 4.11 Elevated mesh

4.7 Creation of Water Shader

This section describes how shaders have been used to visualize the ocean.There aretwo different types of shaders. The vertex shader sets the position of each vertex. The fragment shader describes the color setting on each fragment. There are two different implementations of the fragment shader.

To create realistic water in this project, the following elements has been used in the water shader: Reflection of the cube map (a sky dome), Refraction of the terrain underneath, Creation of waves (using normal maps), Directional waves (flow maps), Water becomes more transparent at the edges (depth buffer), Foam at the edges (using the gradient texture created in this project), Lighting, like the sun (using normal maps). In this report these items will not be handled in detail, since most of these elements are quite common in the rendering of water in real-time [17].

4.7.1 Vertex Shader

The vertex shader is run for every vertex in the mesh. The vertex shader sets the correct elevation of each node, based on the data given from the heightmaps. This includes the heightmap from the ocean, and the heightmap based on the wake contribution. As described in figure 2.4 the ocean consists of three textures mixed together. These span over different size domains, and they are repeated several times. Based on the vertex position from the mesh, the correct values from the textures must be chosen. A texture is clamped on a domain from [0; 0] to [1; 1]. In order to pick the correct heightmap value, the vertex position has to be converted to a texture coordinate. The code snippet below calculates the texture coordinates for the medium domain and the big domain. ’v’ is the position of the vertex.

```
voidvert(inoutappdata_full v)
                        {
        v.tangent = float4(1,0,0,1);
        float3worldPos = mul(_Object2World, v.vertex).xyz;
        floatdist = clamp(distance(_WorldSpaceCameraPos.xyz, worldPos) / _LodFadeDist, 0.0, 1.0);
        floatlod = _MaxLod * dist;
        floatht = 0.0;
        ht += tex2Dlod(_Map0, float4(worldPos.xz/_GridSizes.x, 0, lod)).x*2.0-1.0;
        ht += tex2Dlod(_Map0, float4(worldPos.xz/_GridSizes.y, 0, lod)).y*2.0-1.0;
```

```
v.vertex.y += ht;
}
```

Once the texture coordinates are created, they can be used to pick the values stored in the textures. First the displacement values are fetched. Only the textures describing the two largest domains are used. The smallest domain is only used for surface details,

```
float2 slope = float2(0,0);
slope += tex2D(_Map1, uv/_GridSizes.x).xy*2.0-1.0;
slope += tex2D(_Map1, uv/_GridSizes.y).zw*2.0-1.0;
slope += tex2D(_Map2, uv/_GridSizes.z).xy*2.0-1.0;
slope += tex2D(_Map2, uv/_GridSizes.w).zw*2.0-1.0;
```

Second, the normals for the vertex are fetched, and sent to the fragment shader for lighting calculations. Fetching a texture to vertex shader is less time consuming than fetching a texture to the fragment shader. The grid of the biggest waves is coarser than the grid 28 of the vertices. Fetching the gradient of the biggest waves can be done in vertex shader without losing data. But the two smaller wave systems have to be loaded in fragment shader.

```
o.Normal = N.xzy;
```

4.7.2 Fragment Shaders

The fragment shader sets the final color of each pixel. Normals are added together from each of the wave textures, and the fresnel effect is calculated. Two different fragment shaders have been made, one version using the skybox approach, and the other using parameterized approach. An option in Ocean Simulator decides which one to use. Both shaders start with loading the texture values and calculating the gradients.

First gradients from the different textures are loaded.

```
half3 bump1 = UnpackNormal(tex2D( _BumpMap, i.bumpuv[0] )).rgb;
half3 bump2 = UnpackNormal(tex2D( _BumpMap, i.bumpuv[1] )).rgb;
half3 bump = (bump1 + bump2) * 0.5;
```

To be able to calculate the intensity of the light, the reflection vector is calculated.

```
half4 water = tex2D( _ColorControl, float2(fresnel,fresnel) );
half4 col;
col.rgb = lerp(water.rgb, _horizonColor.rgb, water.a );
col.a = _horizonColor.a;
```

4.7.3 Reflection

The water uses a cube map to give the environment a “sky”. The cube map acts as a so called “sky dome” to enhance the concept of viewing a virtual world. For reflection and refractions to work, a view vector is needed. Unity3D comes with a function which does this for user. Now that the “eye” vector is known for the water surface itself (either calculated in a shader or from a normal map), the Fresnel term can be calculated.

Using a Fresnel term the amount of reflected light by the viewing angle can be changed. The lower the viewing angle, surface gets more visually reflective. In shader programming commonly an approximation

is used, since a real Fresnel integration would be too much of a computational strain to calculate per pixel. The Fresnel term is calculated in the shader using:

```
float3calcfresnel(float3 viewdir, float3 normal, float R0)
{
floatfresnelBias = 0.2f;
floatfresnelScale = 0.1f;
returnfresnelBias + fresnelScale * pow(1 + dot(viewdir, normal), R0);
}
```

Unfortunately “RenderTextures” cannot be used in Unity3D’s indie version, which means real-time reflection is impossible to perform at the moment. But in this project there are no other dynamic elements in the environment, so this is not much of a problem.

4.7.4 Refraction

Refraction is the change of direction of a wave due to its speed or due to crossing a different medium. Since light also travels in very small waves, refraction also applies. Which means that the land underneath the water should be somewhat refracted. Water has a common refraction index of 1.33[17].The Cg shader language has a standard function to perform this calculation:

```
// Refract the reflection vector, to make the reflection less perfect:
float3 Refract = normalize(refract(normalize(i.eyeDir),final normal, 1.333f));
```

Eventually the refraction was removed from the shader: Refraction works well with transparent water and not for the dark water.

4.7.5 Lighting

Lighting is calculated based upon the water normal map in figure 4.9, the flow normal, and a specified light color and direction. The actual lighting calculation is based upon Lambertian lighting (“N dot L”) which means that the dot product of the normal vector and the lighting direction vector are used to determine whether a pixel should be lighted or not. The intensity of light reflected is calculated by the angle of incidence the light has on the surface. This type of lighting has a rather uniform intensity and the distance does not matter.

4.7.6 Water Transparency

Unfortunately Unity3D’s indie version does not have the ability to use render textures, which means that a depth texture cannot be used. With a depth texture the water transparency could have been altered dynamically based on the distance to the camera. When the camera is closer the transparency should normally increase. In this case the water transparency could be rendered using the water depth texture itself created by the interpolation tool.

The transparency slider is introduced, which controls the transparency of ocean water by controlling the alpha values.

```
o.Albedo = lerp (_SeaColor, skyColor, fresnel) + Sun(V,N2);
o.Alpha = _Transp;
```

4.7.7 Foam

Foam occurs normally when water mixes with air. This happens when the water reaches the shoreline at a coast, but also because of water flowing across uneven surfaces, various directions of water merging each other (like at sea) and so on.

When simulating foam, further manipulation of the color is needed. The following code is only executed when foam is simulated. The foam is simulated by combining a CPU generated matrix with a height check in shader. First it makes a check for simulating whether a wave is breaking or not. If a wave is breaking, the color is changed according to the code below. Foam map is a simple picture used to give texture to the breaking wave.

```
Shader "Foam" {
		Properties {
				_Offset ("Offset", Range (0.00,1.00)) = 1.000
				_Color ("Main Color", Color) = (1,1,1,1)
				_MainTex ("Base (RGB)", 2D) = "white" {}
				_Cutout ("Mask (A)", 2D) = "white" {}
		}

		SubShader {
				Tags { "Queue" = "Transparent-110" }  // water uses -120
				ZWrite off
				Offset -1, -1
				Blend SrcAlphaOneMinusSrcAlpha
				ColorMask RGB
				Pass {
				Color [_Color]
				SetTexture [_MainTex] {constantColor[_Color] combine texture * primary
double, texture * constant double}
				SetTexture [_Cutout] { combine previous, previous * texture }
				}
		}
```

In the shader, foam is also created using the gradient texture (Figure 4.12).

Figure 4.12 Gradient Texture that controls front wave and foam intensity

4.8 Landscape generation

Landscape generation was implemented in a number ways.

1. Static terrain
2. Dynamic terrain
    a. Using .Tiff files from GIS-Lab
    b. Using data from Land Processes Distributed Active Archive Center (LP DAAC)
    c. Using data from Google maps

4.8.1 Static terrain

The generation of static terrain is an implementations of Unity‘s regular terrain system where a terrain game object is drawn into the scene and modified to show bumps resembling mountainous regions and set texture element to give the appearance of a landscape or shore.

Terrain generated for this scenario needs the terrain to follow a sailboat throughout the race and by default, terrain is stationary and to make the terrain change position, it is required to implement a simple logic. The terrain is tagged to follow the ocean plane and camera, thereby making it appear as though a shore is visible to the viewers at all times.

Figure 4.13 is a screen from the outcome of using this approach. The region highlighted in red indicates the surface of terrain, which is drawn by raising the edges of the terrain, giving the appearance of a shore. The height of terrain can be tweaked and depends on the terrain designer.

Figure 4.13 Static terrain representing a shoreline

A drawback that was noticed at this stage was the static appearance of the landscape. The terrain followed the ocean mesh giving a monotonous facade to the landscape. Moreover, this system did not adapt to real-time changes, thereby moving to a dynamic approach.

4.8.2 Dynamic terrain

4.8.2.1 *Using .Tiff files from GIS-Lab*

The Tagged image file format (TIFF) is an uncompressed image derived from RAW images captured by cameras and converted to form TIFF. GIS-Lab has SRTM data available free of charge in several forms including .tiff. Unity does not have a built-in reader to process .tiff file and to set elevation values to a height map, it was necessary to have a custom built .tiff reader.

The implementation of the .tiff reader is based on Richard Baxter‘s [50] open sourced project using the strip method to split an image into separate strips for increased editing flexibility and I/O buffering. The tilling approach is used to load terrains and each terrain is fed with an elevation data corresponding to its DEM value extracted and stored to a float array. The terrain dataset uses the float array to read height values. Edges of the terrain are set using bounds which specify the top, right, bottom and left values of a .tiff file. These values were retrieved from the High dynamic Range imaging data that is stored inside the TIFF image.

Corner Coordinates:
Upper Left (-180.0004167, 55.0004166)
Lower Left (-180.0004167, 49.9995833)
Upper Right (-174.9995833, 55.0004166)
Lower Right (-174.9995833, 49.9995833)
Center (-177.5000000, 52.5000000)

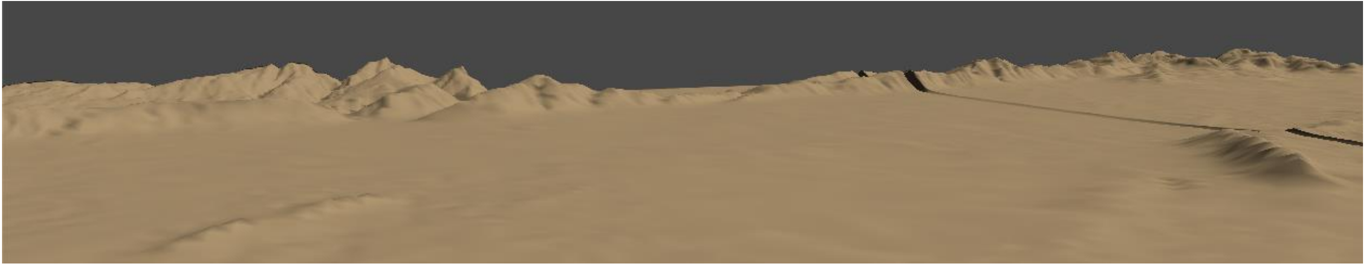


Figure 4.14 Terrain generated using TIFF files

The terrain drawn using this approach is shown in Figure 4.14. When implementing a real world system using this technique, it led to a relatively slow processing of elevation data strips. This downtime eventually led to undrawn terrains and crashed the application since the motion of boats are continuous and lags to synchronize between ocean, land and sails. Moreover, the memory used by the application is large since the resource files (TIFF) are stored within the application.

4.8.2.2 *Using RAW data from LP DAAC*

Land Processes distributed Active Archive Center is yet another data center within NASA Earth which provides direct access to SRTM data. Here, the LP DAAC system provided access to elevation data in JPEG format. The below batch process was written to automatically download large set of files from the server to local system using windows bitsadmin tool. Since these were in a format that needed editing, it was converted to Raw using Photoshop's auto-batch process. This saved lot of time compared to the manual download and conversion process which would have taken quiet a large amount of time.

```
FOR /L %%G IN (50,1,60) DO (
		FOR /L %%H IN (0,1,40) DO (
				IF %%H LSS 10 (
						bitsadmin.exe /transfer DownloadMap
http://e4ftl01.cr.usgs.gov/SRTM/SRTMGL3.003/2000.02.11/N%%GE00%%H.SRTMGL3.jpg.2.jpg
C:\Unity\map\dl\N%%GE00%%H.SRTMGL3.jpg.2.jpg
				) ELSE (
						bitsadmin.exe /transfer DownloadMap
http://e4ftl01.cr.usgs.gov/SRTM/SRTMGL3.003/2000.02.11/N%%GE0%%H.SRTMGL3.jpg.2.jpg
C:\Unity\map\dl\N%%GE0%%H.SRTMGL3.jpg.2.jpg
				)
		)
)
PAUSE
```

A naming convention is used while downloading the DEM tiles from the LP DAAC server. The naming convention helps the terrain tilling system by making it easy for the process to identify its neighbors. In this case, it is necessary to have all the RAW files stored within the Unity project for retrieving the height map data in runtime. Figure 4.15 is a representation of the terrain drawn using the RAW data files with single texture overlaid throughout all terrains.

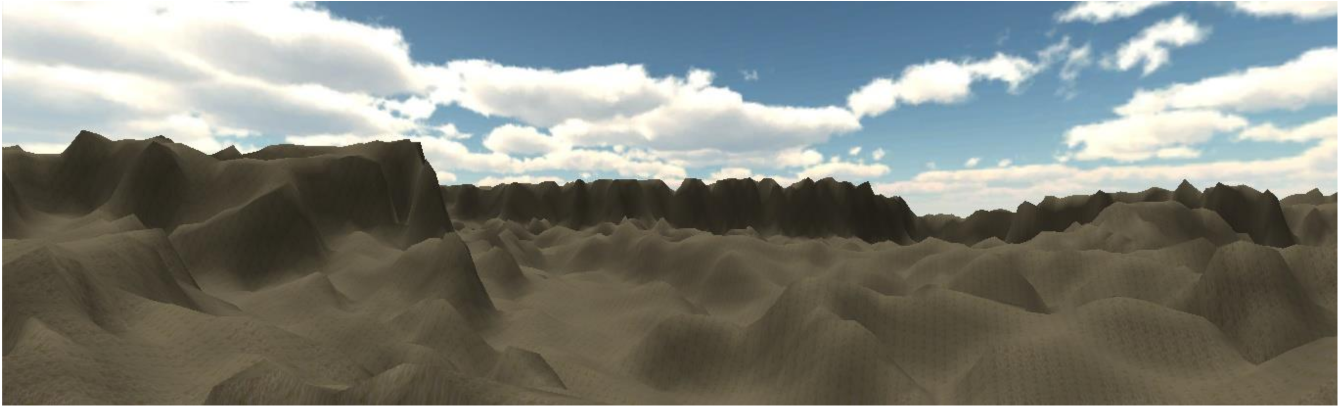


Figure 4.15 Terrain generated from RAW image

Terrains formation rendered as expected but there is no variation in the splat texture. Texture splatting is introduced to have variance in texture, thereby ensuring the terrains appear natural. Unity3D has a built-in array - SplatPrototype[] that allows to store 2D texture. Unlike the previous implementation where only one texture (sand texture) was applied, giving the same pattern look throughout the entire stretch of terrain. It is possible to store multiple textures and for this part of the project, texture was generated using terrains normals and steepness. Using a 3D array of floats to store splat data, the x and y coordinates represent terrain coordinates and z coordinate stores the weight assigned to each texture. Figure 4.16 is a representation of the generated terrain with splat texture applied.

Unity does not align terrains the way it is imported or read from file. The orientation changes and has to be handled manually. It required shifting coordinates and redrawing the heights.

```
using (System.IO.FileStream f = System.IO.File.OpenRead(heighmapFile)) {
using (System.IO.BinaryReader br = new System.IO.BinaryReader(f)) {
for (int x=0; x<512; x++) {
for (int y=0; y<512; y++) {
heightData [x, y] = heights [br.ReadUInt16 ()];}}
for (int y=0; y <512; y++) {
        for (int x=0; x <512; x++) {
                rotHeightData [y, x] = heightData [511 - y,  x];}}
```

After the swap, terrains were rendered in the right orientation corresponding to real world.

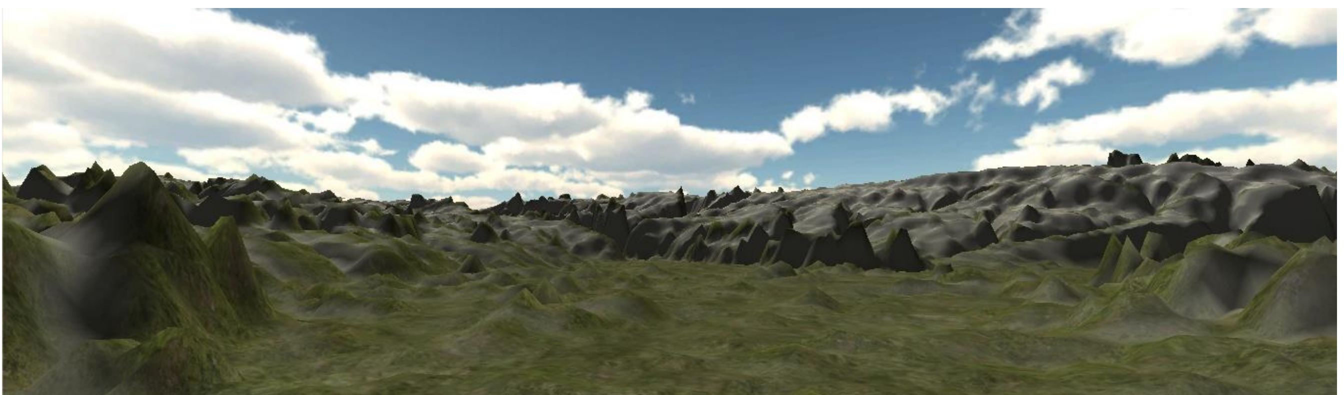


Figure 4.16 Terrain generated from RAW image and textured using splatting

During the test phase, loading of terrains blocked the Unity process until all terrains were loaded onscreen. This seemed to be a road block but Unity allows having multiple scenes within a project. Usage

of multiple scenes and setting priority levels to scenes as primary and secondary helped unlock the hindrance caused while loading terrains. However, this did not solve the problem with load time since terrains became visible only after the race was underway and yet again the resources were stored within the applications resource folder causing a memory issue. Further, this process is not a one step retrieval of data as it required sourcing of resources from LP DAAC and converting them to RAW prior to having them fed into unity.

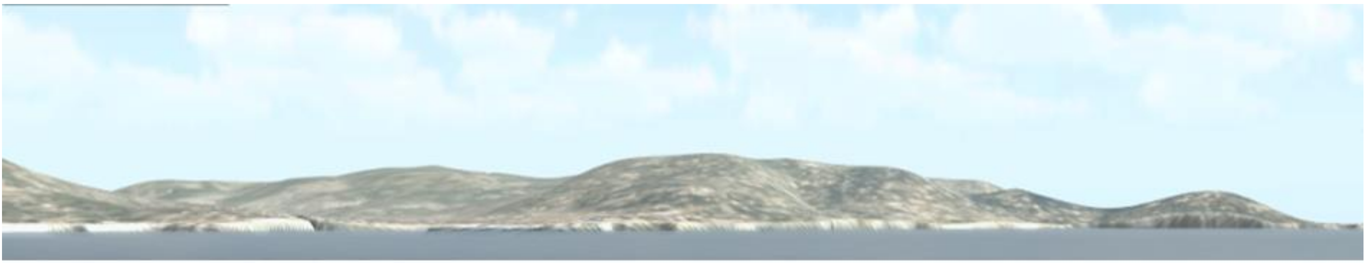
4.8.2.3 *Using data from Google maps*

The Google static map API allows for integration of maps and images and elevation data is retrieved from the service and this is processed on the server end to create an elevation grid of 128 by 128. It is faster to read 128*128 compared to a grid with higher resolution. The image is processed and stored as RAW. Unity requests data from the Sailracer server pertaining to a sailboats position and at the server backend, the data is processed, stored and simultaneously send to Unity for drawing terrain's heightmap.

Each terrain tile adapts to a scale in world space and for this project it is considered that each tile corresponds to 1km in world space. Hence the tiling system implemented, uses this scale to set its neighbours. By putting this approach in practice, the generation of terrain is fast.

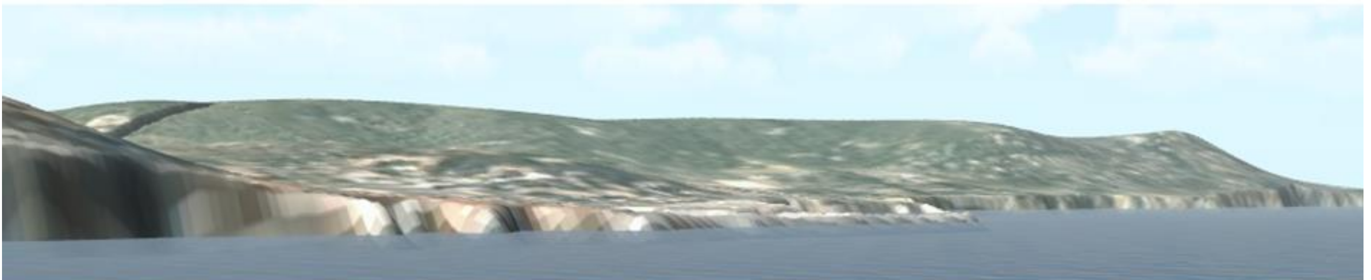
To make appearance look more natural, satellite images form Google are used as splat textures to cover the terrain surface. However, this again requires the mapping of coordinates of the terrain tile to match the coordinates of image tiles. The same technique is used to retrieve the image tiles since the server processes each image as chunks of 128*128 pixels and sets a naming convention similar to its respective terrain tile. The images are stored as PNG and placed as 2D texture within Unity and applied as a splat texture on the terrain. A view of the terrain is shown is Figure 4.17.

The data acquired using this method loaded terrains faster compared to the previous tests implemented. There is a small lag caused in loading terrains on race start but this is covered using a loading screen which will overcome the initial load. The subsequent loading of terrains occurred during the race is processed at the backend and is not very obvious to a viewer, thus making it an efficient technique of terrain generation. Furthermore, the memory handling is managed to make sure unused memory is released using the garbage collector - System.GC.Collect() and Resources.UnloadUnusedAssets () functions.



(a)

(b)



(c)

Figure 4.17 (a),(b) and (c) are Terrains generated after processing data received from Goole services and viewed at various zoom levels

4.9 Performance Test

4.9.1 Ocean Model

For the simulation to run smoothly, a frame rate above 50 is required. The table 4.1 below show how the frame rate responds to a change in the grid size .As expected, the relationship between the computation time and the grid size seems to be linear. The test is done with current model and Philips spectra model.

The performance has been tested on a 2 year old computer with Intel core i3 Cpu @ 1.90 Ghz . The tests have been done without displaying the boat and without simulating foam. The results are varying with a factor of 8 based on other programs running in Windows. To establish the reason for this variation, more testing is needed. 'Ocean Simulator' is optimized to display waves without effects like foam. Including foam therefore reduces the frame rate significantly. Including boat in addition results in a reduction of the FPS of approximately one half

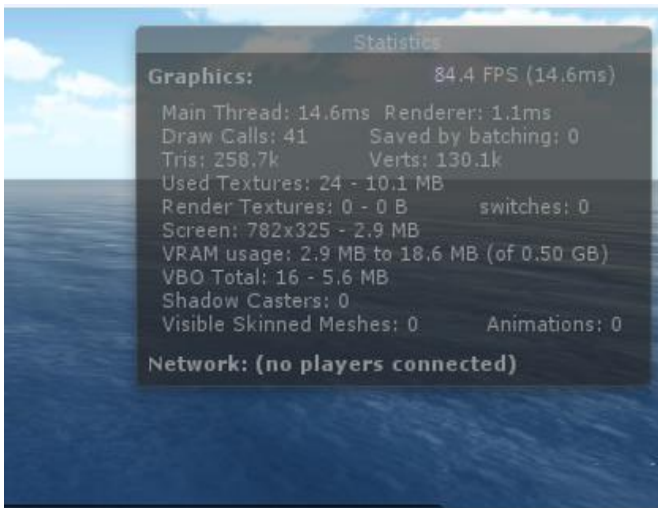
.

Table 4-2 FPS comparison with Current ocean model and Philips spectra

	Current Ocean Model	Philips Spectra
Grid Size	FPS	FPS
8	162.5	102.0
16	102.9	75.3
32	84.4	37.4
64	64.6	26.5
128	36.6	25.4
256	25	15.9
512	8.3	4.9

Figure 4.18 FPS of Current ocean model and Philips spectra in 8 * 8 Grid .

Figure 4.19 FPS of Current ocean model and Philips spectra in 16 * 16 Grid .



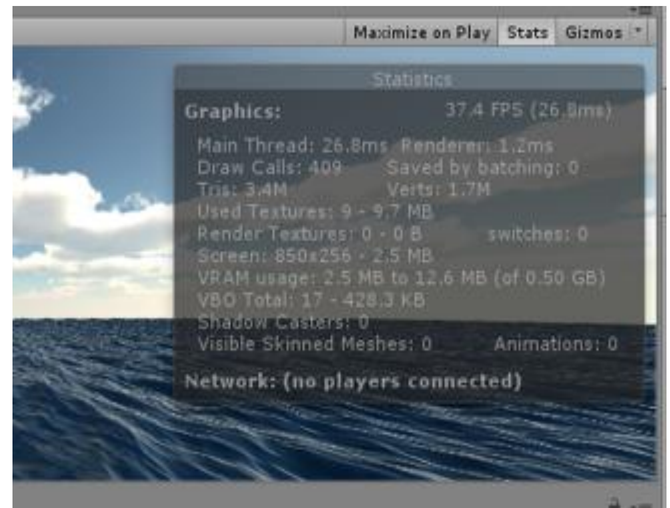


Figure 4.20 FPS of Current ocean model and Philips spectra in 32 * 32 Grid .

Figure 4.21 FPS of Current ocean model and Philips spectra in 64 * 64 Grid .

Figure 4.22 FPS of Current ocean model and Philips spectra in 128 * 128 Grid .

Figure 4.23 FPS of Current ocean model and Philips spectra in 256 * 256 Grid .

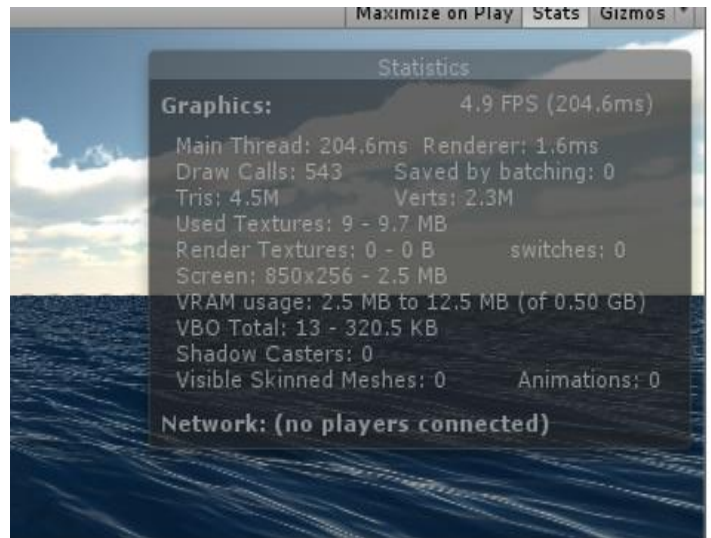


Figure 4.24 FPS of Current ocean model and Philips spectra in 512 * 512 Grid .

For grid size below 256, both CPU's run through the algorithm quite fast. The CPU time is not significant for the total performance of the program. But when the grid size exceeds 256, both processors use significantly longer time. Both processors have a L1 cache of 512 kB, which is the same size as a 256 by 256 matrix of complex floating point numbers. The limitation of L1 cache can therefore explain the jump in time consumption when grid size exceeds 256 by 256 elements. But for the purpose of Ocean Simulator, a grid size of 64 by 64 is enough. The IFFT algorithm is therefore not a limiting factor.

4.9.2 Terrain generation

The implementations used for terrain generation were carried out successfully to create shorelines on runtime but not all of them proved to be efficient. The terrains generated using TIFF as an input data, visibly showed poor load time but this needed test parameters to prove the same. Time, seemed like the parameter of choice since it was necessary to have a system that generated terrains faster.

Firstly, the terrain created using Google map's services was initialised to generate a 3*3 grid of 9 tiles, without any texture mapping or any other external game objects in the scene. Time was measured from starting of scene till rendering of the last terrain tile. The tests were run to capture 10 successive readings. The average load time taken for terrains to be generated using this approach was 4.1489 seconds.

Secondly, the RAW data from LP DAAC was tested for the same grid size of 3*3 and using the same terrain size, which was 1024 by 1024. The average load time for 9 terrains showed 4.9078 seconds. The last test of this series was performed on the data used from GIS-Lab. The TIFF had a poor average load time of 69.7731 seconds for the generation of 9 terrains in a 3*3 grid.

Figure 4.25 is a graphical representation of time consumed to load a set of 9 terrains using the terrain generation techniques implemented in this paper.

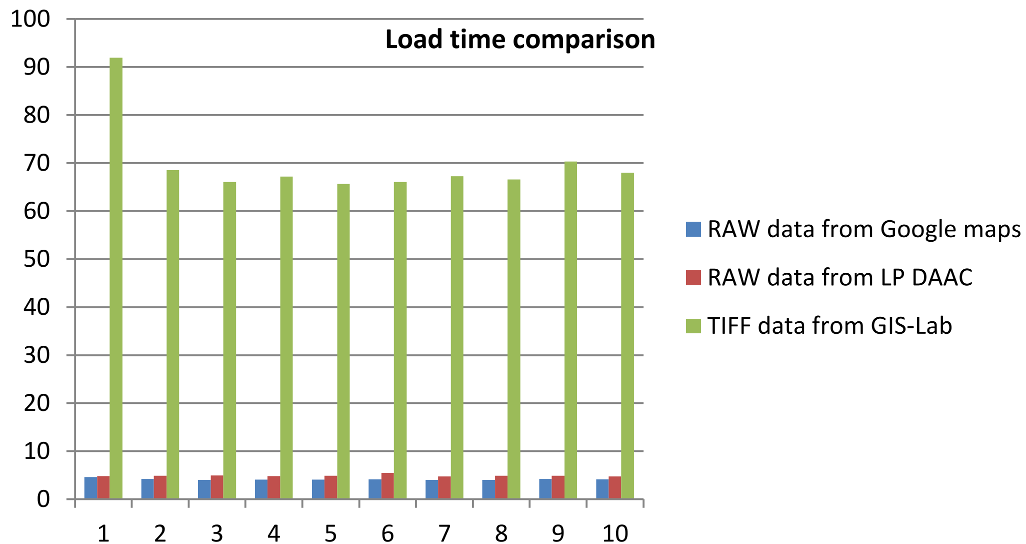


Figure 4.25 Comparison of load time using the implemented terrain generation techniques

During the process of optimization, it was found that load time can further be improved to render terrains faster. The test was carried to verify if the dataset type used to store data had an effect on the processing time.

Once again, time was the deciding factor for the analysis. Two dataset types were considered for the test. The first being character array and second one, byte array. In both cases, the tests involved generation of terrain with texture splatting. Each test type went through 11 successive runs under two different scenarios.

In the first scenario, terrain generation was limited to grid size of 3*3 whereas in the later scenario, a 5*5 grid was used. Figure 4.26 and Figure 4.27 are a time comparison set of 3*3 and 5*5 terrains generated using the dataset types Char[] and Byte[] respectively.

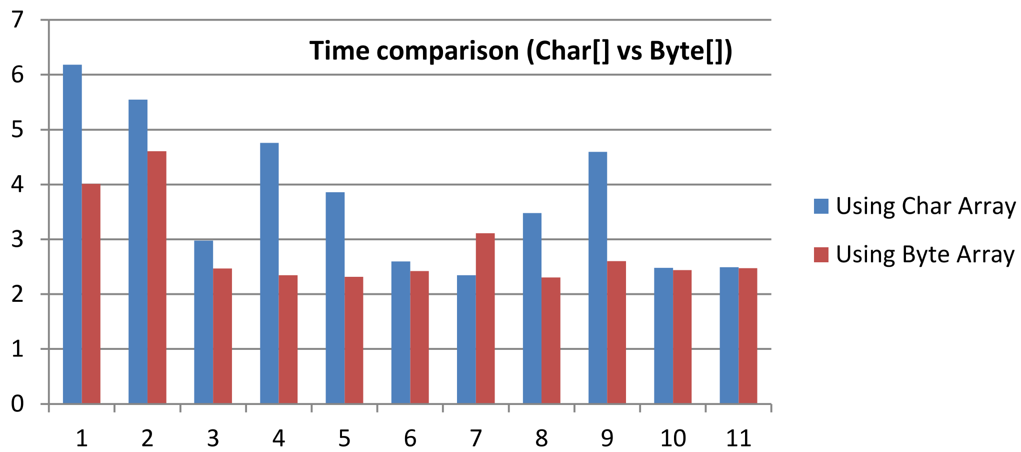


Figure 4.26 Time comparison of dataset types for generation of 9 terrains

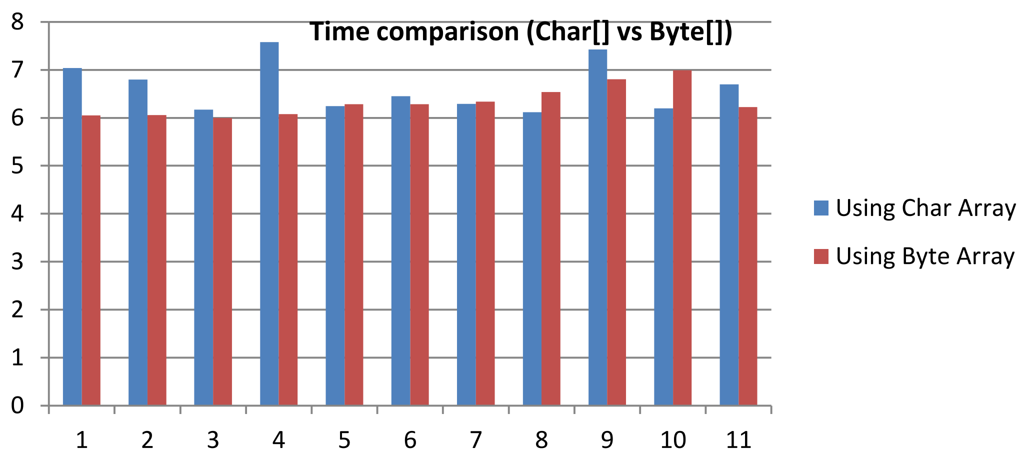


Figure 4.27 Time comparison of dataset types for generation of 25 terrains

4.9.3 3D Sail Racing

As seen in 4.9.1, for the simulation to run smoothly, a frame rate above 50 is required. In this section Testing and comparison of the current 3D sail racing model with different system is done. All the system used for this test are 2 year old computers. The FPS of the completed model with different system is given below.

Table 4-3 FPS comparison with Different System

Operating System	RAM	Processor	Graphics	FPS
Windows 8	2 GB	Core i3	Cpu @1.90 Ghz	60.2
Windows 8	4GB	Core i5	Cpu @ 2.40 Ghz	83.2
Windows 8	16GB	Core i7	Cpu @2.40 Ghz	108.0
Windows 8	4GB	AMD A10	HD Graphics @2.50 Ghz	79.3

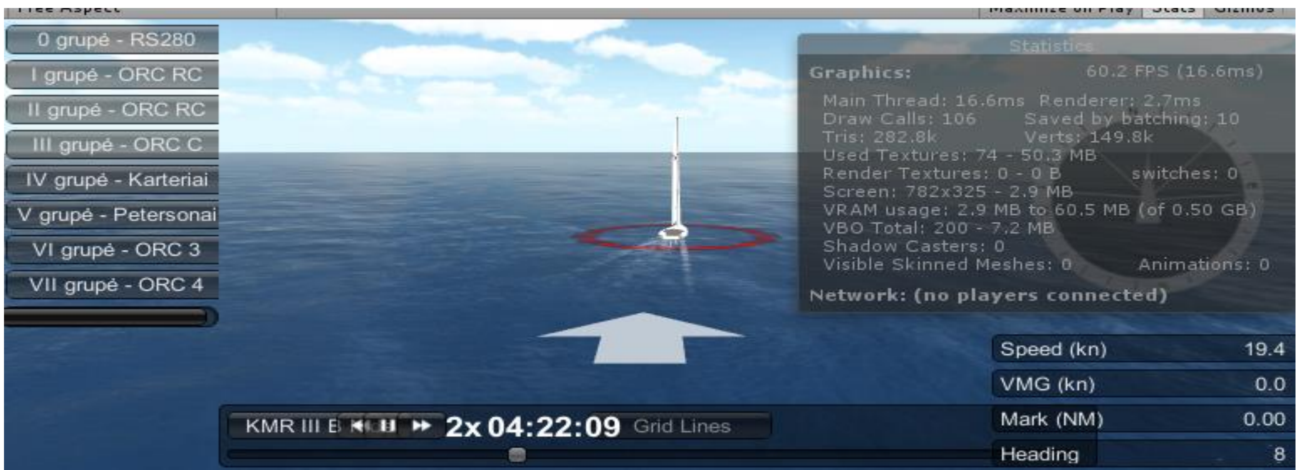


Figure 4.28 FPS from Core i3 processor

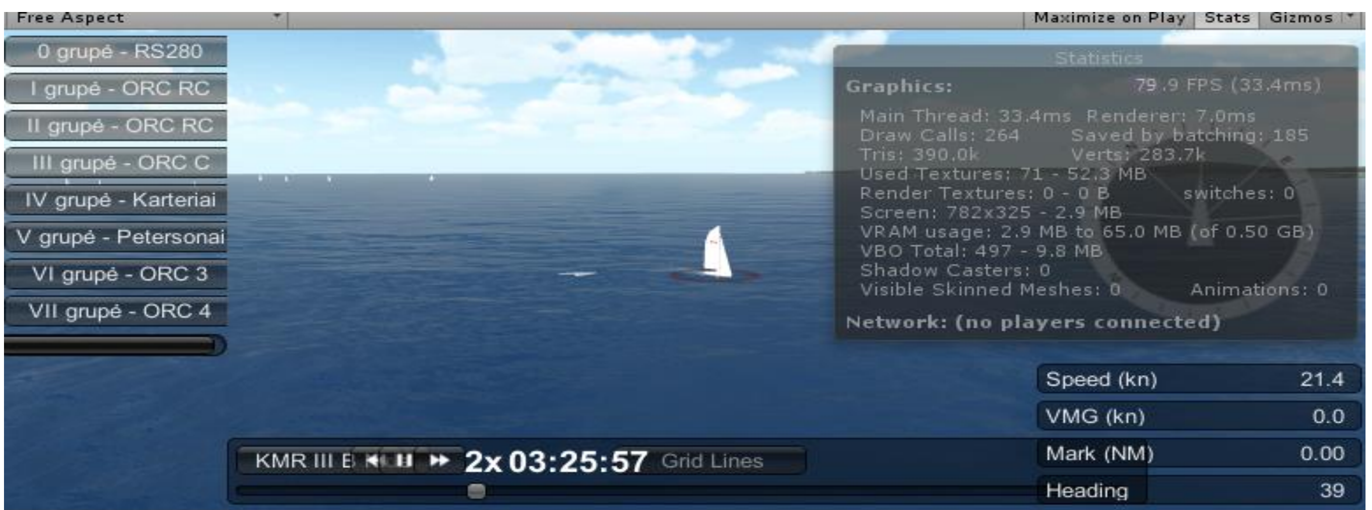


Figure 4.29 FPS from AMD A10 Processor

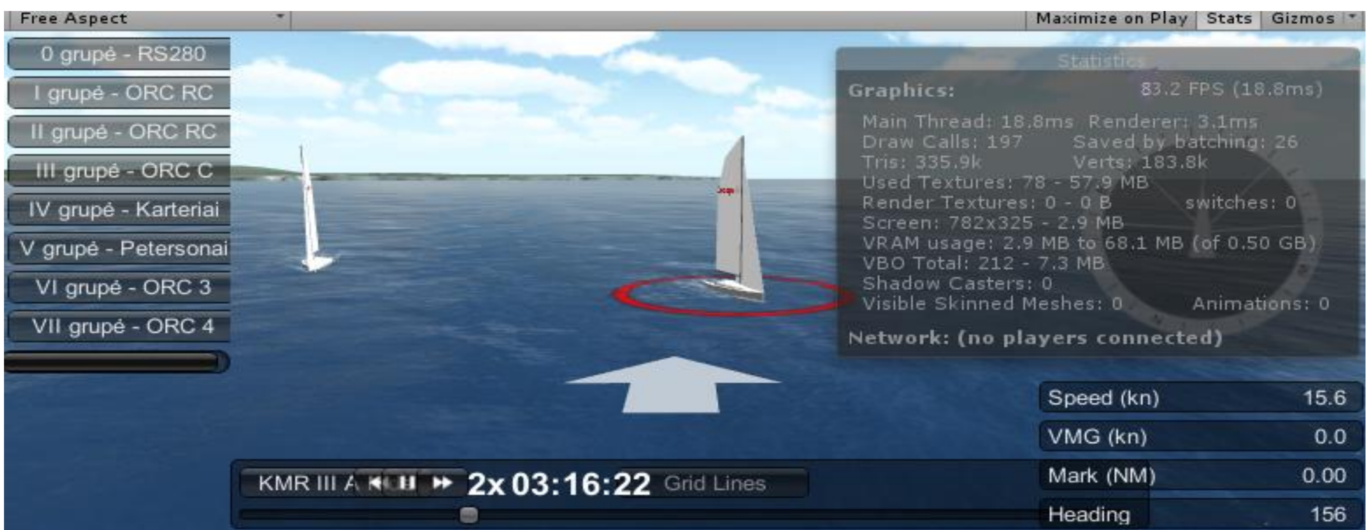


Figure 4.30 FPS from Core i5 processor

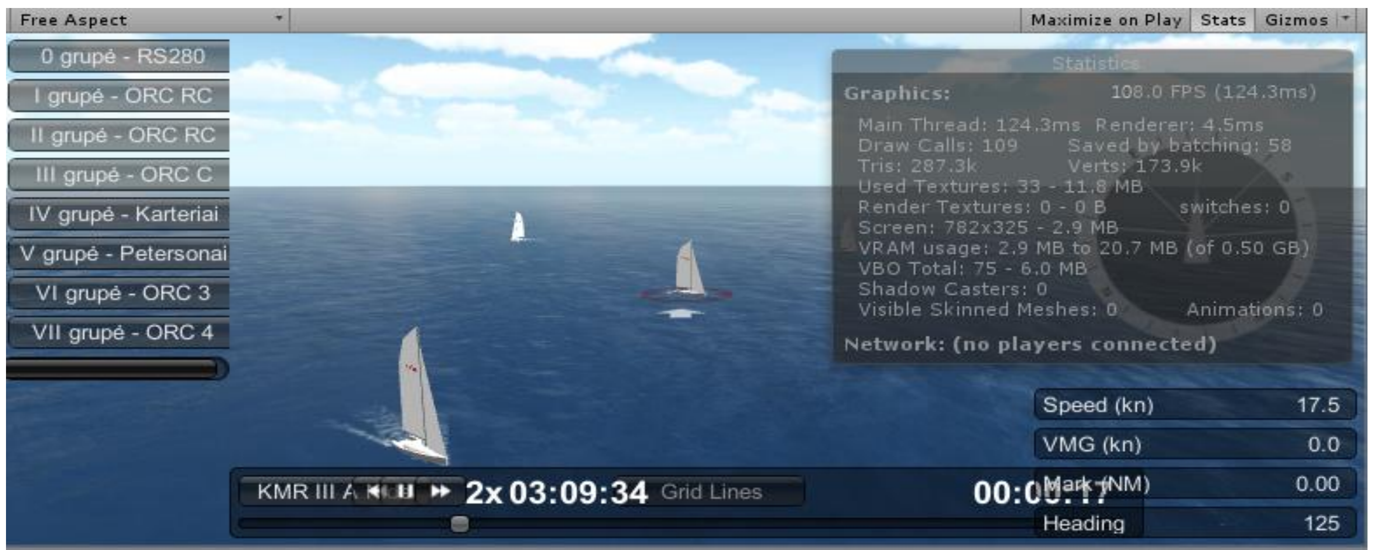


Figure 4.31 FPS from Core i7 processor

5 Future Scope

5.1 Web Player Streaming

Web Player Streaming is critical for providing a great web gaming experience for the end user. The idea behind web games is that the user can view your content almost immediately and start playing the game as soon as possible instead of making him wait for a progress bar. This is very achievable, and we will explain how.

5.2 Tuning for Portals

This section focuses on publishing for online game portals. Streaming is useful for all kinds of contents, and it can easily be applied to many other situations. Online game portals expect that some form of game play really starts after downloading at most 1 MB of data. If it is not reached it more likely doesn't allow portal to accept the content. From the user's perspective, the game needs to start quickly. Otherwise user time is being wasted and user might as well close the window. On a 128 kilobit cable connection user can download 16 KB per second or 1 MB per minute. This is the low end of bandwidth online portals target.

The game would optimally be set up to stream something like 50 KB to display the logo and menu (4 seconds),320 KB for the user to play a tiny tutorial level or to do some fun interaction in the menu (20 seconds),800 KB for the user to play the first small level (50 seconds) and Finish downloading the entire game within 1–5 MB (1–5 minutes).The key point to keep in mind is to think in wait times for a user on a slow connection and never let user wait.

5.3 Live Streaming

In order to see the live streaming of sail racing, user can request the video by clicking the option live streaming in web player. Web player will display a prompt window which gives three channels of Upstream .Then user can select one of the channels to view the live streaming in the prompt available on the web player. By adding this to the real time simulation project, it will state the accuracy of simulation with original game object. Video streaming is done by Unity GUI with the help of WWW class. The prototype is designed in same way of displaying three videos and when user click one of the channel ,the prompt occur on web player and the video will be played with the same process mentioned above. Figure 5.1 is an outline of the live streaming process.

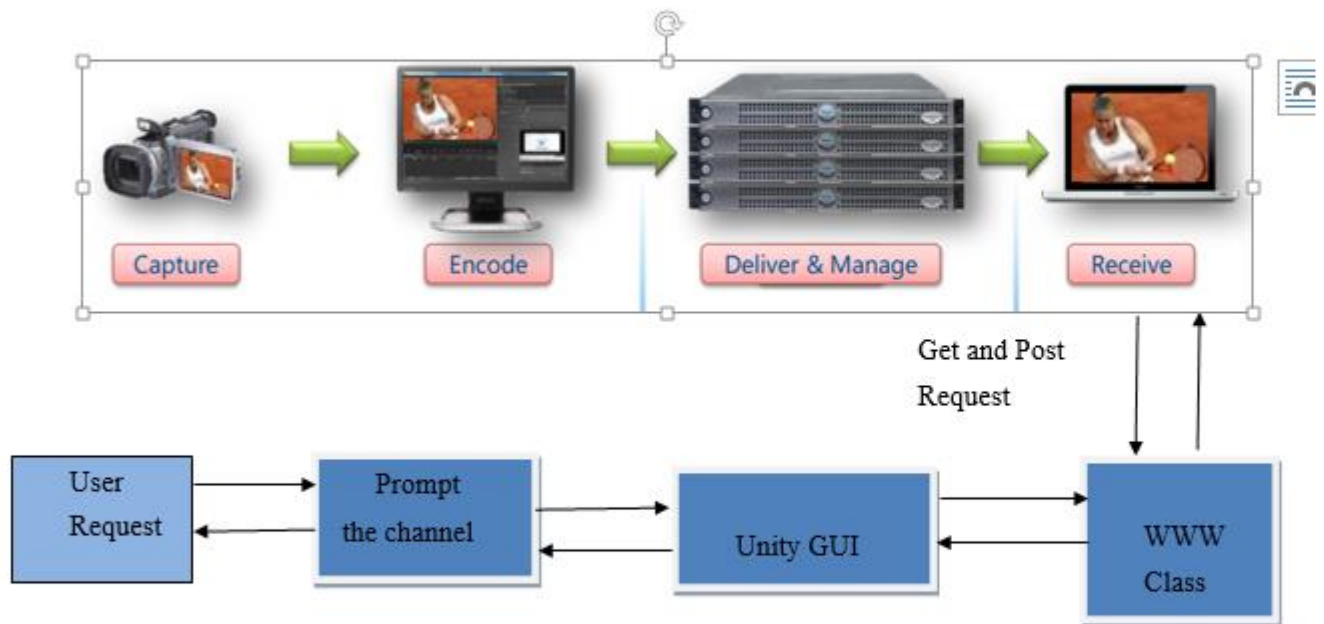


Figure 5.1 Live streaming Process

5.3.1 Video Streaming Prototype

A prototype of video streaming in web player is done by adding three videos url to WWW classes and then with the help of Unity GUI and Textures a prompt is created on click and it will give three options for the user and user can select one of the video among it and by clicking, it will play the video available in that URL.

5.3.1.1 *WWW class*

WWW class is used for Simple access to web pages. It is a small utility module for retrieving the contents of URLs. User start a download in the background by calling WWW (url) which returns a new WWWobject.

User can inspect the property to see if the download has completed or yield the download object to automatically wait until it is (without blocking the rest of the game).It can be used to get some data from a web server for integration with a game such as high score lists or calling home for some reason. There is also functionality to create textures from images downloaded from the web and to stream & load new web player data files. The WWW class can be used to send both GET and POST requests to the server. The WWW class will use GET by default and POST if user supply a postData parameter. URLs passed to WWW class must be '%' escaped. The code below shows the WWWClass implemented in prototype.

```
function Start () {
        var www1 = new WWW(video1);
        var www2 = new WWW(video2);
        var www3 = new WWW(video3);

        movieTexture1 = www1.movie;
        movieTexture2 = www2.movie;
```

```
movieTexture3 = www3.movie;
```

5.3.1.2 *GUI Texture*

GUI texture is optimised using import setting. After externalizing music, textures easily take up 90% of the game. Typical texture sizes are too big for web deployment. In a small browser window, sometimes big textures don't even increase the visual fidelity. Halving the texture resolution actually makes the texture size a quarter of what it was. Generally the size of the web player will be reduced. Unity uses cutting edge LZMA-based compression which usually compresses game data to anywhere from one half to a third of the uncompressed size.

```
@script RequireComponent (GUITexture)
@script RequireComponent (AudioSource)
```

In the below code switching of video as per requirement of user is given

```
function switchVideo (moveTex : MovieTexture) {
        var gt = GetComponent.<GUITexture>();
        gt.texture = moveTex;

        transform.localScale = Vector3 (0,0,0);
        transform.position = Vector3 (0.5,0.5,0);
        gt.pixelInset.xMin = -moveTex.width / 2;
        gt.pixelInset.xMax = moveTex.width / 2;
        gt.pixelInset.yMin = -moveTex.height / 2;
        gt.pixelInset.yMax = moveTex.height / 2;

        aud = GetComponent.<AudioSource>();
        aud.clip = moveTex.audioClip;
```

5.4 Publishing Streaming Web Players

Streaming in Unity is level based, and there is an easy workflow to set this up. Internally, Unity does all the work of tracking assets and organizing them in the compressed data files optimally, ordering it by the first scene that uses them. In order to use streaming in Unity, select the Web Player Streamed in the Build Settings. Then the content automatically starts as soon as all assets used by the first level are loaded. It is good to keep the "menu level" to something like 50–100 KB. The stream continues to load as fast as it can, and meanwhile live decompresses.

6 Conclusion

- Though GPS data is readily available and easy to track, it is important to have them processed efficiently to meet the requirements pertaining to an event of interest. This paper makes use of GIS data used in sailboats to calculate the estimated path taken by a skipper and translate it to cartesian coordinates for Unity3D to immediately process them, saving it from calculation overhead and thereby use estimated path to determine the point of crossing and rounding at any given time. Using the physics of sailing, the headings and wind direction intercepted by the sailboat during the race event are used to animate the boats component such as sail sheets and heel.
- The implementation of realistic looking ocean simulator with a sail boat floating on the surface is done in user interactive environment. Foam, and wake is included to add to the realism. The application is designed to run on a normally equipped laptop with Windows as the operation system without compromising the frame rate. All calculations except the wake generation are performed in real time. The waves are represented with the JONSWAP spectrum and ocean elevation is calculated using fast fourier transformation algorithm. The code is optimized with respect to common computer architecture to provide realistic visualization at a high frame rate. To get the required visual effects, shaders are used and cubemapping is done for lighting. Current ocean model is tested with existing model for performance.
- The dynamic generation of shorelines are procedurally generated using the elevation data from global data centers, which provide data in various formats. By testing the two data formats – RAW and TIFF, it is obvious that iterating through a character set represented in RAW was faster to read compared to the character set in TIFF within Unity3D. TIFF are encoded as strips of data and since Unity does not have a native TIFF reader, it gets time consuming to loop through each strip and store them to a buffer before they are redrawn to the terrain mesh.
- The built-in data types in Unity are quicker to access since they have a pre-defined data structure, which means it is easy to reference and loop through these datatypes. While storing RAW data to built-in data types of Char[] and Byte[], the process fairly generated terrains in a similar time span but on further analysis, it is noted that storing a raw values to byte[] rendered terrains slightly faster to processing them over Char[]. Byte[] gains the higher position since the Raw data requested by the WWW class in unity has the option of retrieving data in a couple of formats and one of them being Byte, thus making it readily available for the byte array to process faster.
- The methodologies and implementations used in developing the 3D visualization system is viable for real-time use in wide number of area.
    - Applications using GPS for tracking can utilize the same coordinates to automate tracking systems by setting waypoints along its path and with more efficient time discretion process, making it is possible to attain data responding to sensitive changes made in position.
    - Real-time strategy games involving water bodies or terrain such as sailing, rowing, windsurfing, wargame..etc
    - Simulation of real world using data corresponding to Streets, roads, walkways.. etc from maps and superimposing 3D models to represent objects in real-time using a path-find algorithm.

References

1. http://svgeurope.org/blog/headlines/trimaran-ceo-olivier-emery-on-current-applications-future-opportunities-for-the-georacing-virtual-tracking-system/
2. http://virtualeye.tv/index.php/the-sports/virtual-eye-sailing
3. http://pressreleaseping.com/enhance-sailboat-racing-performance-raceqs-sailing-system
4. Stan Melax, A Simple, Fast, and Effective Polygon Reduction Algorithm, November 98 Game Developer Magazine
5. Hugues Hoppe Frank Losasso. Geometry clipmaps: Terrain rendering using nested regular grids. ACM Trans. Graphics (SIGGRAPH), 23(3), 2004
6. Filip Strugar. Continuous Distance-Dependent Level of Detail for Rendering Heightmaps(CDLOD).
7. Luebke, D. *et al.* Level of detail for 3D graphics. Morgan Kaufmann Pub, morgankaufmann publishers ed, 2003.
8. Jordan, L., The Basics of HTTP Live Streaming . *Larry's Blog*. Larry Jordan & Associates. Retrieved 18 June 2013.
9. Sayma, A. (2009). *Computational fluid dynamics*. Bookboon
10. Fournier, A., & Reeves, W. T. (1986, August). A simple model of ocean waves. In *ACM Siggraph Computer Graphics* (Vol. 20, No. 4, pp. 75-84). ACM.
11. Mitchell, J. L. (2005). Real-time synthesis and rendering of ocean water. *ATI ResearchTechnical Report*, 121-126.
12. Le Touzé, D., Bonnefoy, F., & Ferrant, P. (2002, January). Second-order spectral simulation of directional wave generation and propagation in a 3d tank. In *The Twelfth International Offshore and Polar Engineering Conference*. International Society of Offshore and Polar Engineers.
13. Bruneton, E., Neyret, F., & Holzschuch, N. (2010, May). Real-time Realistic Ocean Lighting using Seamless Transitions from Geometry to BRDF. In*Computer Graphics Forum* (Vol. 29, No. 2, pp. 487-496). Blackwell Publishing Ltd.
14. Tessendorf, J. (2001). Simulating ocean water. *Simulating Nature: Realistic and Interactive Techniques. SIGGRAPH*, *1*(2), 5.
15. Tuck, E. O., Scullen, D. C., & Lazauskas, L. (1999). *Sea wave pattern evaluation*. Technical report, The University of Adelaide, 2007
16. Faltinsen, O. M. (2005). *Hydrodynamics of high-speed marine vehicles*. Cambridge university press.
17. Kryachko, Y. (2005). Using vertex texture displacement for realistic water rendering. *GPU Gems*, *2*, 283-294.
18. www.keithLantz.net
19. Duchaineau, M., Wolinsky, M., Sigeti, D., Millery, M., Aldrich, C., Mineev-Weinstein, M.: ROAMing terrain: Real-time optimally adapting meshes. In: VIS'97: Proceedings of the 8th conference on Visualization '97, Los Alamitos, CA,USA, IEEE Computer Society Press (1997) 81–88
20. Olsen, J., Real-time procedural terrain generation, *Real-time synthesis of eroded fractal terrain for use in computer games*. Department of Mathematics And ComputerScience (IMADA), 2004.
21. A. Fournier, D. Fussell, and L. Carpenter, "Computer rendering ofstochastic models," Communications of the ACM, vol. 25, no. 6, pp.371–384, 1982.
22. D. Ashlock, S. Gent, and K. Bryden, Evolution of l-systems forcompact virtual landscape generation, *Evolutionary Computation*,2005, The 2005 IEEE Congress on, vol. 3. IEEE, 2005, pp. 2760–2767.
23. P. Walsh and P. Gade, Terrain generation using an Interactive GeneticAlgorithm, *Evolutionary Computation* (CEC), 2010, The IEEE Congress on. IEEE, 2010, pp. 1–7.
24. Nullpinter; http://www.nullpointer.co.uk/content/how-to-make-an-infinite-world
25. Alastair Aitchison;https://alastaira.wordpress.com/2010/12/17/examining-3d-terrain-of-bing-maps-tiles-with-sql-server-2008-and-wpf-part-1/
26. Bing Maps Tile System; https://msdn.microsoft.com/en-us/library/bb259689.aspx/

27. K. Stolze, The Standard to Manage Spatial Data in Relational Database Systems, *Database and Information Systems Group*, BTW 2003, Leipzig, Feb 2003
28. Unity Manual., http://docs.unity3d.com/Manual/index.html/
29. B. Fraser., Understanding Digital Raw Capture, Adobe Systems Incorporated , Whitepaper, 2004.
30. Le Roux, J. P. (2008). An extension of the Airy theory for linear waves into shallow water. *Coastal Engineering*, *55*(4), 295-301
31. Heath, J. P., & Peachey, L. D. (1989). Morphology of fibroblasts in collagen gels: a study using 400 keV electron microscopy and computer graphics. *Cell motility and the cytoskeleton*, *14*(3), 382-392
32. Wess, J., & Zumino, B. (1974). A Lagrangian model invariant under supergauge transformations. *Physics Letters B*, *49*(1), 52-54
33. Hinsinger, D., Neyret, F., & Cani, M. P. (2002, July). Interactive animation of ocean waves. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (pp. 161-166). ACM.
34. Mastin, G. A., Watterberg, P. A., & Mareda, J. F. (1987). Fourier synthesis of ocean scenes. *Computer Graphics and Applications, IEEE*, *7*(3), 16-23.
35. Premože, S., & Ashikhmin, M. (2001, December). Rendering natural waters. In*Computer graphics forum* (Vol. 20, No. 4, pp. 189-200). Blackwell Publishers Ltd.)
36. Huang, N. E., Shen, Z., & Long, S. R. (1999). A new view of nonlinear water waves: The Hilbert Spectrum 1. *Annual review of fluid mechanics*, *31*(1), 417-457.
37. McClain, C. R., Chen, D. T., & Hart, W. D. (1982). On the use of laser profilometry for ocean wave studies. *Journal of Geophysical Research: Oceans (1978–2012)*, *87*(C12), 9509-9515.
38. Sobieski, P., Guissard, A., Baufays, C., & Siraut, P. (1993). Sea surface scattering calculations in maritime satellite communications. *Communications, IEEE Transactions on*, *41*(10), 1525-1533
39. Elfouhaily, T., Chapron, B., Katsaros, K., & Vandemark, D. (1997). A unified directional spectrum for long and short wind-driven waves. *Journal of Geophysical Research: Oceans (1978–2012)*, *102*(C7), 15781-15796.
40. Jeschke, S., Birkholz, H., & Schmann, H. (2003). A procedural model for interactive animation of breaking ocean waves.
41. R. Baxter., https://sites.google.com/site/raxterbaxter/projects/current-projects/streaming-osm-terrain-unity-package

Appendix

This part of document contains additional information not included into main document.