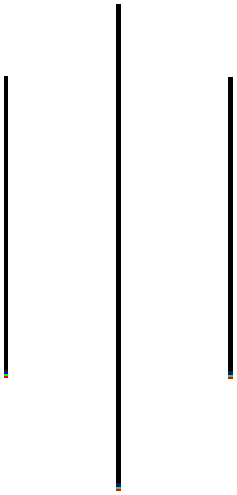
TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS



A project report on: Computer Graphics
(Island Simulation)

Submitted by:		Submitted to:
Anisha Adhikari	073/BEX/450	Department of Electronics
Pravesh Gaire	073/BEX/427	and
Ram Sharan Rimal	073/BEX/431	Computer Engineering
Sampada Dhakal	073/BEX/437	

ACKNOWLEDGMENT

It is a genuine pleasure to express our deep thanks and gratitude to Electronics and Computer Department which granted us this platform to make a graphics project. Our appreciation to Mr. BJ sir, our graphics teacher whose contribution in stimulating suggestions and encouragement helped us a lot for initiating this project. His dedication and keen interest to help his students has been solely responsible for this work. We are thankful to Mr. Suresh sir, our instructor, who inspires us constantly for making a project. His scholarly advice and timely approach have helped us a great deal to begin this task. Furthermore we would also like to acknowledge with much appreciation the crucial role of the library of Pulchowk Campus, which provides us the permission to use all required equipment and the necessary materials. Pulchowk Campus family is to be thanked profusely for providing us a platform to grow and ultimately develop as an educated citizen. We are extremely thankful to our friends and family for becoming a source of constant encouragement and invaluable assistance for this process.

ABSTRACT

Computer and video game popularity has increased steadily since long. Developers and designers are challenged to come up with innovative ideas to gain their share of the ever increasing demand. Virtual reality visors, touch suits, and, games that respond to the movement of a game controller will become commonplace. Games with improved artificial intelligence will allow players the flexibility to change scenarios and storylines. Increasing demand will result in advancements in gaming technology that will give the gamer the ultimate gaming experience.

The scope of game development is very big. Our project “Island Simulation” is just an initiation to game development. We have used C++ as our programming language. To interact with GPU OpenGL is used. Various functions from OpenGL has assisted us in many ways to implement graphics functions such as clipping ,loading from frame buffer, sending datas to shaders, etc.

The main idea behind our project is to create an infinite landscape for which we have used Procedural Terrain Generation using perlin noise. We have implemented various illumination models for lighting. Z buffer algorithm is used for implementing shadow effects. Water simulation is done using mirrored environment and normal mapping.

As this project is just the beginning, it can be further enhanced in many ways to make it look more realistic and more user friendly.

Table of Contents

Table of Figures

1. INTRODUCTION

1.1. Background

Any image taken by a camera or that displayed in the screen is 2D. To add more realistic features to any image, graphics can be used. For this, the programming should be done in Graphics Processing Unit. Various algorithms and various concepts have been used in this project to create a 3D image.

1.2. Objective

The main objective of this project is to create an infinite terrain and make it user friendly by the use of keyboard shortcuts and mouse.

1.3. Problem Overview

As the main goal of our project, we have to create a landscape. There are various methods to accomplish this goal. In our very project, we have used Perlin noise algorithm for the generation of noise which generates a smooth random number than random number generation which has no relation to the other number besides it. The main advantage of using perlin noise is its relation to every other points around it so that the generated image looks more organic.

After the generation of perlin noise, different colors have to be allocated for different pixels so that the generated image looks like a landscape. Heightmap is used to give different heights to different pixels.

2. THEORETICAL BACKGROUND

2.1. OpenGL

OpenGL is merely a specification. It is developed by Khronos Group. It is also considered an API (Application Programming Interface) which is used to interact with Graphical Processing Unit. It provides us with large set of functions which is used to manipulate graphics and images.

The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform. It is then up to the developers implementing this specification to come up with a solution of how this function should operate.

When developing in immediate mode, in old days, most of the functionality was hidden in the library and developers didn't have much freedom at how OpenGl does its calculations which was an easy-to-use method for drawing graphics.

When developing in core-profile which is a modern practice that is very flexible and efficient but a bit difficult to learn than the immediate mode.

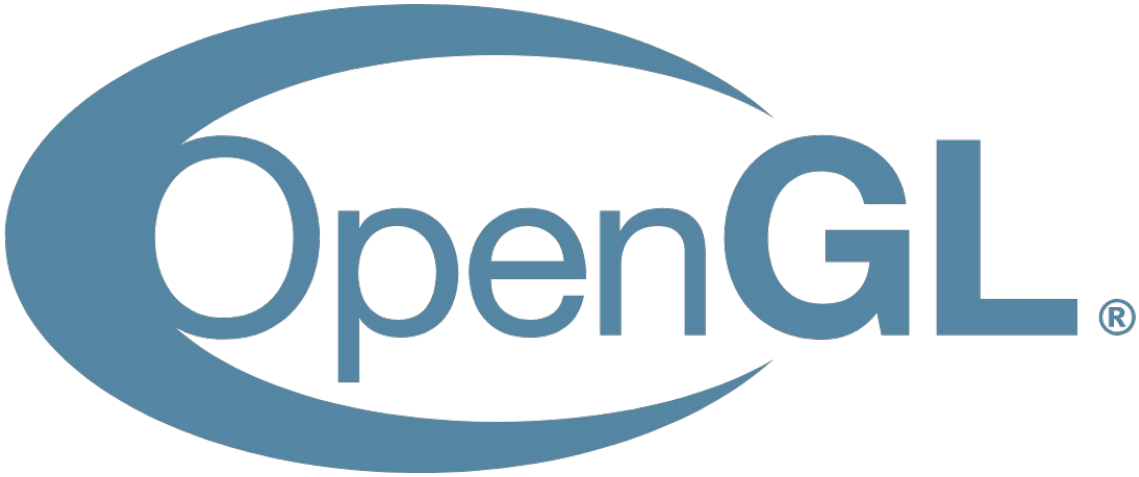


Figure 1.OpenGl Logo

2.1.1. Creating a window:

To create stunning graphics we need to create an OpenGL context and an application window to draw in. OpenGL purposely tries to abstract from these operations as those operations are specific per operating system. This means we have to create a window, define a context and handle user input all by ourselves. Here we will be using GLFW libraries that already provide the functionality we seek.

2.1.2. GLFW:

It is a type of library that is specifically targeted to OpenGl which allows us to create an OpenGL context, define window parameters and handle user input which is all that we need. It is written in C.

2.1.3. GLEW:

It stands for OpenGL Extension Wrangler Library. It is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. It provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

2.1.4. Cmake:

A type of tool that can generate project files of user's choice from a collection of source code files using pre-defined CMake scripts. We will be using Visual Studio. We need to install and set the source and destination folders then generate the build folder.

2.1.5. Rendering:

Rendering is the process of add shading, color and lamination to a 2-D or 3-D wireframe to create life-like images on a screen. According to the time of rendering, we can render in two ways called pre-rendering and real-time rendering. Real-time rendering is often used for 3-D video games, which require a high level of interactivity with the player. Pre-rendering is a CPU-intensive but can be used to create more realistic images, is typically used for movie creation. Shaders are also very isolated programs in that they're not allowed to communicate with each other. The only communication they have is via their inputs and outputs.

2.1.5.1. Rendering pipeline

Rendering pipeline also called graphics pipeline is a conceptual model that describes what steps a graphics system needs to perform to render a 3D scene to a 2D screen. It is process of turning that 3D model into what the computer displays. The model of the graphics pipeline is usually used in real-time rendering. Often, most of the pipeline steps are implemented in hardware, which allows for special optimizations. Graphics pipeline can be divided into three partsi.e application, geometry and rasterization.

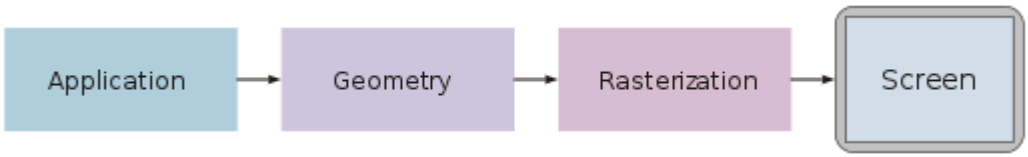


Figure 2Rendering Pipeline

2.1.6. Vertex input:

We have to first give OpenGl some input vertex data in order to draw something. OpenGL is a 3D graphics library so all coordinates that we specify in OpenGL are in 3D (x, y and z coordinate)OpenGL doesn't simply transform all your 3D coordinates to 2D pixels on your screen; OpenGL only processes 3D coordinates when they're in a specific range between -1.0 and 1.0 on all 3 axes (x, y and z).

2.1.8. Shader:

A shader is a type of computer program that was originally used for shading. Shaders are also very isolated programs in that they're not allowed to communicate with each other. The only communication they have is via their inputs and outputs. There are two types of shaders used and they are:

1. Fragment shader

2. Vertex shader

2.1.8.1. Fragment shader:

A fragment shader is a piece of code that is executed once, and only once, per fragment which are responsible for painting each primitive's area.The minimum task for this shader is to output an RGBA color.

2.1.8.2 Vertex shader:

Vertex shader provides the vertex positions to clip coordinates, to take us to the next stage. They are oriented to the scene geometry and can do things like adding cartoony silhouette edges to objects, etc.

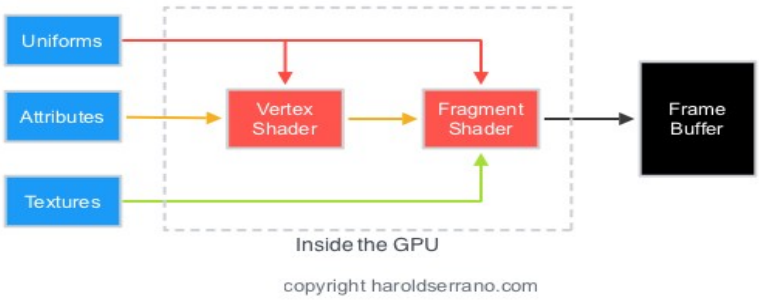


Figure 3 Vertex and Frame Shader

2.1.9. Textures:

Textures are typically used for images to decorate 3D models, but in reality they can be used to store many different kinds of data. It is used to add detail to an object. Aside from images, textures can also be used to store a large collection of data to send to the shaders.

Figure 4Sand Texture

2.2. Camera/View space:

When we're talking about camera we're talking about all the vertex coordinates as seen from the camera's perspective as the origin of the scene: the view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction. To define a camera we need its position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera. We're actually going to create a coordinate system with 3 perpendicular unit axes with the camera's position as the origin. The basic things to define the camera are as en-listed below:

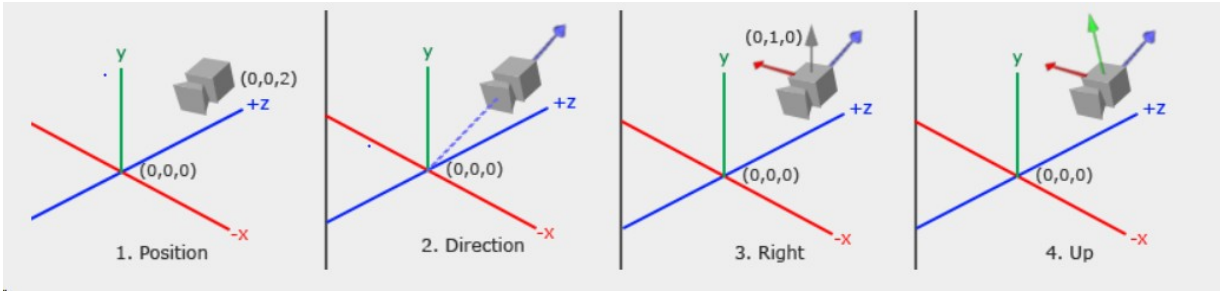


Figure 5Camera parameters

2.2.1. Camera Position:

The camera position is basically a vector in world space that points to the camera's position.

2.2.2. Camera Direction:

The next vector required is the vector containing the information about the direction in which the camera is pointing at. If we define a vector which contains the points where we want to look at,then the difference of the camera position with the look-at vector gives the direction of the camera. Since in the left handed system the direction in which camera looks is negative direction. So this vector also defines our view coordinate system z-axis.

2.2.3. Right Axis:

The next vector to define the camera information is the right vector which represents the x-axis of the camera space. To calculate the right axis we take the help of the up-vector (a vector that points towards up in world space).Then a cross product between the up vector and the direction vector gives the right axis vector.

2.2.4. Up Axis:

Now after getting the x-axis and the z-axis vector the vector pointing in the up direction (Up-Axis) is calculated from the cross product of the two vectors. It defines the y-axis of the camera/viewspace or our View Coordinate System.

2.2.5. Look-At Matrix:

The look-at matrix creates a view matrix that looks at a given target. The look-at matrix contains three perpendicular axis and the position vector to define the camera/viewspace. The above look-at matrices contains **R** (which is the right vector),**U** (up-vector),**D** (the direction vector) and **P** (the position vector).So a look-at matrix defines the complete coordinate system for the camera in the Opengl.While using this in opengl we only have to specify the a camera position, a target position and a vector that represents the up vector in world space. GLM then creates the LookAt matrix that we can use as our

view matrix.

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 6 Look-At MAtrix

2.3. Lighting:

Lighting in the real world is extremely complicated and depends on way too many factors, something we can't afford to calculate on the limited processing power we have. Lighting in OpenGL is therefore based on approximations of reality using simplified models that are much easier to process and look relatively similar. These lighting models are based on the physics of light as we understand it.

2.3.1. Diffuse Lighting:

Ambient lighting by itself does not produce the most interesting results, but diffuse lighting will start to give a significant visual impact on the object. Diffused light is a soft light with neither the intensity nor the glare of direct light. It is scattered and comes from all directions. Thus, it seems to wrap around objects. It is softer and does not cast harsh shadows. Diffuse lighting gives the object more brightness the closer its fragments are aligned to the light rays from a light source.

The sun is the only source of light present in the simulation. So every illumination is dependent to it. The sun acts as a diffused lighting source. The working of this method is described below in brief.

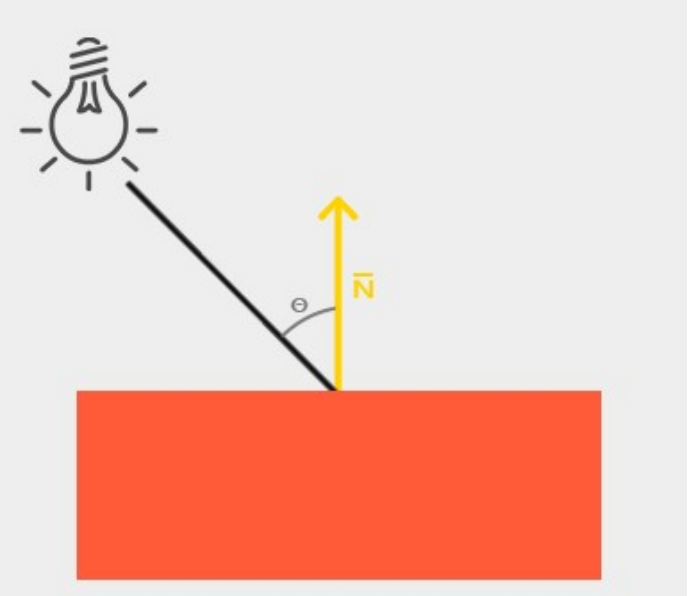


Figure 7Reflection

To the left we find a light source with a light ray targeted at a single fragment of our object. We then need to measure at what angle the light ray touches the fragment. If the light ray is perpendicular to the obect's surface the light has the greatest impact. To measure the angle between the light ray and the fragment we use something called a normal vector that is a vector perpendicular to fragment's surface. The angle between two vectors can be easily calculated using dot product. Next we want to calculate the actual diffuse impact the light has on the current fragment by taking the dot product of the norm and lightDir vector. The resulting value is then multiplied with the light's color to get the diffuse component, resulting in a darker diffuse component the greater the angle is between both vectors:

2.3.2. Atmospheric Light Scattering:

A simple way to apply a sky color to out skybox is to just apply some tint color to our box plane for a nice sunny and clear sky day. The atmospheric light scattering occurs when the sunlight passes down through the different kinds of gases in the different layers of the atmosphere, which cause the reflected sunlight to reflect different colors.

2.3.2.1. Rayleigh and Mie Scattering:

The most two forms of scattering in the atmosphere are Rayleigh and Mie scattering. Rayleigh scattering is caused by small molecules in air and scatters light more heavily at shorter wavelengths. The sky is blue because the blue light bounces all over the place, and ultimately reaches our eyes from every direction. The sun's light turns yellow/orange/red at sunset because as light travels far through the atmosphere, almost all of the blue and much of the green light is scattered away before it reaches us, leaving just the reddish colors. Mie scattering is caused by larger molecules in air called aerosols such as dust and pollution and it tends to scatter all wavelengths of the light equally.

2.3.3. Shadows:

Shadows are a result of the absence of light due to the occlusion; when a light source's light rays do not hit an object because it gets occluded by some other object the object is in shadow. Shadows add a great deal of realism to the lighted scene and make it easier for a

viewer to observe spatial relationship between objects. They give a greater sense of depth to the scene.

The idea behind shadow mapping is quite simple: we render the scene from the light's point of view and everything we see from light's perspective is lit and everything we can't see must be in shadow.

In the figure below, all the blue lines represent the fragments that the light source can see. The occluded fragments are shown as black lines: these are rendered as being shadowed. If we were to draw a line or ray from the light source to a fragment on the right-most box we can see the ray first hits the floating container before hitting the right-most container. As a result, the floating container's fragment is lit and the right-most container's fragment is not lit and thus in shadow.

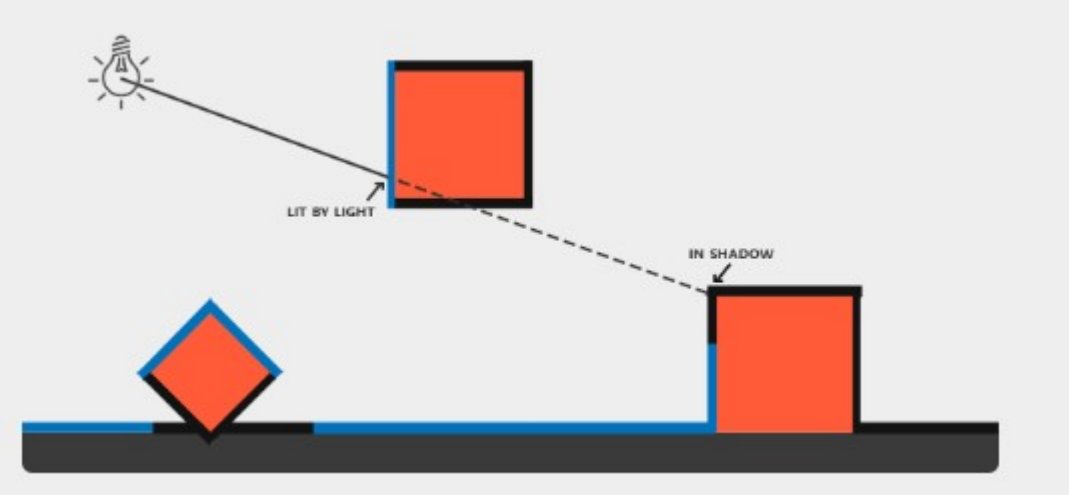


Figure 8Shadow Formation

We render the scene from light's perspective and store the resulting depth values in a texture. This way we can sample the closest depth values as seen from light's perspective. After all depth values show the first fragment visible from light's perspective. We store all these depth values in a texture that we call a depth map or shadow map.

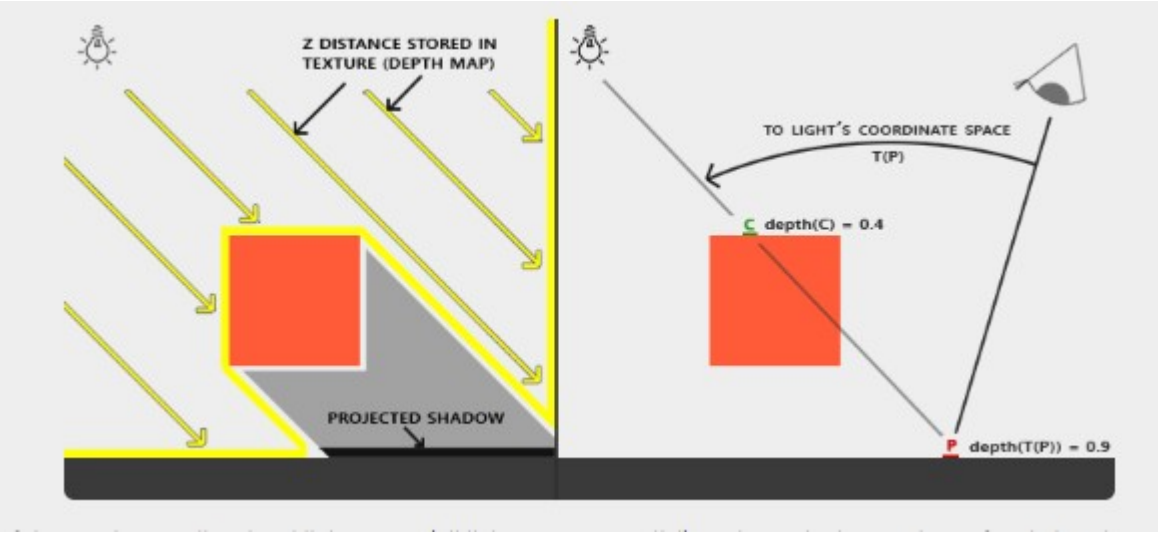


Figure 9Depth map

2.4. Terrain Geometry:

The data that makes up a height-field forms a regular grid of height samples. To render a terrain from these height samples, a surface must be defined from the samples. Also, a material is applied to the surface to shade the triangles. Although it is possible to create and render smooth spline surfaces based on these samples, the sample grid is normally directly triangulated, as triangles are much faster to render using today's graphics hardware.

2.4.1. Terrain Texture:

To make a interesting rendered image of any type of geometry using modern programmable graphics hardware, geometry (i.e. the triangle meshes) will have surface properties assigned to it. In general, these properties consists of local mapping parameters (i.e the texture mapping) and a pixel shader.

The shader uses the local parameters, the camera direction, the local geometry and possibly multiple input images called textures and other parameters to calculate the color of each covered screen pixel. Surface shading might be as simple as outputting an evenly lit projected texture on a surface. Lighting calculations can be balanced between vertex shaders and pixel shaders to get per-vertex and per-pixel effect, respectively. As there are generally less vertices than pixels rendered at any time, it is often faster to calculate effects in the vertex shader.

When a heightfield is to be used in a real-time engine, this heightfield will generally be rendered as a set of triangles. Simply assigning uniform colors to these polygons will not create very convincing images. Photorealistic textures can be assigned instead to increase the visual resolution of the material the terrain is made of. Typical textures include images of mud, snow, dirt, sand, grass and rock. These terrain textures can be created by artists from edited photographs or might even be generated procedurally.

2.4.2. Heightfields:

A heightfield, also called heightmap,(DEM) Digital Elevation Map, represents a height function of 2D coordinates on the horizontal plane, defined by height samples at regular spacing. The height samples can be both regular and irregular, depending on the requirements. Heightfields can be stored very compactly, because only data for the vertical axis is needs to be stored, as the horizontal components are completely regular. As heightfields are the discrete functions of 2D space, they can be stored, visualized and even edited as grayscale images. The grayscale values therefore indicates the local height. By convention, the maximum, the maximum altitude is represented by white and the minimum altitude by black.

Another advantage of the heightsfields is that the ease of texture mapping, i.e. the process of mapping (color) detail imagery onto the rendered geometry. A simple vertical orthographic projection of a detailed texture image onto the heightfields is generally sufficient. However, when a heightfield contains very steep areas, a simple vertical projection leads to an uneven distribution of texture resolution. Because of the overall advantages of memory and render efficiency, heightfields are still the most common way to specify terrain for real-time 3D applications.

2.5. Skybox

A skybox is a large cube that encompasses the entire scene and contains 6 images of a surrounding environment, giving the viewer the illusion that the environment he's in is actually much larger than it actually is. Example: Clouds or starts in sky. We have a cube that has 6 faces and needs to be textured per face.

2.6. Water Simulation

Real time simulation of liquids like water is an important task nowadays in the field of graphics. The process of rendering water in real-time is highly dependent on how much realism is needed in an application which can be a computer game, a movie or just some scientific setup being used to run some simulations. In the early days of computer graphics, water basically used to be treated as a planar surface. But with drastic improvement in the rest of the computer graphics, the need for better animation of water surface grew strong.

The complex task of water simulation can be easily understood if we can break it down into subparts.

1. Surface Representation
2. Height Field Generation
3. Reflection and Refraction
4. Fresnel term and other various phenomena like illumination to make the scene more realistic.

2.6.1 Surface Representation:

There have been various techniques that have been used in the past to generate the surface grid so as to suit the requirements of the applications. The schemes used to make the surface grid makes us help chose the points on the grid on which to apply the height-field function. This is necessary because there are infinite points on the water surface and it won't be feasible to apply the height-field function to every single one of them. Some techniques to represent the surfaces are mentioned in brief details below.

Using 2d Grids

The concept behind using a 2d grid is to represent the surface basically using a height field function. It is not accurate but simple and light-weight computation method.

2.6.2. Height Field Generation

Height fields were generated using Perlin Noise. This method provides us with a continuous noise which is very much alike or similar to random noises found in nature.

2.6.3. Reflection and Refraction

It is important to understand how light behaves when it interacts with water in reality because that is what mainly decides how it is going to look to someone viewing it. In this method, we assume the surface to be acting as a mirror. We reflect the world onto the other side of the water surface and store it into a 2d reflection texture. We then apply this texture to water surface by projective texturing. The refraction is done using something called normal map and dudv map.

2.6.4. Fresnel Equation

Fresnel equation is used to determine how much light gets reflected and what portion gets refracted.

2.7. Noise:

2.7.1. Random Number:

Random Numbers are used to introduce unpredictability. Unpredictability is generally used to define nature. Most of our daily tasks has random patterns. Even the tasks we do, the things we see and sounds we hear. It has been quite difficult to handle this randomness of nature and this difficulty was quite prominent in Computer Graphics. Generating nature like texture for items such as woods, clouds, metals, stones etc. was difficult. Using a texture image would cost memory as it would require us to use many such texture for a single item. So, use of random number to generate texture could be solution.

So, Random numbers could certainly be used to introduce Unpredictability, but often their uncontrollable output can look unnatural. As Ken Perlin said in one of his presentations, "Noise appears random, but isn't really. If it were really random, then you'd get a different result every time you call it. Instead, it's "pseudo-random" - it gives the appearance of randomness."

He clearly states that a Random Number Generator won't be enough to define noise. He says noise is not random but he defines it as a "pseudo-random" pattern. He also

says, "Noise is band-limited - almost all of its energy (when looked at as a signal) is concentrated in a small part of the frequency spectrum."

So, it pretty much explains why a simple Random Number Generator, that we are familiar with, won't be enough to define Noise.

2.7.2. The Noise Function

A noise function is a special random number generating function which takes an integer parameter, and returns a random number based on the input parameter. It is a Rn -> R mapping function when n defines the number of input parameters (dimensions). If we pass the same parameter twice, it returns the same output.

Noise function takes some input parameter, let's say 1D parameter X and returns a random number, let's say between (0 and 1). We can't just calculate values for every possible points in a given range. So, we can say its discrete. But we make it smooth using various interpolation techniques.

2.7.3. Perlin Noise:

Noise is useful for generating random patterns, especially for unpredictable natural phenomena.

However, most things aren't purely random. Smoke and clouds and terrain may look like they have elements of randomness, but they were created by a set of very complex interactions between lots of tiny particles. White noise is defined by having all the particles (or pixels) not depend on each other. To generate something more interesting than gravel, we need a different kind of noise.

That noise is often Perlin noise.

2.7.4. Perlin Noise Algorithm

Perlin noise is most commonly implemented as a two-, three- or four-dimensional function, but can be defined for any number of dimensions. An implementation typically involves three steps: grid definition with random gradient vectors, computation of the dot product between the distance-gradient vectors and interpolation between these values.

2.7.4.1. Grid Definition:

Define an n-dimensional grid where each point has a random n-dimensional unit-length gradient vector, except in the one dimensional case where the gradients are random scalars between -1 and 1. Assigning the random gradients in one and two dimensions is trivial using a random number generator.

2.7.4.2. Dot Product:

Given an n-dimensional argument for the noise function, the next step in the algorithm is to determine into which grid cell the given point falls. For each corner node of that cell, the distance vector between the point and the node is determined. The dot product between the gradient vector at the node and the distance vector is then computed.

For a point in a two-dimensional grid, this will require the computation of 4 distance vectors and dot products, while in three dimensions 8 distance vectors and 8 dot products are needed.

2.7.4.3. Interpolation:

The final step is interpolation between the 2^n dot products computed at the nodes of the cell containing the argument point. This has the consequence that the noise function returns 0 when evaluated at the grid nodes themselves.

2.7.5. Properties of Perlin Noise:

2.7.5.1. Amplitude:

The maximum absolute value that a specific perlin noise function can output.

A perlin noise function with an amplitude of n generates values between 0 and 1.

2.7.5.2. Gain:

A multiplier that determines how quickly the amplitude increases for each successive change in a Perlin-noise input.

2.7.5.3. Frequency:

The number of cycles per unit length that a specific perlin noise function outputs is the frequency.

Perlin noise has properties similar to that of a sine wave:

- It has periodic cycles of length 1/f, where f is its frequency.
- At the start of each cycle, it outputs a value of zero.

Unlike a sine wave, the output of a coherent-noise function may or may not cross zero during the middle of a cycle.

2.7.5.4. Octave:

One of the coherent-noise functions in a series of coherent-noise functions that are added together to form Perlin noise. These coherent-noise functions are called octaves because each octave has, by default, double the frequency of the previous octave. Musical tones have this property as well; a musical C tone that is one octave higher than the previous C tone has double the frequency.

The number of octaves control the amount of detail of Perlin noise. Adding more octaves increases the detail of Perlin noise, with the added drawback of increasing the calculation time.

The following graphs show the outputs of one-dimensional Perlin-noise functions n(x) with 1, 2, 4, and 8 octaves. In these graphs, o refers to the octave count.

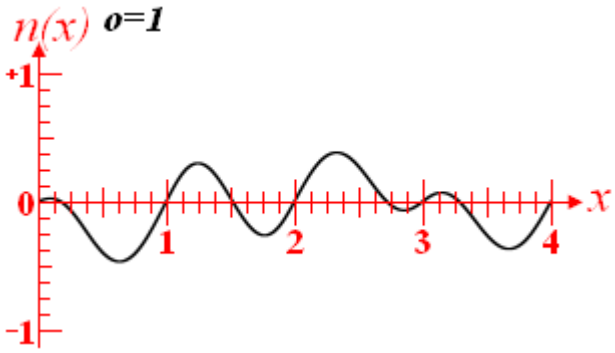


Figure 10Octave when n=1

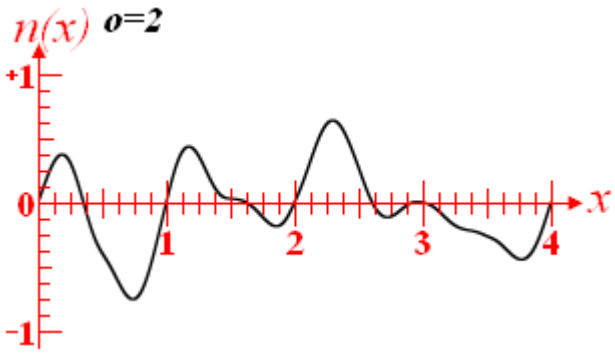


Figure 11Octave when n = 2

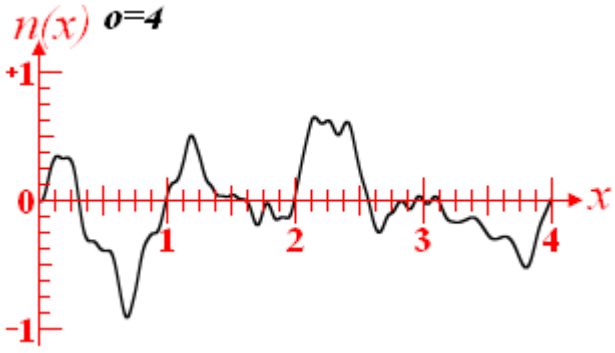


Figure 12Octave when n = 4

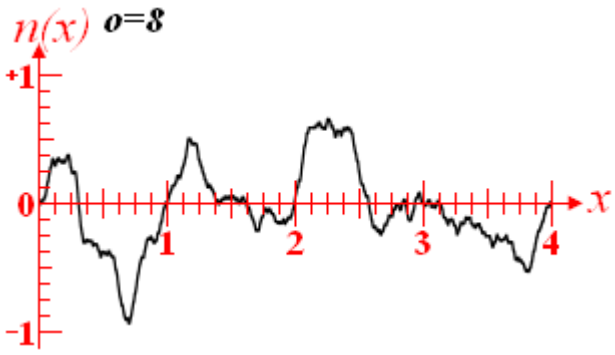


Figure 13Octave when n = 8

2.7.5.5. Lacunarity:

A multiplier that determines how quickly the frequency increases for each successive octave in a Perlin-noise function.

The frequency of each successive octave is equal to the product of the previous octave's frequency and the lacunarity value.

A similar property to lacunarity is persistence, which modifies the octaves' amplitudes in a similar way.

2.7.5.6. Seed:

A value that changes the output of a perlin noise function.

Seeds in perlin noise functions are used in the same way as seeds in standard random-number generators. By changing the seed of a perlin noise function, we change its output values; however, that function's frequency and amplitude are not altered.

2.7.5.6. Fractal Brownian Motion:

fBM (fractional Brownian motion) is the simplest of the composite Perlin noise algorithms. Because more complex turbulence algorithms are extensions of this basic idea, this algorithm is simply called 'turbulence' in Scape.

It calculates the weighted sum a number of scaled Perlin noise ‘octaves’, adding details over a broad range of scales. Each octave is responsible for a different size of features, or, inversely, a dominant 'frequency' in the generated 2D signal.

The seed should be any number between 0.0 and 1.0. The lacunarity (the frequency multiplier between octaves) is typically set to a number just under 2.0 (e.g. 1.92) to prevent regular overlap in Perlin noise lattices.

To cover all possible feature scales, the number of octaves is typically a bit less than

Log (terrain width) / log(lacunarity)

So, for a 1024 x 1024 heightfield, about 10 octaves are needed. The gain is controlled by the user and directly influences the terrain roughness.

3. METHODOLOGY

3.1. Initialization of Required Libraries:

The project used glew for querying and loading OpengGL extensions, glfw to create and manage windows and OpenGL contexts as well as use input devices and stb_image to load texture images. GLM, a math library was used for mathematical calculations and operations while imgui was used for the graphical interface for changing the different parameters in runtime.

3.2. Grid-Mesh Creation:

The first step was to create a mesh of triangles connected to each other in a plane that act as base for the entire project. OpenGL takes vertex array and indices array in the buffer and connects the vertex according to indices provided which were calculated inside the Init function in terrain.h. The grid mesh was drawn in xy plane resulting a output as shown below:

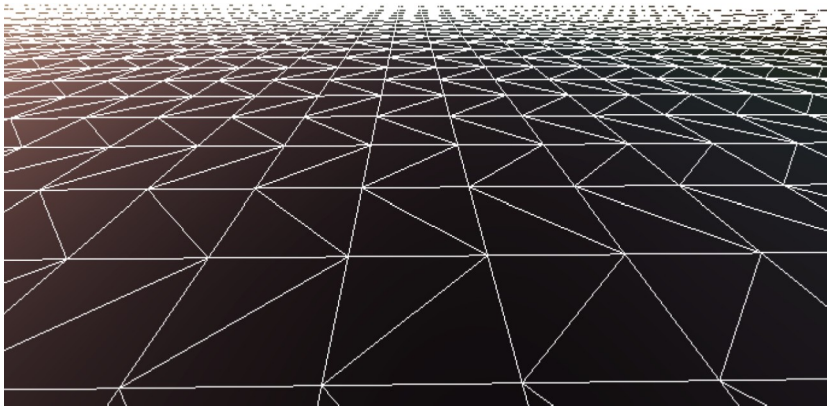


Figure 14Grid Mesh Creation

3.3. Heightmap Generation:

Above we generated a mesh of triangles that act as a base, now we need a height for each vertex defined. The heights were calculated using different perlin based algorithms like fractional brown method. They take a point and seed as input and give us a randomly generated output which depends on its counterparts. Therefore the mesh now looks more like a landscape.

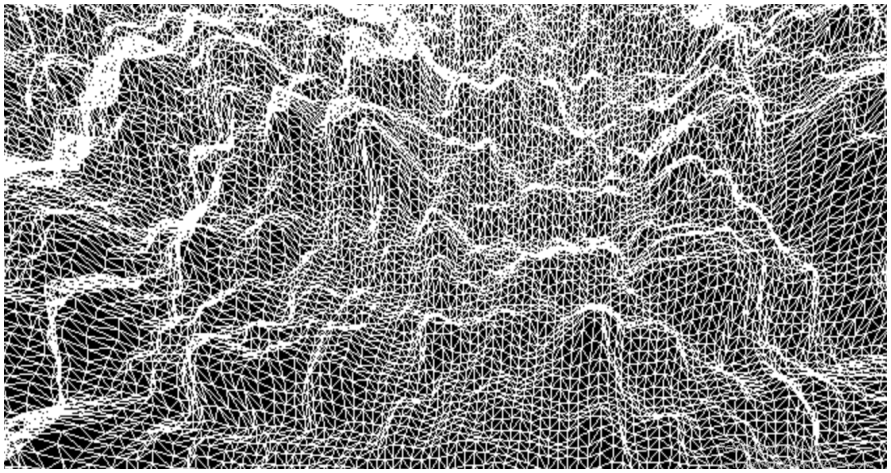


Figure 15Height map generation

3.4. Texture Mapping for terrain and Skybox:

Now after the terrain is made we need to make it look realistic using different natural textures like sand, rock, grass and snow. We categorize different textures based on heights, so snow is at the top of terrain and sand at the bottom while rock and grass in middle. The textures are loaded in shader and mapped to vertex's using suitable height conditions. Also skyboxing is used to bound the world inside a cube that makes it look more realistic having blue skies. The resulting output is as follows:

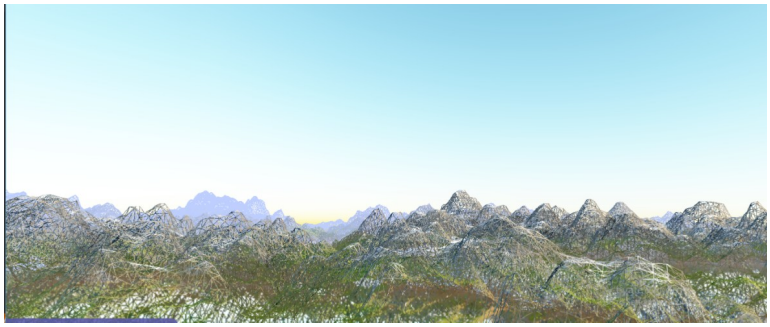


Figure 16Skybox

3.5. Water Simulation:

For simulating water we first create the mirror of the environment below it, then water texture is added to the parts of terrain where the height is below the plane. The water texture has some transparency so that the mirrored environment texture is shown. The reflection is just taking the environment and flipping it upside down, clipping it to the surface that we want to show. The waves moving in water causes the reflected object to be distorted i.e. called refraction. Refraction in created using dudv map.

A normal texture looks like this:

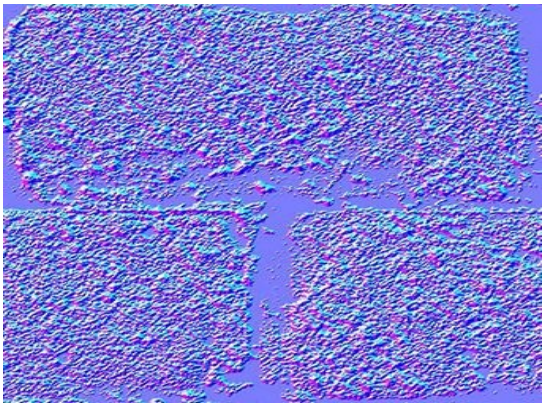
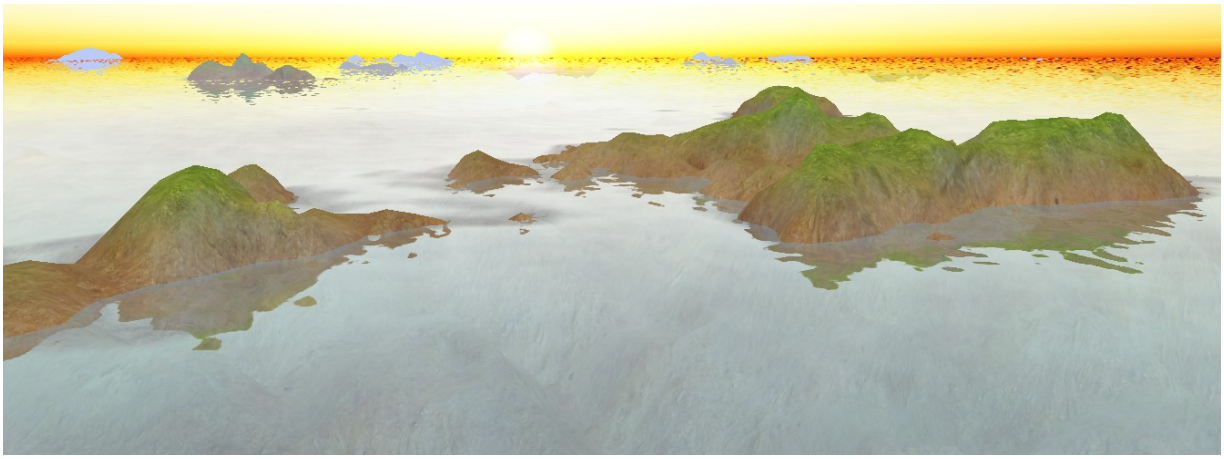


Figure 17Water texture

In each RGB texel is encoded a XYZ vector: each color component is between 0 and 1, and each vector component is between -1 and 1, so this simple mapping goes from the texel to the normal. What we do with this image is basically use it to bump map our water to give it the appearance of highly tessellated and realistic. The realism comes when we use per lighting on the water. Basically if we didn't have a normal map we would just depend on the normal of the surface, which is pointing straight up. The light would look horrible and realism would be absent. We also use normal map for our Fresnel term calculations.

4. RESULTS



5. CONCLUSION

Thus the terrain was created using perlin noise and height map algorithms. Textures for sands and rocks were loaded from shaders and mapped to vertex using suitable height conditions to make the terrain look more realistic. The water in the landscape possessed transparency and hence reflection and refraction phenomenon was observed. The sun set and rise showed the different lights at the horizon due to atmosphere scattering.

6. FURTHER ENHANCEMENT

A number of areas for future improvements might extend the capabilities of the suggested and implemented techniques. The procedural terrain generation algorithm used in the project can be very handy on creating open world environment games like PUBG or GTA. Currently the world is populated with textures only, we can add various trees, grasses and other natural things on the terrain. Also the water simulation uses the mirrored environment which consumes more computation. Other optimized algorithms can be used to make water simulation more realistic. We currently use perlin noise to render the heightmaps for terrain. There are other algorithms like simplex noise that solves the problems of perlin algorithms

7. REFERENCES

- *https://www.khronos.org/opengl/wiki/*
- *http://www.opengl-tutorial.org/*
- *https://learnopengl.com/*
- *http://ogldev.atspace.co.uk/*
- *http://nehe.gamedev.net/*
- *https://open.gl/*
- *http://openglbook.com/chapter-1-getting-started.html*
- *https://www.youtube.com/watch?v=W3gAzLwfIP0&list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2&index=1*
- *http://docs.gl/*
- *https://www.3dgep.com*
- *https://mzucker.github.io/html/perlin-noise-math-faq.html*
- *http://www.decarpentier.nl/scape-procedural-basics*
- *https://thebookofshaders.com/*
- *http://libnoise.sourceforge.net/index.html*