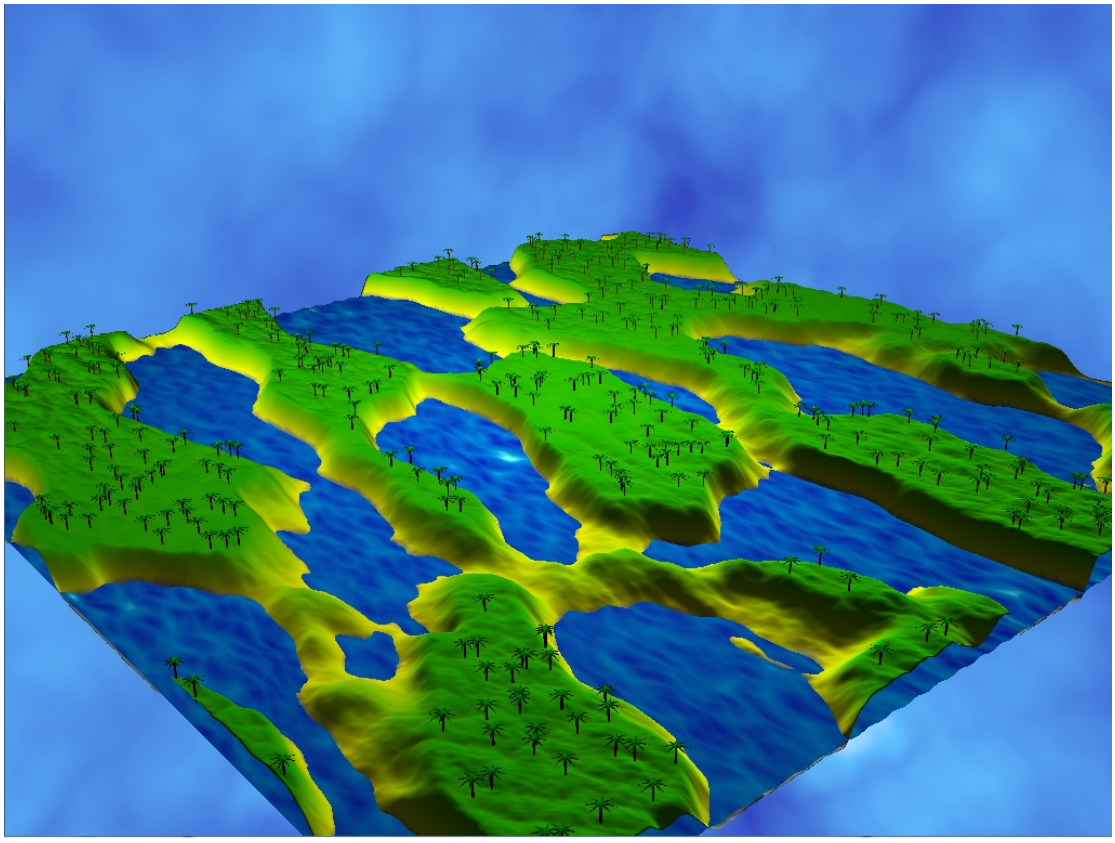
AG0902A – 3D Graphics Programming

Real Time 3D Procedural Terrain Modelling

By Vladeta Stojanovic
(0602920@live.abertay.ac.uk)



Introduction

This report provides an overview of the design and development of the *Tropical Island Generator* application for the second semester coursework of the AG0902A 3D Graphics Programming module. The report also describes the theory behind the key algorithms used in the development of the application.

The *Tropical Island Generator* (abbreviated from now on as TIG) application was designed to demonstrate the key concepts behind real time 2D & 3D procedural generation of terrain meshes. The area of real time procedural content generation is becoming increasingly popular as interactive 3D applications become more complex and the hardware that runs them becomes more powerful.
The main use of procedural modelling in terms of computer games, 3D applications and other interactive media is that is allows for the generation of both 2D and 3D data (either offline or in real time), that would otherwise be costly or time consuming to produce manually (or in some cases the sought after data would be unavailable). The procedurally generated content can be anything from 2D textures and 3D mesh data, to animation and sound data, and even certain game logic elements (like AI routines that rely on pre-existing input data).

However, the most poplar use of procedural modelling in interactive 3D applications is in the filed of terrain generation. Procedural modelling of terrains allows for the creation of realistic scenes that can depict anything from a topologically common terrain to an alien landscape, along with generation of naturally occurring elements such as vast mountain ranges, erosion, glaciers, hills, canyons, lakes and islands. Procedural modelling can also be used to generate trees, vegetation, water, clouds and other natural phenomena that can populate and compliment the generated scene.
The complexity of these scenes can be anything from a simple 512x512 procedurally generated coherent noise terrain mesh in a 3D game or application, to a highly complex fractal based terrain model that can only be rendered offline and requiring vast amounts of computational power.

The main requirement for the TIG application was the procedural generation of terrain meshes. The TIG application demonstrates by procedurally generating and texturing an island terrain mesh, along with a water mesh and a skybox, as well as the random placement of 3D palm tree models at certain elevations on the terrain mesh. The tree model data is hard coded into the application and all of the textures are generated procedurally, thus no external 2D or 3D data used. The application then allows the user to move around and inspect the generated 3D scene.

The application was programmed using the provided framework, which makes use of OpenGL 3.3 and Win32. LibNoise (http://libnoise.sourceforge.net) was also used extensively for the procedural generation of the terrain 3D mesh data, as well as the water and the skybox 2D texture data. The application makes use of vertex and fragment shaders for lighting and texturing of the 3D scene, and the animated displacement of the water mesh (in order to simulate wave motion).

Overview of Noise Generation

The definition of noise can be broadly interpreted depending to what medium it is applied to. In terms of 2D and 3D computer graphics, specifically procedural modelling, noise can be defined as a randomly generated array consisting 1D, 2D or 3D coordinate values. Though noise itself is a function, and not a primitive, the generated values can be thought of as a primitive. The actually definition of noise can often be confusing to define, as there are many different noise functions that can be used for the generation of noise primitives.

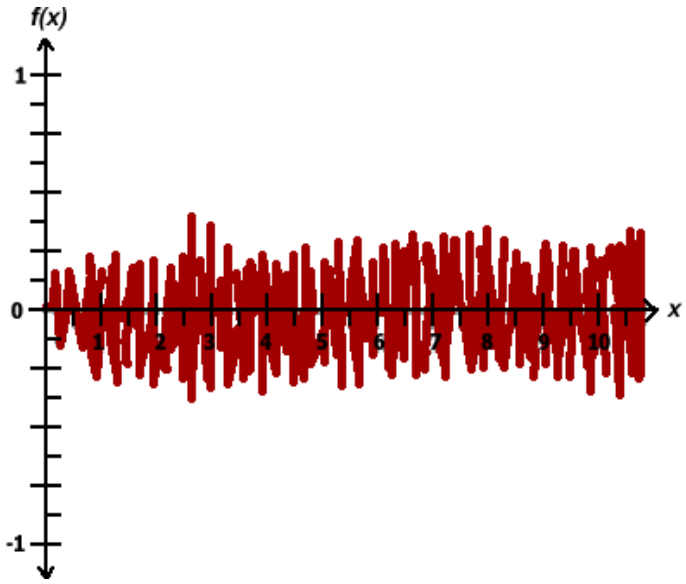
However, the standard interpreted definition of an ideal noise function (Peachey, 1998 and Rost, 2008) can be defined as:

- Being able to create repeatable stochastic values based on the given input values.
- Having a high-precision floating point output range from -1 to 1.
- Being band-limited, with a maximum frequency of 1.
- Not showing regular patterns or periods after repeated generation.
- Being stationary, where the small floating point value of the function is independent of its large scale position (depending on its application, i.e. stationary in it's size regardless if it's used to generate a random blade of grass or an entire planet).
- Being rotationally invariant, meaning that its value is uniform in all orientations regardless of direction.
- Being able to be defined in multiple dimensions.

The values of the generated coordinates are based on a given input to the noise function that acts as a filter for creating repeatable stochastic values. A coherent noise function when analysed will present the following five properties (Bevins, 2005):

1) Passing in the **same input value** will generate the **same output value**.
2) Passing in an input value with a **small change** will generate an output value with a **small difference**.
3) Passing in an input value with a **large change** will generate an output value with a **random difference**.
4) Depending on the size of the dimension of the noise function, the size of the input dimension must be the same.
5) The returned output value is a scalar value.

A one-dimensional non-coherent noise function (defined as *f(x)*), will produce the following result on a 2D graph:



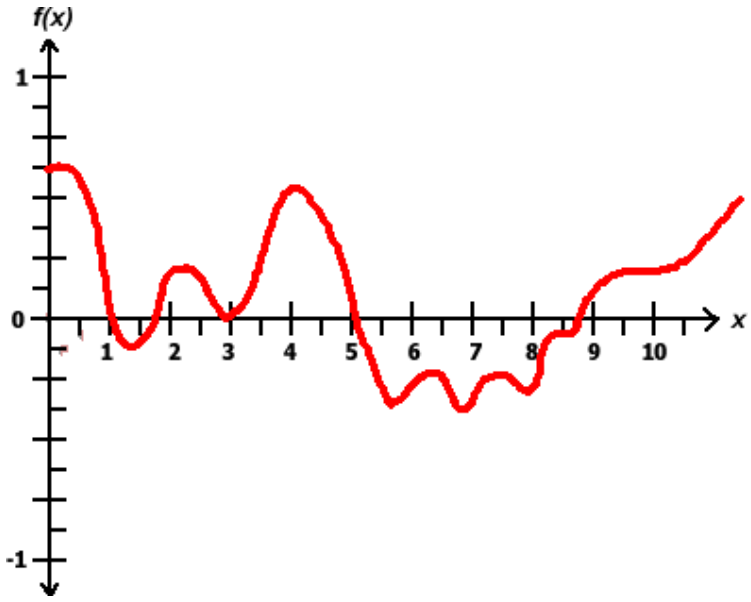
The non-coherent nature of the undefined noise function above shows that non-coherent noise is not suited for use in procedural generation due to the normalized high-frequency of the outputted value. The non-coherent appearance of the above noise function occurs in non-stationary and variant coordinate noise functions.

There are many different noise function implementations available, and some give very similar results. Notable noise functions include lattice noise, value noise, gradient noise, value-gradient noise, lattice-convolution noise, sparse-convolution noise and most notably Perlin noise (as well as a few others not mentioned here).
Since there are so many different noise functions, sometimes it can be hard to differentiate between them. For the sake of clarity, the use of noise functions that were used in the TIG application make use of coherent noise combined with other different noise modules (such as Perlin noise). Coherent noise can be thought of as a template noise function from which other noise functions can be derived and combined with, in order to form the final desired noise value coordinates. Since coherent noise is very similar to value noise, the operation of the value noise function can be used as a basis for explaining how coherent noise works.

A value noise function is constructed by using pseudo-random points at each lattice point. The given value noise function can then be interpolated from these random values. The lattice grid for the value noise function is constructed by uniformly distributing pseudo-random number values at every point in texture space whose coordinates values are whole numbers (integers). This in turn makes up a sort of net of integer points that can be interpolated across to produce a coherent noise output value. Pseudo-random values in the range fro -1 to 1 are assigned to each integer lattice point and are interpolated, and in the case of the examples below, along the x-axis (Peachey 1998).

The quality of the noise function output largely depends on the type of interpolation that is used to interpolate between each point in the given axis. This along with the varying values of frequencies and amplitudes is used to create various different noise functions (Rost 2008).

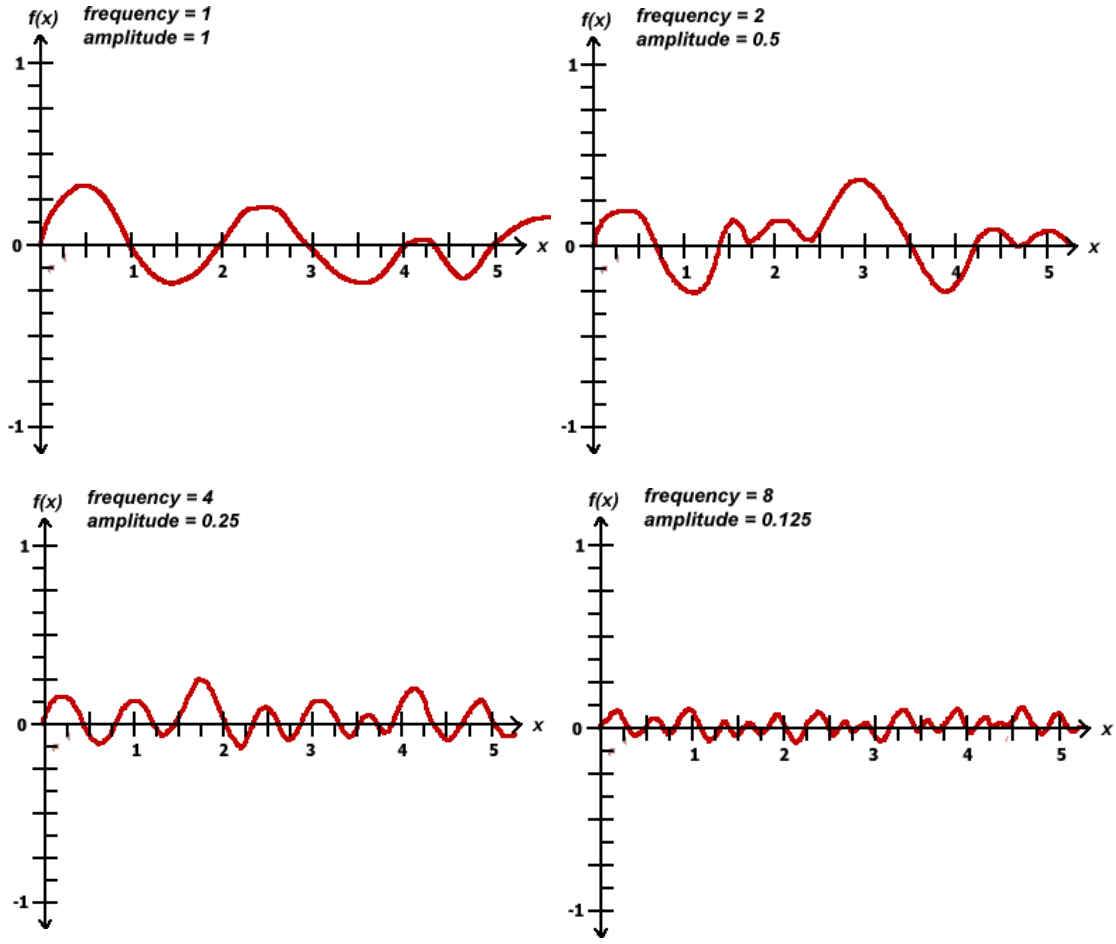
Now to move onto the actual representation of a continuous 1D coherent noise function, below is an example of a generic value noise function in the range of -1 to 1:



By altering the frequency, amplitude, octaves and persistence of a given coherent noise function, new noise functions can be derived.
The frequency of the given coherent noise function is used to define the number of cycles of unit length the noise function will output. A cycle is defined as point on the x-axis where the graph noise function intersects.
One point to note is that the output value of a coherent noise function may or may not cross the x-axis (go above or below zero) during the middle of a cycle. Thus the frequency of a coherent noise function is non-conforming in terms of cycles, unlike the sine or cosine functions (Bevins 2005).

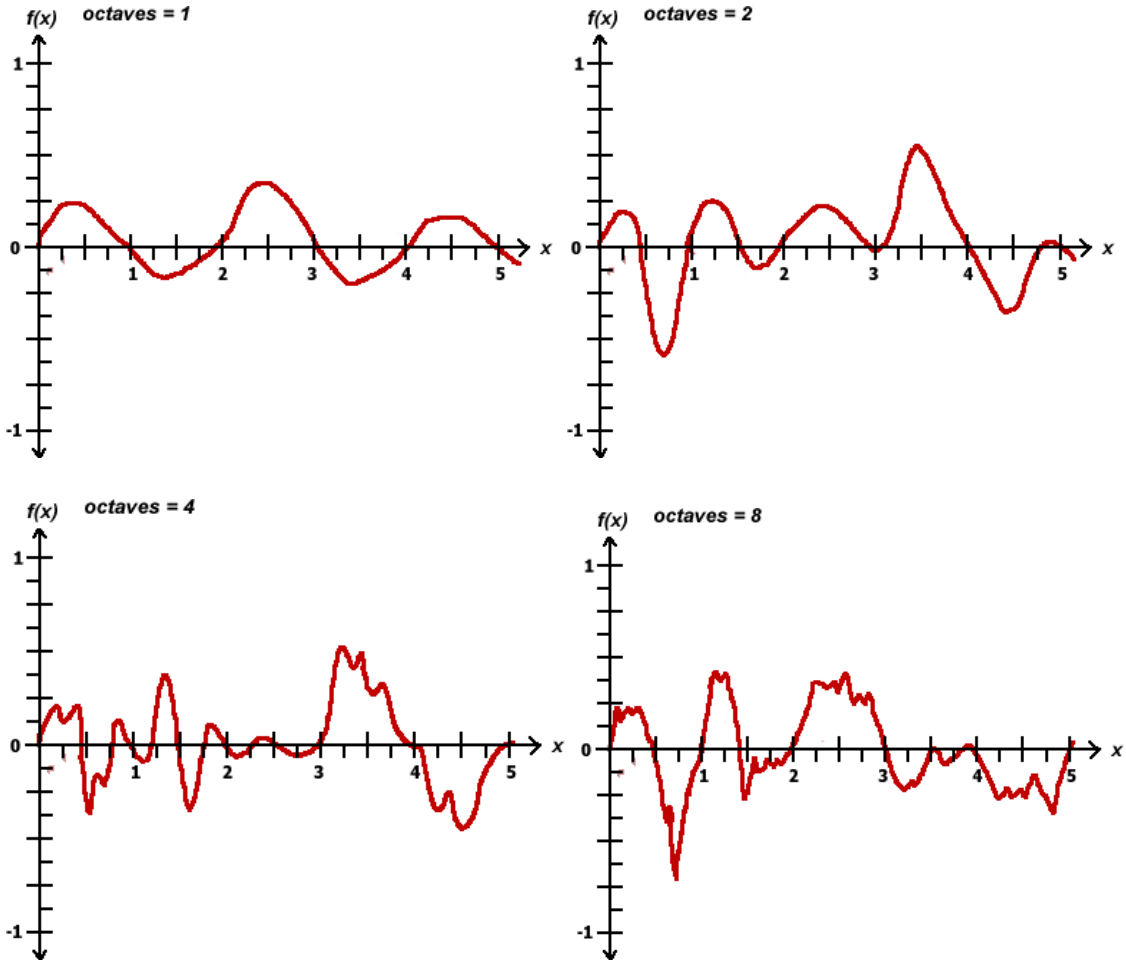
Increasing or decreasing amplitude of a coherent noise function alters the maximum absolute value range that the function can output. The typical noise amplitude value is in the range from -1 to 1 (as for the value noise function and most examples of coherent noise). The generated output values are placed within the given amplitude bracket.

Below are examples of a coherent noise function with different frequencies and amplitudes:



One way to model Perlin-like noise using coherent noise functions is by representing each different coherent noise function as an octave. This terminology is used because in terms of coherent noise functions, each successive function has double the frequency of the previous coherent noise function. Thus when modelling a coherent noise function, the number of octaves for that function can be used to control the amount of detail of the generated noise.

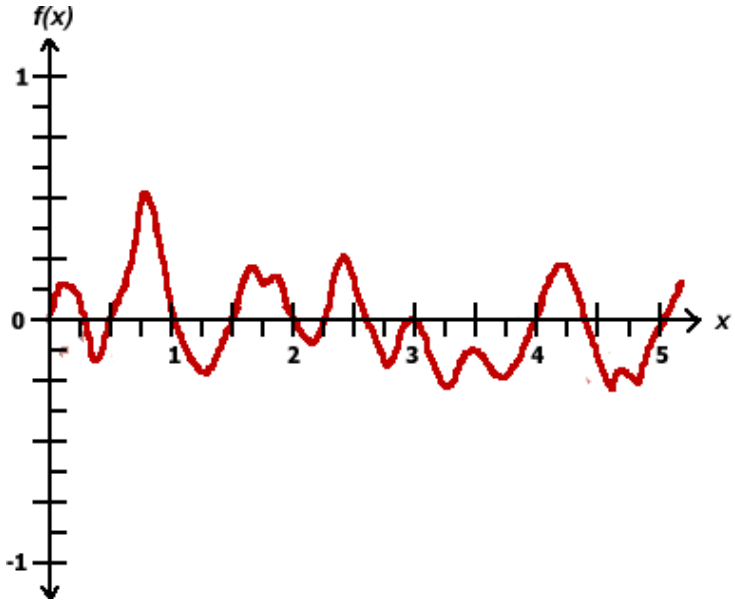
Below is an example of a coherent noise function with increasing octaves:



Likewise, persistence in a coherent-noise function is used to define how quickly the the amplitudes (maximum absolute output value bracket) diminish for each iteration of an octave (coherent noise function) that is used to model the Perlin-like noise function. The amplitude of each iteration of an octave is equal to the product of the previous iteration of the octaves amplitude and persistence values. Thus increasing the value of persistence creates more rougher and detailed noise output for the Perlin-like noise function (Bevins 2005).

In terms of generating Perlin-like noise from coherent noise functions, the constructed coherent noise function can be defined as a product of a number of coherent noise functions (octaves) with infinitely increasing frequencies and amplitudes.

Below is an example of a coherent noise function series (which can be referred to as a series of octaves), that can be used to form the final noise value output of a Perlin-like noise function:



Though this is **not real Perlin noise** (thus why it was being referred to as Perlin-like noise). Instead, this combination of various coherent noise functions is often referred to as "**Fractional Brownian Motion**" (FBM). FBM is based on the Gaussian process, which is used for generation and normal distribution of random values. It is in essence an imitation of Perlin noise, but less expensive to compute. LibNoise can combine FBM noise functions with other noise functions, such as the Perlin noise function, in order to generate pleasing noise data.

A brief overview of real Perlin noise

Noise can be mapped in multiple dimensions, from R to R^n. The most common dimensions for mapping noise are R^2 (2D) and R^3 (3D). 3D noise (often referred to as solid-noise), is used for generating continuous procedural textures for 3D objects. The main property of 3D noise is that it does not cause any artefacts (such as texture seams, often found in non-continuous texture maps) in the UV texture coordinates of the textured object. Thus an object that is textured using 3D noise will have a coherent noise value for each of the texels in the local 3D space of the applied object texture.

As mentioned before, noise is band limited, and is concentrated in a very small frequency spectrum, usually in the range form -1 to 1. Change in high and low frequencies doesn't change the range of the spectrum that the noise is concentrated in. Noise is required to be random, but in order to use it as a modelling primitive, it must be applied to a given point in R^n space. Again, remember the previous definition of a noise function being stationary and rotationally invariant. This means, in terms of Perlin noise, that regardless of the point in R^n that the noise function is applied to, it's spectrum (range) does not change.

From now on, for simplicity, it is assumed that the Perlin noise function being described is used to create a 2D noise map (Thus it is defined in R^2 space).

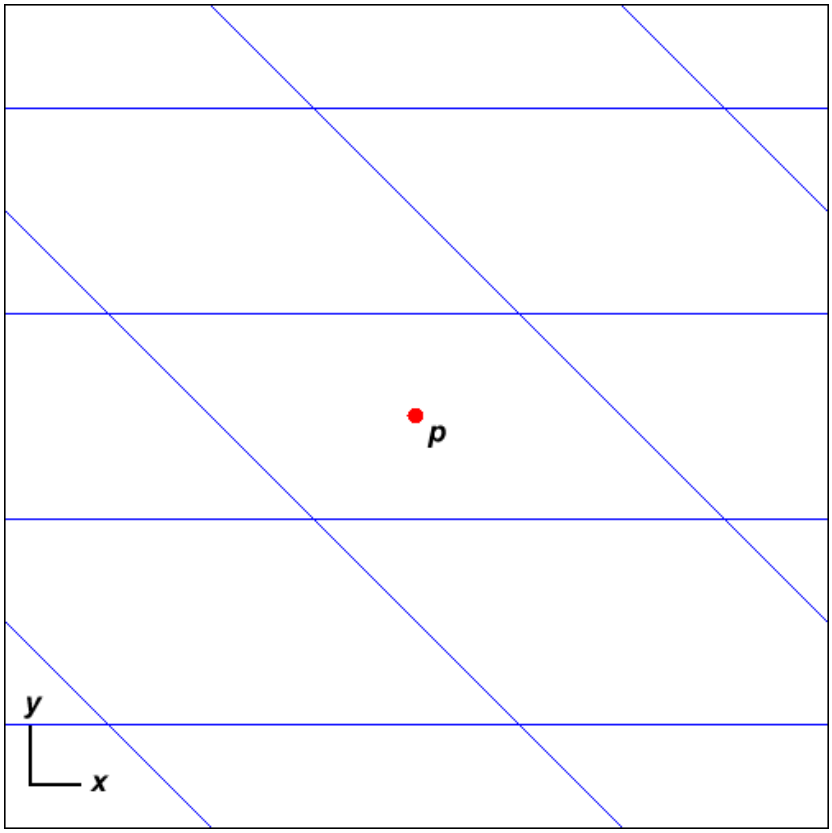
Given that a Px is a vector value point in R^n (in this case presumed to be in R^2), applying the function called *noise()* to Px as *noise(*Px*)* will alter the value of Px based on the input parameters of the noise function. Likewise, increasing the size of Px will generate that many more frequencies when applied to the noise function as *noise(*n.Px*)*, with the frequency parameter in the noise function is defined as: *frequency* = n.

Going back once again to what was said about value noise, the given noise function is constructed by using pseudo-random points at each generated lattice point. Perlin noise uses the methodology a bit differently, as the pseudo-random points are presented in a form of a spline approximation at each corner point of the generated lattice. This provides the Perlin noise function with a distinct advantage of being able to interpolate pseudo-random values at arbitrary points on the spline, without having to pre-compute these values over the entire lattice or some other volume defined in R^n.

The lattice generated by the Perlin noise function exists in the space of R^n (in this case R^2), that is divided into a regular grid of cells (the grid is square for R^3 and a volume cube for R^3). Given an input value, defined as P, the Perlin noise function will look at each of the surrounding grid points, where the size of the grid points around P depends in what dimension P is defined in. For the given size of R^n, P will have 2^n surrounding grid points.

For simplicity, P can be thought of as single dot of noise the size of a pixel in a 2D noise map.

Below is the first picture illustrating P defined in R^2:



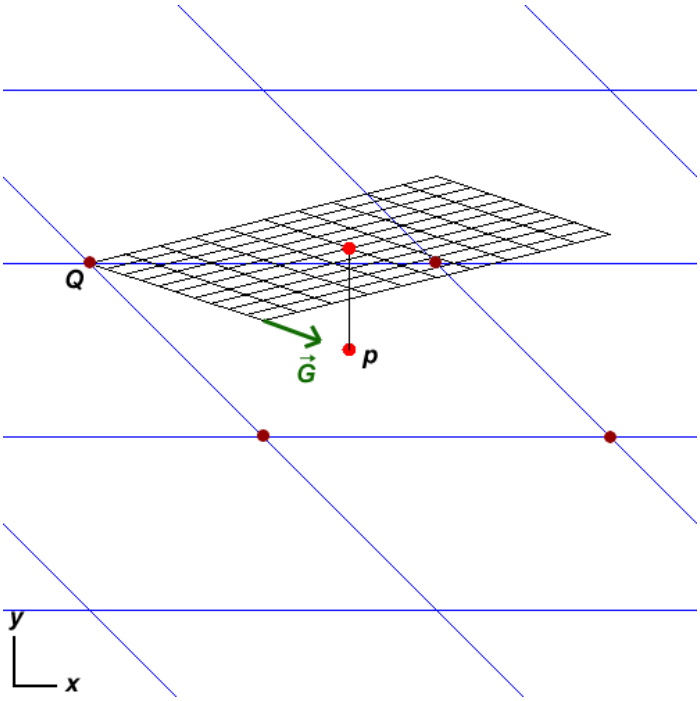
There are two ways to compute the pseudo-random noise values (referred to as gradient noise values). The first way makes use of bi-cubic and tri-cubic spline interpolation (for R^2 and R^3), along each of the gird points around P (Perlin 1983). The other way involves using wavelet functions along each of the grid points around P to in order to interpolate the generated gradient values (Perlin 1998). Wavelets can be used instead of splines, but it beyond the scope of this report to explain how they function, so the Perlin's original method using spline interpolation from each grid point around P will be explained instead.

Now going back to the original definition of P defined as a point in R^2, each surrounding grid point around P is defined as Q. For each of the grid points around P, a pseudo-random gradient vector G is computed. The same value gradient vector is computed for all instances of Q (all surrounding grid points around P).

The inner product is then computed as:

$$\vec{G} . (P - Q)$$

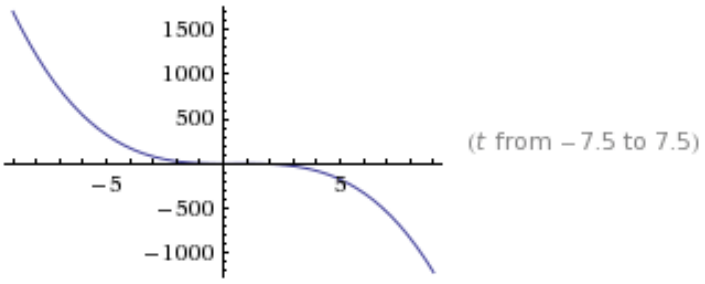
This gives the value at P of the above linear function with a gradient value of G, which has a value of zero at grid point Q (Perlin 1999). If the noise function is defined in R^2, there are 2^2 grid points to evaluate around P. A linear gradient value based on G is constructed from any of the surrounding grid points (via the above linear function).



After the gradient value of G for the given instance of Q is computed, a “drop-off filter” is applied to the gradient line function from Q (Perlin 1999). The drop-off filter is the parametric S-shaped curve. It's purpose is to make the value of G drop off to zero at the unit distance to the next grid point.
The drop-off filter is constructed using two cubic S-shaped curves. The curve is defined as:

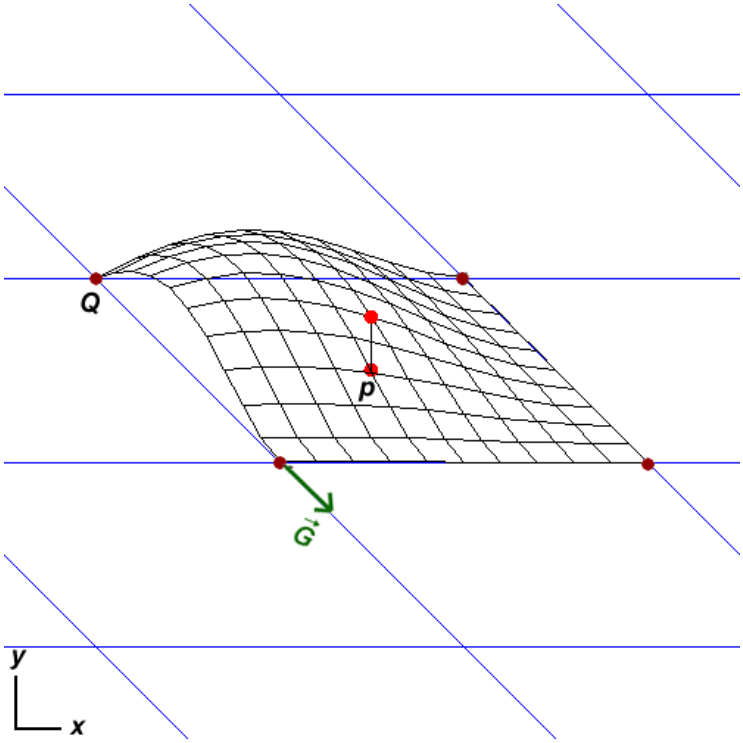
$$3t^2 - 2t^3$$

And looks like this:



The number of curves used by the filter is once again dependent of the size of R^n. For 2D noise, as used for this explanation, two curves are needed for each of the axis (x and y). If the noise function was defined in R^3, a third curve would be needed for the z-axis.

Below is an example of how this curve looks when applied to G:



An additional “weighting filter” is applied to the linear gradient value of the grid point. This linear interpolation is defined as 2^n -1 (corresponding to R^n), with each curve defined as n.S for each instance of Q, which is defined as 2^n (again, based on the initial definition of R^n). The linear weighting function then clamps the starting value of the curve from Q to zero along G. The constructed spline is then used to interpolate the value of G from the zero value of the spline starting at Q, to the maximum absolute value of the spline, and down again to the drop-off value, which saturates to zero. This is done for each of the neighbouring grid points around P. The gradient value of G is then interpolated along each of the splines from each of the grid points that are an instance of Q.

The above computation must be done very quickly, since the noise function has to be iterated many times, for each input point (Perlin 1999).

A noise function will eventually repeat it's values, but only after a long distance (of unit measurement), thus it's repeated values are not really noticeable. The lattice generated for each point P is constructed with 256x256 grid points (and 256x256x256 for R^3). A pre-computed table of gradient values can be used, containing 256 pseudo-random coefficient values, from which the Perlin noise function maps the values of I & J (I & J corresponding to the x & y values of the given lattice point) along the given spline into a unique number between 0 and 255.

The pre-compute table of gradients can be found using a Monte Carlo simulation. A Monte Carlo simulation is an algorithm making use of repeated random sampling to output it's results. In terms of implementation of the Monte Carlo simulation for computing the table of gradients for Perlin noise, the 256 gradient vectors are derived from uniformly distributed points on the surface a of a sphere (defined by a parametric sphere equation, whose unit radius is defined as 1.0).

Each point on the sphere surface represents a vector with a uniformly random direction. The uniform distribution of points on the sphere surface is computed via the following conditions (Perlin 1998):

- Choose points uniformly within the cube, whose range is defined as [-1...1]^3.
- Discard any points outside the unit radius of the sphere.
- Project remaining points onto the sphere surface.

This is done for x, y and z coordinates, but if it's only being used for 2D noise, the z coordinate values can be discarded. This instance of the Monte Carlo algorithm can be defined mathematically as:

$$\sum_{i=0}^{i=255} \sum_{\vec{p_{(x^2)}}+\vec{p_{(y^2)}}+\vec{p_{(z^2)}}=1.0}^{\vec{p_{(x^2)}}+\vec{p_{(y^2)}}+\vec{p_{(z^2)}}<1.0} \vec{P_{xyz}} = random(f:-1\rightarrow 1)\rightarrow \vec{G}_i = normalize(\vec{p}_x, \vec{p}_y, \vec{p}_z)$$

This can be expressed in pseudo-code (Perlin 1998):

```
for 'i' in [0 to 255]
    repeat
        x = random[-1 to 1]
        y = random[-1 to 1]
        z = random[-1 to 1]
    until x^2 + y^2  + z^2 < 1.0
    G[i] = normalize [x, y, z]
```

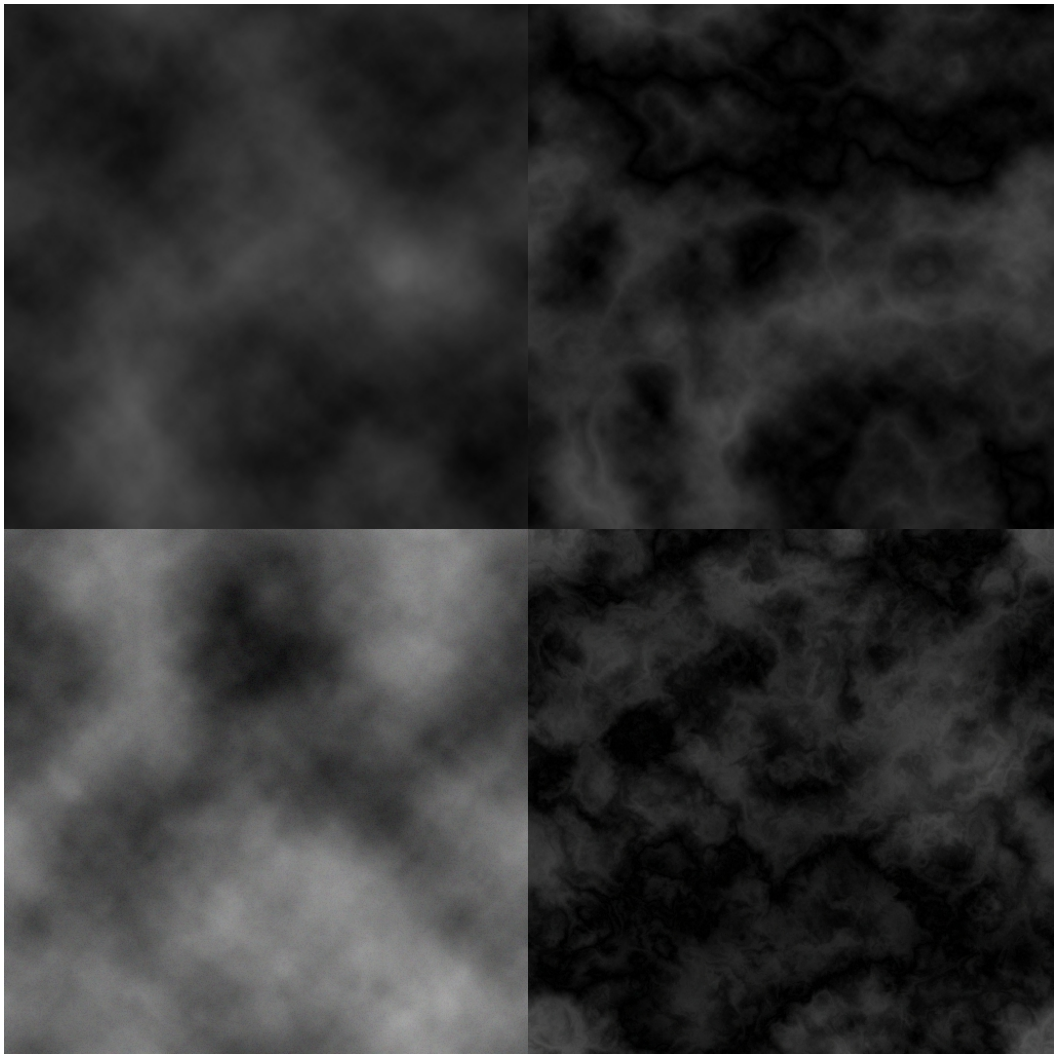
The generated 256 values are then normalized. The actual interpolated noise value for point P is computed relative to the centre of the spline. This value is derived from the gradient value lookup table set by the vector G as it drops off to zero.

The final computed value for 2D noise defined in R^2 is given by the sum of the four spline interpolation values for the given input values of the x and y coordinates. In terms of mapping these values as UV texture coordinates on a 2D bitmap array, the transfer of the noise value for each input point (in this case a pixel) can be defined as the value of the constructed spline, S, with gradient values *ijk*, at a given point with input coordinates x and y:

$$\boldsymbol{S}_{(i,j,k)} . (u, v, w) . (\vec{\boldsymbol{G}}_{(i,j,k)}\, dot\, [u, v, w])$$

Where the drop-off value for each component of the UV map is given by the cubic spline approximation defined as: drop(t) = 1 – 3(t)^2 + 2(t)3, where the returned value is zero if t > 1 (Perlin 1999).

Since the noise function mapping is done only 2D, the W component of the UVW map can be disregarded. Ideally, no matter if a 1D, 2D or 3D dimensions are defined, for the Perlin noise function is by default defined in R^3 space (even though the above explanation is only relative to R^2 space).



The above images are rendered using the Photoshop cloud and difference could filters, which are essentially the alterations of the Perlin noise functions ***sum 1/f(noise)*** and ***sum 1/f(|noise|)***.

Overview of Fractals

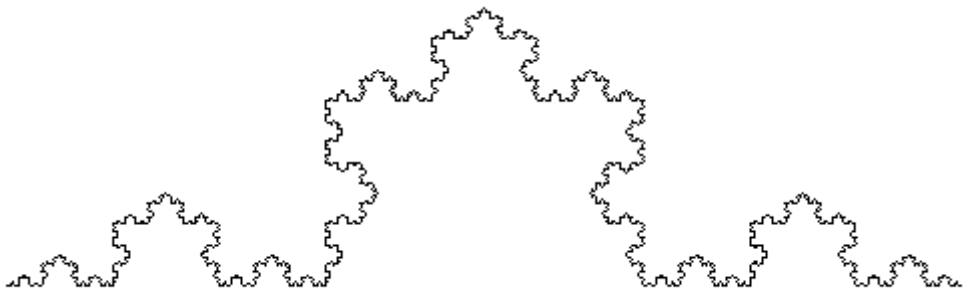
Fractal geometry, in terms of computer graphics, is used to map chaotic and complex patterns into a series of geometric primitives consisting of theoretically infinite depth.
Fractal patterns are often found in nature, thus the procedural modelling of 2D and 3D data based on fractal generation algorithms can be used to create realistic natural elements such as chaotic and random displacement of terrain geometry. The key to understanding the nature of fractals is to understand that in terms of visual complexity, a fractal has infinite complexity, but at different scales. Thus why images generated using fractal rendering algorithms can theoretically create an infinitely complex amount of detail per each iteration.

Fractal geometry is projected in a spatial dimension. A fractal spatial dimension can have a real number value, such as 1.8 or 2.7. These real values of fractal dimensions provide a continuous “slider” for the visual complexity of the generated fractal (Musgrave 1998). Thus fractals can exist in infinite dimensions with infinite complexity

The operation of a fractal relies on a basis function. The basis function is used to generate a result that is repeated and scaled at infinite complexity. The basis function should return a particular result, and should ideally have the following features:

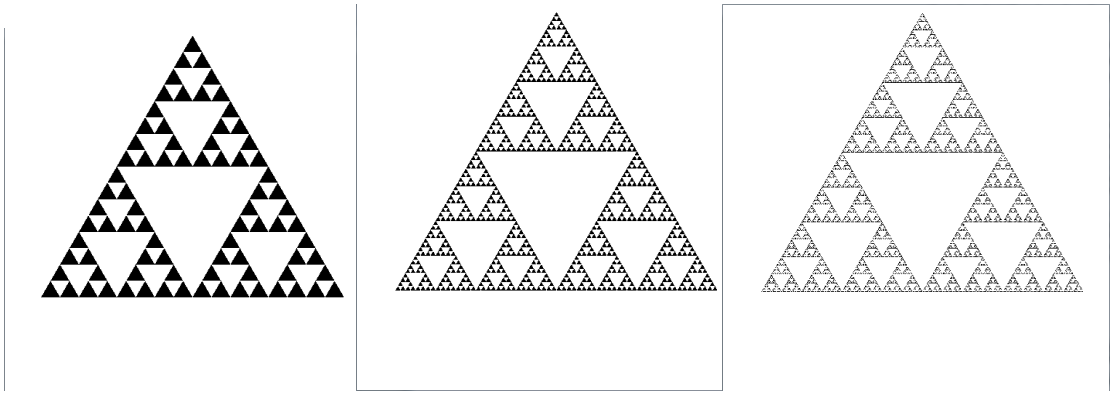
- It should be controllable
- It should be self-similar
- It should be deterministic

These features are very similar to the outlined requirements for the ideal noise function, that was described at the beginning of this report. Therefore the use of a noise function as the basis function for the generation of a fractal is a good choice.
The common basis functions used for fractal generation are Perlin noise and FBM noise functions. Thus fractals can be used to procedurally generate extreme complexity using relatively simple basis noise functions. Discussion of the mathematical properties of fractals is omitted from this report, since fractals are heuristic, understanding the underlying maths is not necessary for their application. However, further research into mathematical theory and implementation of fractal geometry generation algorithms is highly recommended.
Below is an example of a Koch snowflake fractal:



The Koch snowflake was one of the earliest discovered fractals. Along with the Sierpinski gasket (described below), it is one of the most commonly modelled fractals.

Another very simple example of 2D fractal rendering can represented using the traditional example of a 2D Sierpinski gasket. The Sierpinski gasket is based on a fixed point base function that constructs a self-similar fractal representation per iteration. Thus, the 2D Sierpinski gasket can be rendered at any set zoom level. Below is an example of a 2D Sierpinski gasket fractal rendered in OpenGL (at varying complexities):



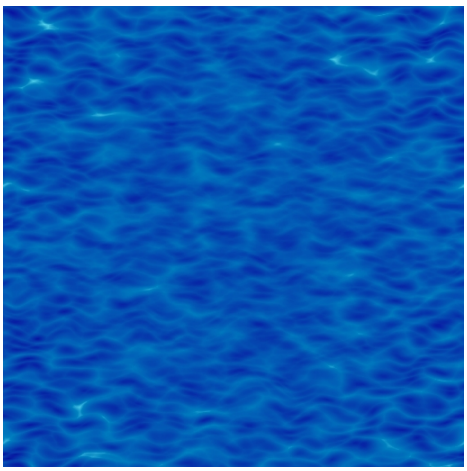
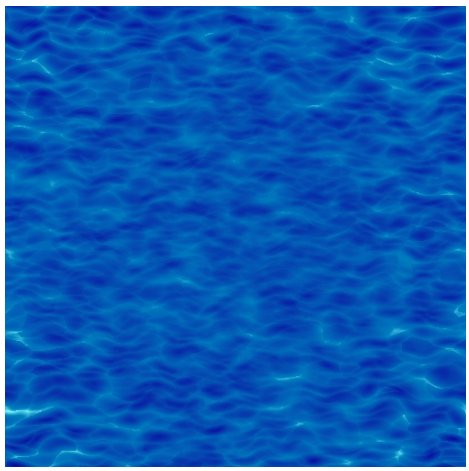
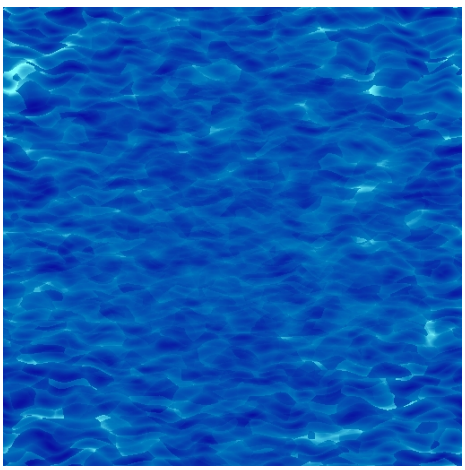
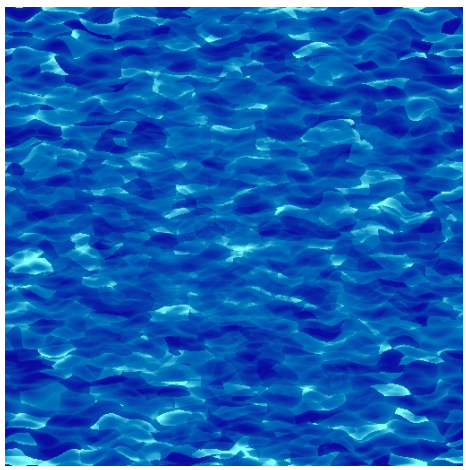
The actual gasket drawing function increments the complexity of the gasket recursively. This is an example of where the fractal base function is the fractal itself:

```
void DrawGasket(point2 a, point2 b, point2 c, int m)
{
    point2 v0, v1, v2;
    int j;
    if(m>0)
    {
        for(j=0; j<2; j++) v0[j]=(a[j]+b[j])/2;
        for(j=0; j<2; j++) v1[j]=(a[j]+c[j])/2;
        for(j=0; j<2; j++) v2[j]=(b[j]+c[j])/2;
        DrawGasket(a, v0, v1, m-1);
        DrawGasket(c, v1, v2, m-1);
        DrawGasket(b, v2, v0, m-1);
    }
    else(triangle(a,b,c));
}
```

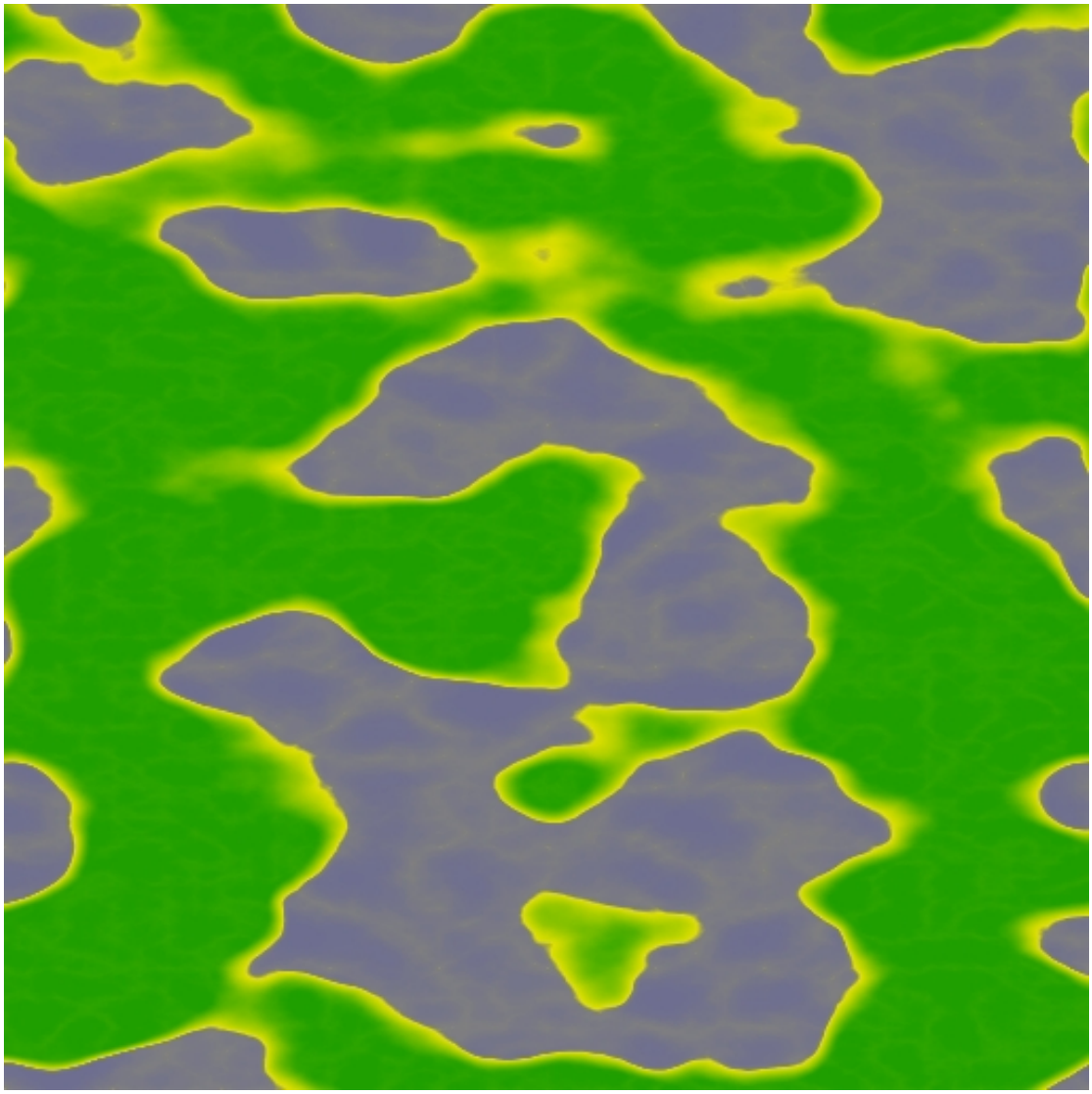
The Sierpinski gasket is a very simple example of a fractal, as it's recursive generative nature is due to the fractal algorithm having a fixed geometric replacement rule. These types of fractals have a very artificial appearance are are useful for procedurally generating non-organic models. Potential future investigations for demonstrative purposes could include the application of a noise function as an addition to the recursive gasket generation base function. The full example program is included with the accompanying source material for this report.

In terms of the the application of noise based fractal modelling for the development of the TIG application, one of the main noise generation modules used from NoiseLib was the Ridged-Multifractal noise module (along with the Perlin Noise, Billow and Scale Bias noise modules). The Ridged-Multifractal noise module is very similar to Perlin noise, but the key difference is that the Ridged-Multifractal noise module does not make use of persistence. Instead, it uses the values of previously generated octaves, which are then modified by an absolute value function, and the noise map is generated in a recursive manner. Additionally, the bandwidth of the Ridged-Multifractal noise module is not guaranteed to output values in the range from -1 to 1. This in turn allows for the procedural modelling of ridges based on the generated noise map, although for the TIG application, this noise module is used sparingly.

Another procedurally generated feature of the TIG application is the water texture. The main visual component of the generated water texture is a Voronoi cell based noise map (along with the turbulence and scale point noise modules). Voronoi cells (sometimes refereed to as a Voronoi diagram), are used to produce polygon formations, by randomly scattering “seed points” in the given space coordinates, which are used to form the cells of the Voronoi cell map. A single Voronoi cell unit region contains all the points that are closest to specific seed point. The displacement value of each seed point is pre-set and used with the the random value modifier in the module.
The seed points are placed randomly within each unit space defined in R^n (in the case of the default function R^3, thus the seed points are placed randomly within a unit cube). The frequency value of the seed points is also modified in order to change the distance between the seed points (this determines how tightly packed the cells in the map will appear). Additionally, the displacement value is used to modify the range of random values that are assigned to each cell. The higher the displacement value, the more intense the cell border outlines become. Below are examples of four different generated water maps by the TIG application, each with the same set frequency value of 8, but the displacement value is increased for each, going from 0, 0.25, 0.5 to 1 (clockwise):

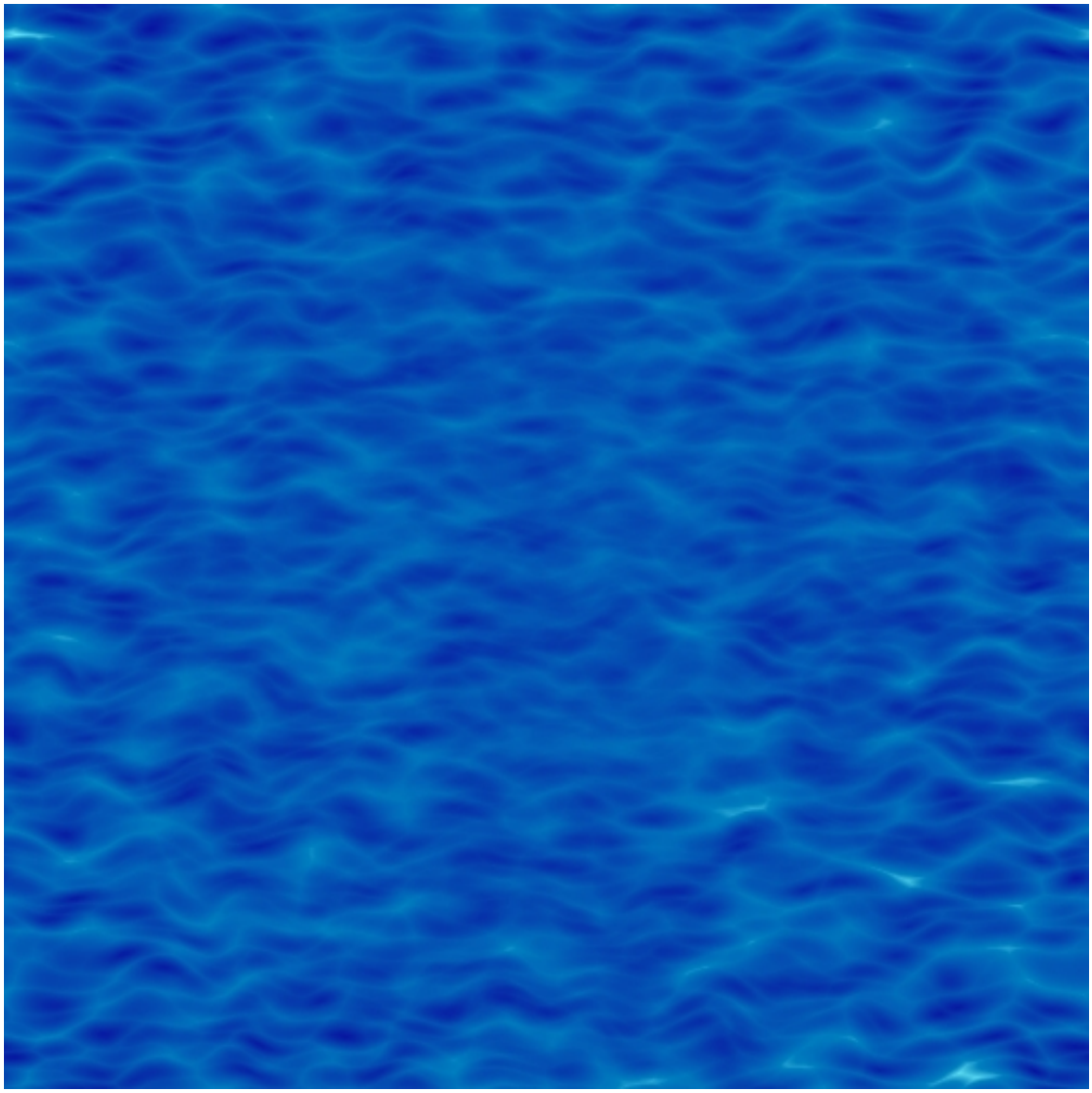





Below are examples of the three main randomly generated procedural textures by the TIG application, along with their pre-set values:



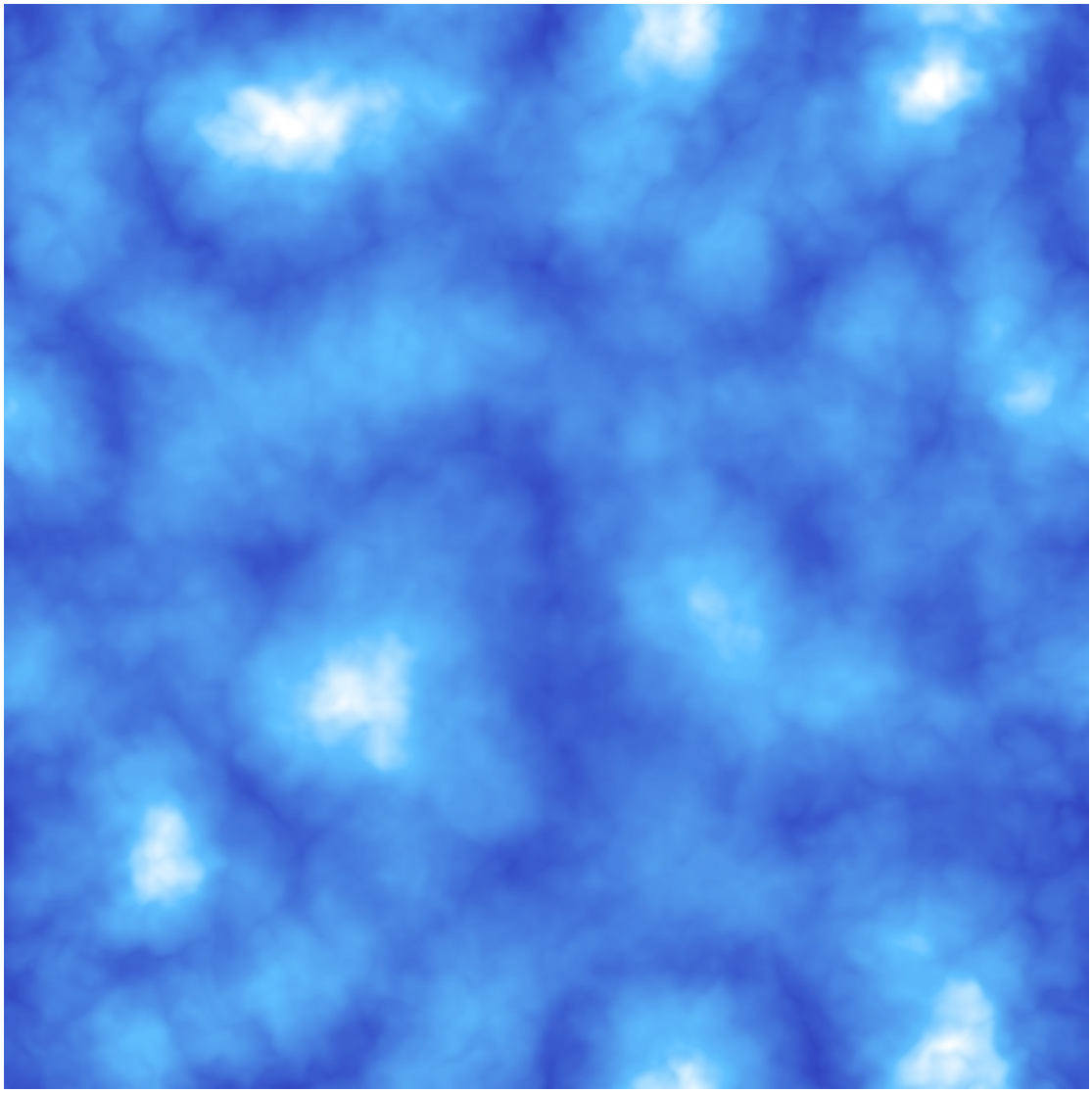
Terrain texture (based on generated heightmap values used to displace the terrain vertex data):

- ***Size: 512x512***
- ***Modules used: Ridged-Multifractal, Billow, Scaled Bias, Perlin Noise***
- ***Octaves: 10***
- ***Base frequency: 2***
- ***Frequency: 0.4***
- ***Persistence: 0.25***
- ***Lower and upper X coordinate values: 10 to 20***
- ***Lower and upper Z coordinate values: 7 to 25***



Water texture:

- ***Size: 512 x 512***
- ***Modules used: Voronoi, Scale Point, Turbulence***
- ***Voronoi frequency: 8.0***
- ***Voronoi displacement: 0***
- ***Turbulence noise power: 1.0 / 32.0***
- ***Turbulence roughness: 1***



Skybox texture:

- ***Size: 512x512***
- ***Modules used: Billow, Turbulence***
- ***Billow frequency: 2.0***
- ***Billow persistence: 0.375***
- ***Billow lacunarity: 2.121***
- ***Billow octave count: 4***
- ***Turbulence frequency: 16.0***
- ***Turbulence power: 1.0 / 64.0***
- ***Turbulence roughness: 2.0***

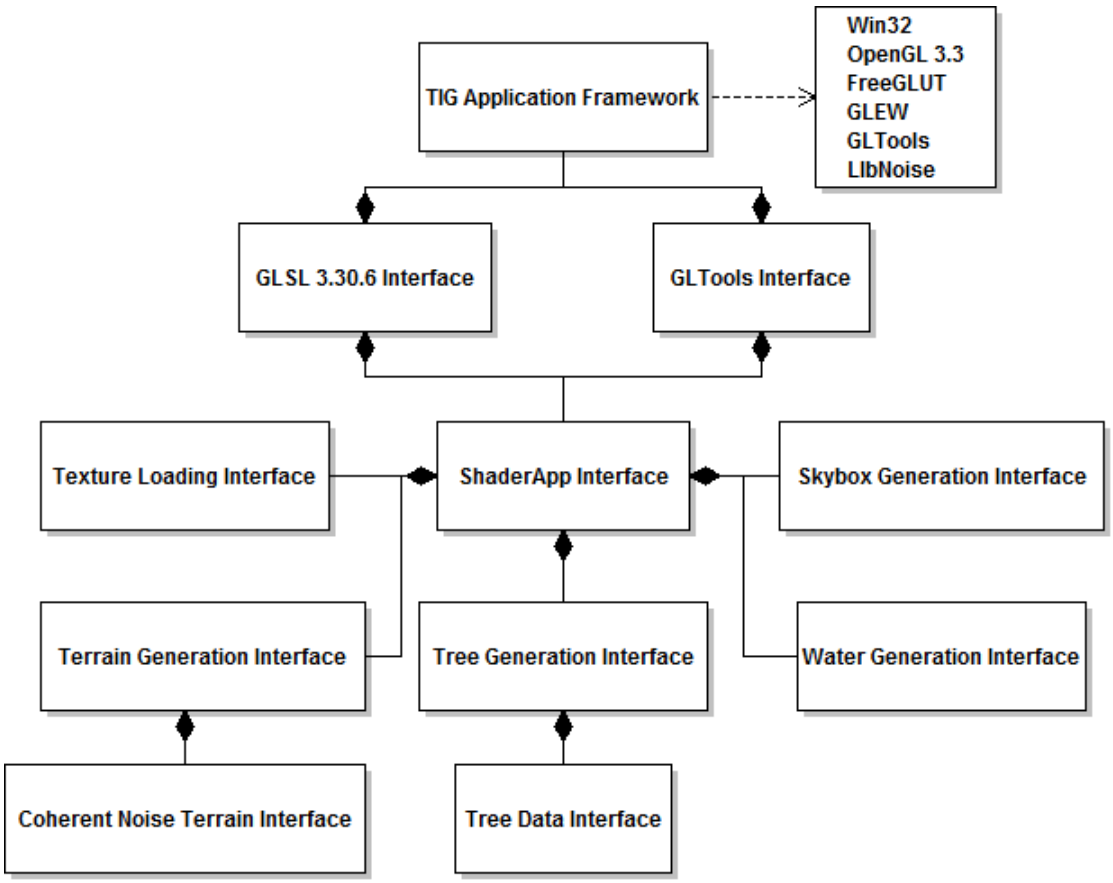
Framework Overview

This section of the report provides a brief overview of the TIG application framework design. The TIG application was developed based on the provided framework, which makes full use of OpenGL 3.3 and interfaces with Win32 as the native windowing environment. The given Framework also has support for DirectInput, but this was not enabled in the TIG application in favour of using the traditional Win32 API input interface.

The following features were designed, developed and tested for the the given framework:

- Implementation of custom GLSL shader based rendering techniques.
- Coherent noise generation module making use of LibNoise.
- Optimized terrain generation interface (custom GLSL shader based rendering).
- Water mesh generation and rendering interface (custom GLSL shader based rendering).
- Skybox generation and rendering interface (custom GLSL shader based rendering).
- Custom camera interface .
- Embedded mesh data format interface for the generation and rendering of palm trees.
- Palm tree generation and rendering interface.
- Expanded texture loading routines to add support for loading and rendering 24bit BMP textures.
- A default terrain generation interface (“dummy interface”), used as a place holder interface for that other terrain generation interfaces inherit their virtual functionality from.

Below is a UML diagram showing the high-level overview of the constructed framework :



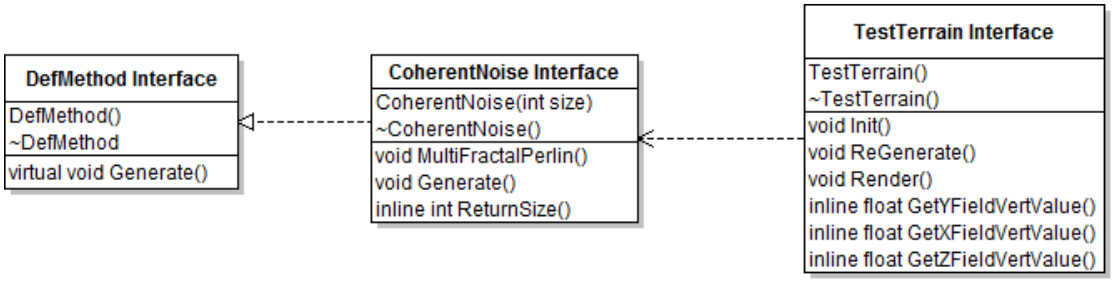
Terrain Generation Implementation

The terrain generation interface consists of three core interfaces:

1) The default terrain rendering interface from which the coherent noise generation interface inherits the class attributes from. Currently, the default terrain interface (called DefMethod.h) is just a place holder.

2) The coherent noise interface (CoherentNoise.h). This interface is used to procedurally generate the terrain displacement and texture data. Makes use of LibNoise.

3) The terrain generation interface (TestTerrain.h). this interface is used to construct the terrain geometry (using the GLBatch interface from GLTools). The constructed terrain geometry can then be rendered using custom GLSL vertex and fragment programs, once it is instanced and initialized in the main ShaderApp.cpp framework implementation.



The terrain data is constructed with the following implementations in the TestTerrain interface:

```
CoherentNoiseTerrain gCoherentNoiseTerrain(512);

TestTerrain::TestTerrain()
{
	//stepsize for texcoord incriment
	float textureCoordIncriment = 1.0
	/gCoherentNoiseTerrain.ReturnSize();

	for(int j=0; j<gCoherentNoiseTerrain.ReturnSize(); j++)
	{
		//i=x
		for(int i=0; i<gCoherentNoiseTerrain.ReturnSize(); i++)
		{
			vTexCoord[i][j][0] = textureCoordIncriment*i;
			vTexCoord[i][j][1] = textureCoordIncriment*j;

			vNormal[i][j][0] = 0.0f;
			vNormal[i][j][1] = 1.0f;
			vNormal[i][j][2] = 0.0f;
		}
	}

	//initialize rand
	srand ( time(NULL) );
}
```

```
void TestTerrain::ReGenerate()
{
	//our vectors to calculate the normals.
	M3DVector3f vector1, vector2, normal;

	////////////////////////////
	//Generate terrain here
	////////////////////////////
	gCoherentNoiseTerrain.MultiFractalPerlin(10, 2, 0.4, 0.25, 10,
20, 7,	25);

	//get the field height
	for(int i = 0; i < gCoherentNoiseTerrain.ReturnSize() ; ++i)
	{
		for(int j = 0; j < gCoherentNoiseTerrain.ReturnSize(); +
+j)
		{
			mYField[(j * gCoherentNoiseTerrain.ReturnSize()) +
i] = gCoherentNoiseTerrain.mfField[(j *
gCoherentNoiseTerrain.ReturnSize()) + i];
		}
	}

	for(int j=0; j<gCoherentNoiseTerrain.ReturnSize(); j++)
	{
		//i=x
		for(int i=0; i<gCoherentNoiseTerrain.ReturnSize(); i++)
		{
			vVertex[i][j][0] = i;
			vVertex[i][j][1] = mYField[(j * 512) + i];
			vVertex[i][j][2] = j;
		}
	}

	//normals.
	for(int j=0; j<gCoherentNoiseTerrain.ReturnSize(); j++)
	{
		//i=x
		for(int i=0; i<gCoherentNoiseTerrain.ReturnSize(); i++)
		{
		  //create vector1	(above subract vector below)
		  m3dSubtractVectors2(vector1, vVertex[i][j+1],
vVertex[i][j-1]);
		  //create vector2	(left subtract right)
		  m3dSubtractVectors2(vector2, vVertex[i-1][j],
vVertex[i+1][j]);
		  //cross product
		  m3dCrossProduct3(normal, vector1, vector2);
		  //normalise
		  m3dNormalizeVector3(normal);

			//set normal
			vNormal[i][j][0] = normal[0];
			vNormal[i][j][1] = normal[1];
			vNormal[i][j][2] = normal[2];
		}
	}
}
```

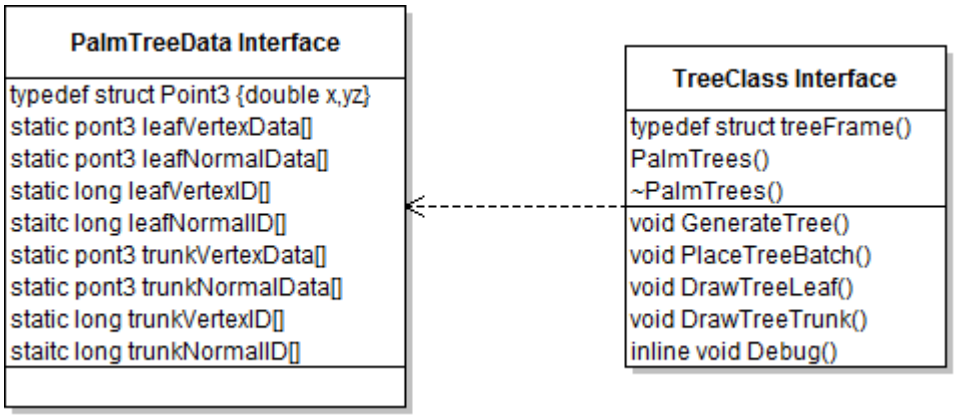
The TestTerrain::Render() function is called in ShaderApp.cpp (using custom GLSL shaders).

Tree Mesh Rendering

One of the main features of the TIG application is that all of the application data is generated by the application, and no external data is used (though external shader programs are used, but these don't really count as data). In order to complement the generated 3D scene, 3D models of palm trees are generated, placed and rendered in the scene.

The palm tree interface consists of two interfaces:

1) The palm tree data interface (PalmTreeData.h). this interface contains the hard coded vertex and normal vector data used to generate the palm tree geometry.
2) The tree generation interface (TreeClass.h), where the data interface from PalmTreeData,h is sued to construct a very simple model frame structure that gets generated into geometry and rendered via the GLBatch interface. Unlike the rest of the geometry interfaces in the TIG application framework, the palm trees are rendered using built in shader programs via the GLTools shaderManager interface.



The palm tree data is generated in much the same way as the terrain data, by passing in the vertex and normal data from the tree model frame into the GLBatch palmLeaves and palmBases objects. The GLBatch objects are then rendered in ShaderApp.cpp, using built in stock shaders from GLTools.
However, the key logical consideration for the generation and rendering of the tree data is how the tree's are placed on the terrain. The PlaceTreeBatch() function from the TreeClass interface is used to randomly place 1024 trees across the the terrain grid, taking in the initial starting function argument variable batchPos as the starting point on the given terrain grid from where batch is placed.
The tree batch can be thought of as a 512x512 square, consisting of 1024 randomly placed trees within in it, that is rendered over the terrain, with the batchPos variable used to determine the origin of the batch square.

```
void PalmTrees::PlaceTreeBatch(float batchPos)
{
	GenerateTree();

	for(int i = 0; i < 1024; ++i)//Set x,y,z coordinates of each
tree
	{
		mX[i] = ((GLfloat)((rand() % 512 + 1 * batchPos)));
		mZ[i] = ((GLfloat)((rand() % 512 + 1 * batchPos)));
		mY[i] = 18.0f;
	}
}
```

However, another logical problem exists once the trees have been placed on the terrain. The initial height of the trees is set to 18.0f units. The problem presented by placing the trees in this manner is that the 1024 trees will be drawn over the terrain, regardless of the of whether or not they are being drawn over the island terrain mesh, or the island water mesh.
Leaving this problem unfixed would result in unrealistic looking terrain, with palm trees being placed over water.

Fortunately a simple solution was implemented that would solve this problem. In ShaderApp.cpp, where the palm trees get drawn and when the palm trees first get initialized, another function (part of the ShaderApp interface), called CalcTreePlacement is called:

```
void ShaderApp::CalcTreePlacement()
{
	gTrees.PlaceTreeBatch(0.01f);

	for(int i = 0; i < 1024; ++i)
	{

	if(g_Terrain.GetYFieldVertValue(gTrees.mX[i],gTrees.mZ[i])  >
	   14) //current elevation restriction is 14. Trees placed on
	any point in the map below the value of 12 won't get rendered.
		{
			treeValid[i] = true;
		}
		else
		{
			treeValid[i] = false;
		}
	}
}
```

The CalcTreePlacement() function takes the x and z coordiantes of each placed tree in the grid, and compares the height of the point on the terrain mesh where the given tree is placed. If the elevation of the given terrain point is greater then 14.0f units (well above the water level), the treeValid[] static boolean array variable for that tree is set to true. The treeValid[] static boolean array variable has 1024 elements, with each element representing the drawing condition of each tree. If the placed tree intersects the pre-set valid height value on the given terrain grid point, then that tree can be drawn, or otherwise it doesn't get drawn.

```
//First the palm leaves
	for(int i = 0; i < MAXTREES ; ++i)
	{
		TreeMVMatrix.PushMatrix();
		TreeMVMatrix.Translate(gTrees.mX[i], gTrees.mY[i],
gTrees.mZ[i]);
		TreeMVMatrix.Scale(0.25f, 0.25f, 0.25f);

		shaderManager.UseStockShader(GLT_SHADER_POINT_LIGHT_DIFF,
treePipeline.GetModelViewMatrix(),

	treePipeline.GetProjectionMatrix(), vLightEyePos, vLeafColor);
		if(treeValid[i])
		{
			gTrees.DrawTreeLeaf();
		}

		TreeMVMatrix.PopMatrix();
	}
```

```
	//Then the trunk
	for(int j = 0; j < MAXTREES ; ++j)
	{
		TreeMVMatrix.PushMatrix();
		TreeMVMatrix.Translate(gTrees.mX[j], gTrees.mY[j],
gTrees.mZ[j]);
		TreeMVMatrix.Scale(0.25f, 0.25f, 0.25f);

		shaderManager.UseStockShader(GLT_SHADER_POINT_LIGHT_DIFF,
treePipeline.GetModelViewMatrix(),

	treePipeline.GetProjectionMatrix(), vLightEyePos, vTrunkColor);
		if(treeValid[j])
		{
			gTrees.DrawTreeTrunk();
		}

		TreeMVMatrix.PopMatrix();
	}
```

Water Mesh Rendering

The TIG application also demonstrates the rendering of simple water bodies using procedurally generated textures and animated displacement of water. The water mesh is part of the Water.h interface. The water mesh itself is just a 512x512 triangle plane, and is constructed in the same way as the rest of the geometry data in the TIG application.

The interesting part of the water rendering feature is that the water body is rendered and displaced using a custom GLSL vertex shader. The vertex shader continually displaces the water mesh data, per-vertex and per-frame, using a simple sine wave equation.

The sine wave equation is defined as:

$$\vec{WaterVetrex}_y = \mathbf{sin}(0.5 \times WaterVetrex_z + \Delta Time * WaveSpeed) \times 0.5$$

This is implemented in the Water.vp vertex shader :

```
#version 130

in vec4 vVertex;
in vec3 vNormal;
in vec4 vTexture0;

uniform float fTime;
uniform float fWaveSpeed;
uniform mat4   mvpMatrix;
uniform mat4   mvMatrix;
uniform mat3   normalMatrix;
uniform vec3   vLightPosition;

smooth out vec3 vVaryingNormal;
smooth out vec3 vVaryingLightDir;
smooth out vec2 vTexCoords;
```

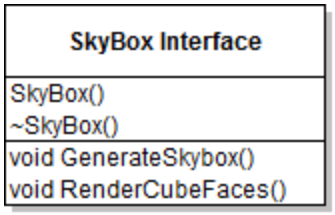
```
void main(void)
{
      vec4 v = vec4(vVertex);

      v.y = sin(0.5*v.z + fTime*fWaveSpeed)* 0.5;

    vVaryingNormal = normalMatrix * vNormal;

    // Get vertex position in eye coordinates
    vec4 vPosition4 = mvMatrix * vVertex;
    vec3 vPosition3 = vPosition4.xyz / vPosition4.w;

    // Get vector to light source
    vVaryingLightDir = normalize(vLightPosition - vPosition3);

    // Pass along the texture coordinates
    vTexCoords = vTexture0.st;

    gl_Position = mvpMatrix * v;
}
```

And the fragment shader is used to calculate the fragment colour based on the procedurally generated water texture (note that the specular component of the final colour is not calculated):

```
#version 130

out vec4 vFragColor;

uniform vec4      ambientColor;
uniform vec4      diffuseColor;
uniform sampler2D colorMap;

smooth in vec3 vVaryingNormal;
smooth in vec3 vVaryingLightDir;
smooth in vec2 vTexCoords;

void main(void)
    {

    // Dot product gives us diffuse intensity
    float diff = max(0.0, dot(normalize(vVaryingNormal),
normalize(vVaryingLightDir)));

    // Multiply intensity by diffuse color, force alpha to 1.0
    vFragColor = diff * diffuseColor;

    // Add in ambient light
    vFragColor += ambientColor;

    // Modulate in the texture
    vFragColor *= texture(colorMap, vTexCoords);

    }
```

Sky Box Rendering

The final main feature of the TIG application is the procedural generation and rendering of the skybox texture. The skybox texture is loaded into the application as a six-sided cubemap. The cubemap uses the same single tiling texture that gets generated when the application is initialized. The skybox generation interface (SkyBox.h), is used to generate the skybox texture.



The generated skybox textures then get loaded into the application from ShaderApp.cpp:

```
const char *szCubeFaces[6] = { "../data/skyTex.bmp",
"../data/skyTex.bmp", "../data/skyTex.bmp", "../data/skyTex.bmp",
"../data/skyTex.bmp", "../data/skyTex.bmp" };
GLenum  cube[6] = {  GL_TEXTURE_CUBE_MAP_POSITIVE_X,
                     GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
                     GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
                     GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
                     GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
                     GL_TEXTURE_CUBE_MAP_NEGATIVE_Z };
      glBindTexture(GL_TEXTURE_CUBE_MAP, cubeTexture);

    for(int i = 0; i < 6; i++)
    {
       LoadBMPCubeTexture(szCubeFaces[i], cube[i]);
    }
```

A custom GLSL shader is also used for the rendering of the skybox. The GLSL skybox vertex shader transforms the vertex position of the skybox based on the skybox transformation pipeline model view projection matrix, while the fragment shader computes the colour of each of the fragments based on the texture data that is sampled from the cubemap.

```
//Vertex shader
#version 130

in vec4 vVertex;
uniform mat4   mvpMatrix;  // Transformation matrix
varying vec3 vVaryingTexCoord;

void main(void)
{
    vVaryingTexCoord = normalize(vVertex.xyz);
    gl_Position = mvpMatrix * vVertex;
}

//Fragment shader
#version 130

out vec4 vFragColor;
uniform samplerCube  cubeMap;
varying vec3 vVaryingTexCoord;

void main(void)
{
    vFragColor = texture(cubeMap, vVaryingTexCoord);
}
```

Camera Interface

A camera interface was one of the main requirements for the development of the the TIG application. A camera object that would allow the user to navigate around the 3D scene had to be implemented. Also, the camera object would have to allow the user to walk on the terrain and move around the scene in an FPS (First Person Shooter) style.

The camera interface that was implemented makes use of math3d.h M3DMatrix44f interface, as well as the GLFrame interface (both are part of GLTools). The camera interface allows the user to fly around the scene using the keyboard arrow keys, to walk on the terrain using the WASD keys (this includes strafing) and to rotate the camera around the XYZ axis using the mouse. The initial camera interface had a problem however, as there is an issue with using the GLFrame interface for performing matrix based rotations around the local camera axis. This is due to the fact that the GLFrame interface makes use of Euler angles to perform the rotations around the rotation axes. This in turn causes “Gimbal lock”. Gimbal lock occurs when two rotation axes are rotated in parallel directions, making the third rotation axis slip. In the TIG application, this was happening as the user was rotating the camera along the X and Y axis, thus causing the Z axis to slip on either of of it's positive or negative sides. This in turn caused the camera to roll.

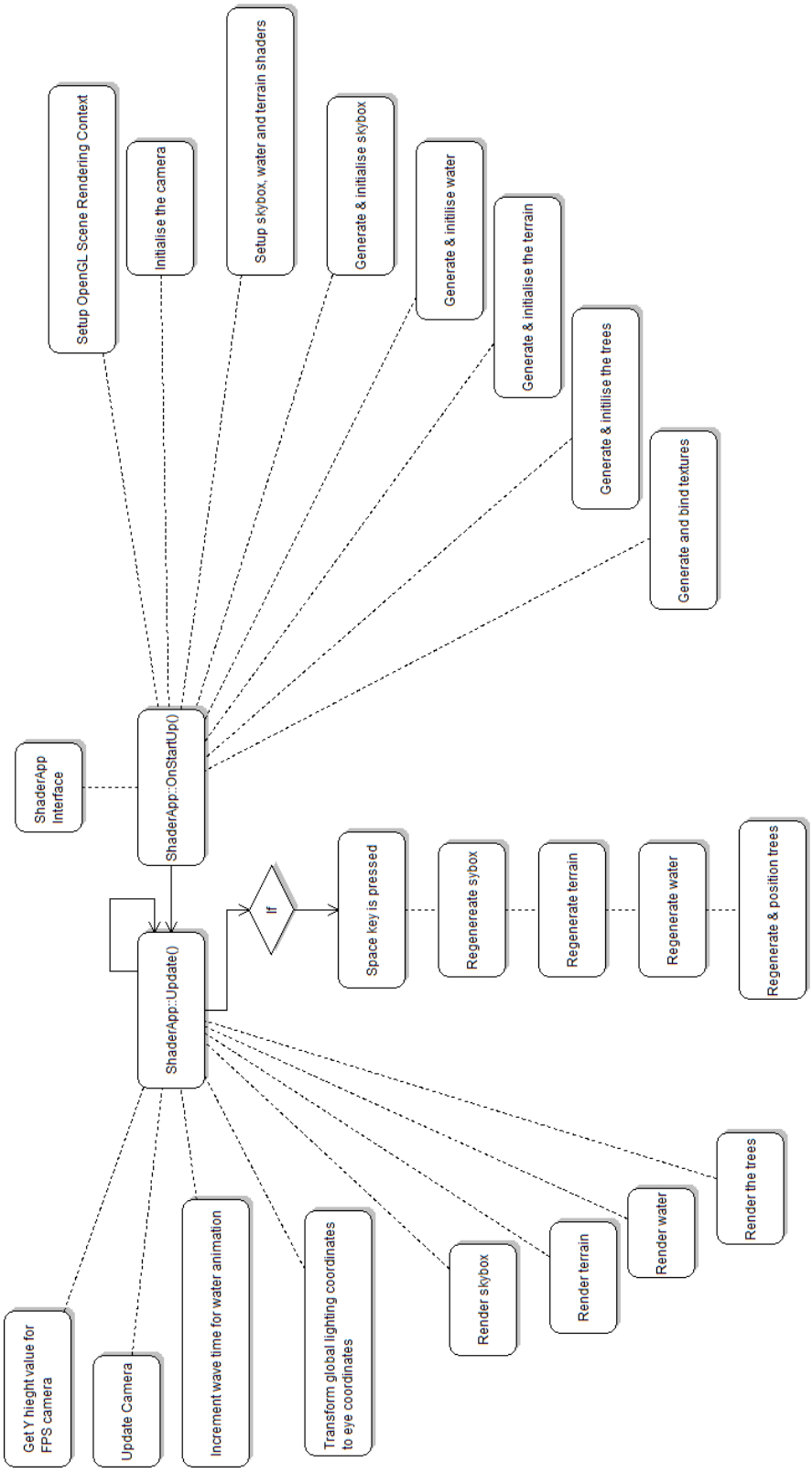
The Gimbal lock issue was addressed by by performing the rotations along the X axis using the local rotation matrix, while performing rotations around the Y axis using world matrix. This way, there is no Gimbal lock, as the the X and Y rotations are never parallel within the same coordinates set. Future releases will eliminate this completely by replacing the local Euler angle rotation routines with a Quaternion based rotation interface. This allows the camera to rotate independently along any of the three axes.
The final feature of the camera interface is allowing the user to walk on the terrain (rather then just being able to fly around it). This feature allows the user to get a better look at the scenery details. The way FPS walking was implemented is by translating the local camera Y coordinate every frame to a pre-set Y value, that is calculated based in the current height the user is above at a given vertex point on the terrain mesh. Below is an example of how this function is implemented in the camera interface:

```
void Camera::FPSWalk(float maxHeight, float height)
{
	float linear = 0.5f;
	if((GetKeyState('W') & 0x80))
	{
		m_cameraFrame.MoveForward(linear);
		m_cameraFrame.TranslateLocal(0.0f, maxHeight - GetY() + height,
0.0f);
	}
	else if((GetKeyState('S') & 0x80))
	{
		m_cameraFrame.MoveForward(-linear);
		m_cameraFrame.TranslateLocal(0.0f, maxHeight - GetY() + height,
0.0f);
	}
	else if((GetKeyState('D') & 0x80))
	{
		m_cameraFrame.MoveRight(-linear);
		m_cameraFrame.TranslateLocal(0.0f, maxHeight - GetY() + height,
0.0f);
	}
	else if((GetKeyState('A') & 0x80))
	{
		m_cameraFrame.MoveRight(linear);
		m_cameraFrame.TranslateLocal(0.0f, maxHeight - GetY() + height,
0.0f);
	}
	GetMouseInput();
}
```

Application Features

Program Operation

Below is a UML diagram showing the operation of the TIG application:



BMP Texture Data Loading

In addition the loading functionality of the TIG application framework being able to load TGA texture files, two additional functions were added to the texture loading interface (texloader.cpp) to allow the loading of 24bit BMP files, which is the primary texture format used for all texture data in the application:

```
bool LoadBMPTexture(const char *szFileName, GLenum minFilter, GLenum
magFilter, GLenum wrapMode)
{
       GLbyte *pBits;
       int nWidth, nHeight;

       pBits = gltReadBMPBits(szFileName, &nWidth, &nHeight);
       if(pBits == NULL)
              return false;

       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrapMode);
       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrapMode);

       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minFilter);
       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magFilter);

       glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nWidth, nHeight, 0,
        GL_BGR, GL_UNSIGNED_BYTE, pBits);
    free(pBits);
    if(minFilter == GL_LINEAR_MIPMAP_LINEAR ||
       minFilter == GL_LINEAR_MIPMAP_NEAREST ||
       minFilter == GL_NEAREST_MIPMAP_LINEAR ||
       minFilter == GL_NEAREST_MIPMAP_NEAREST)
        glGenerateMipmap(GL_TEXTURE_2D);

       return true;
}

bool LoadBMPCubeTexture(const char *szFileName, GLenum  cube)
{
       GLbyte *pBits;
       int nWidth, nHeight;

       pBits = gltReadBMPBits(szFileName, &nWidth, &nHeight);
       if(pBits == NULL)
              return false;

       glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
GL_CLAMP_TO_EDGE);

    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

       glTexImage2D(cube, 0, GL_RGB, nWidth, nHeight, 0,
                            GL_BGR, GL_UNSIGNED_BYTE, pBits);
    free(pBits);

       glGenerateMipmap(GL_TEXTURE_CUBE_MAP);

       return true;
}
```

Critical Analysis

The creation of the TIG application was a learning experience with a fairly steep curve. Thus, this code that was developed is not optimized for performance and there are a lot of things that could have been, and could still be, done better and in a more efficient manner.

LibNoise is not ideally optimized for real time 3D application usage, but is sufficient for demonstrative purposes. Therefore, the procedural generation of each island at run time can take up to 30 seconds (20 seconds on the computer the application was developed on, using a 2.5GHz Intel Core 2 Quad CPU). This is one of the things to be addressed and potentially optimized in the future releases of the application (or other applications that follow the same principals as TIG).

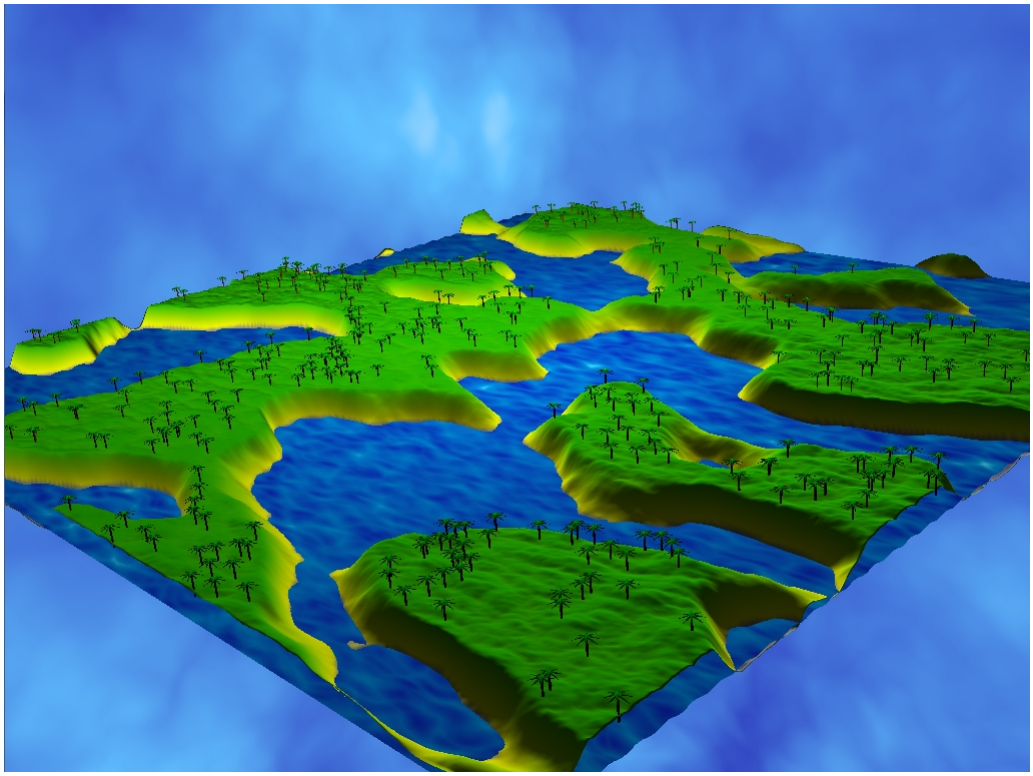
The rendering of mesh data is not very efficient either, since the use of VBO's and geometry shaders was omitted due to time constraints. As a result, 100% of the procedural generation computations are done on CPU, rather then on the GPU. This includes the procedural generation of textures and mesh data (with the exception of the water mesh animation which is done via the vertex shader on the GPU). For the future releases of the application, all of the procedural data generation should be moved to the GPU for more efficient computation.

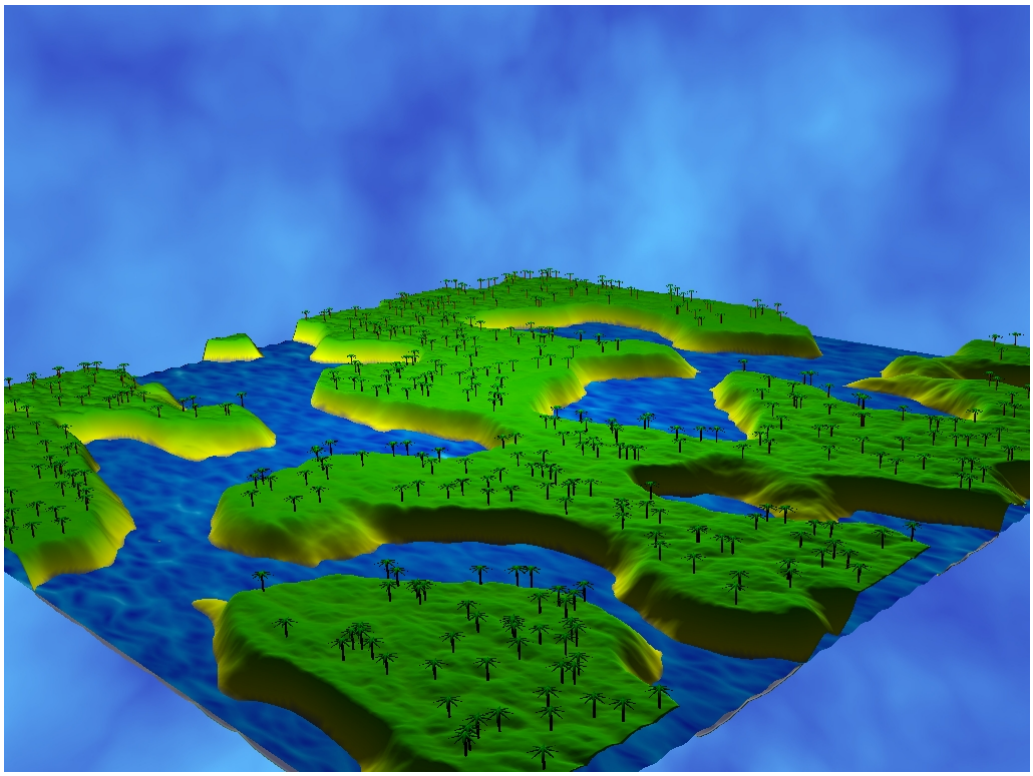
The use of multitexturing was also omitted, mainly because it would take much longer for the application to generate the additional textures. Also there was not really any need for multitexturing, as the single generated terrain texture is of fairly good visual quality and has pre-computed gradient colour data, based on terrain elevation, already rendered to it.

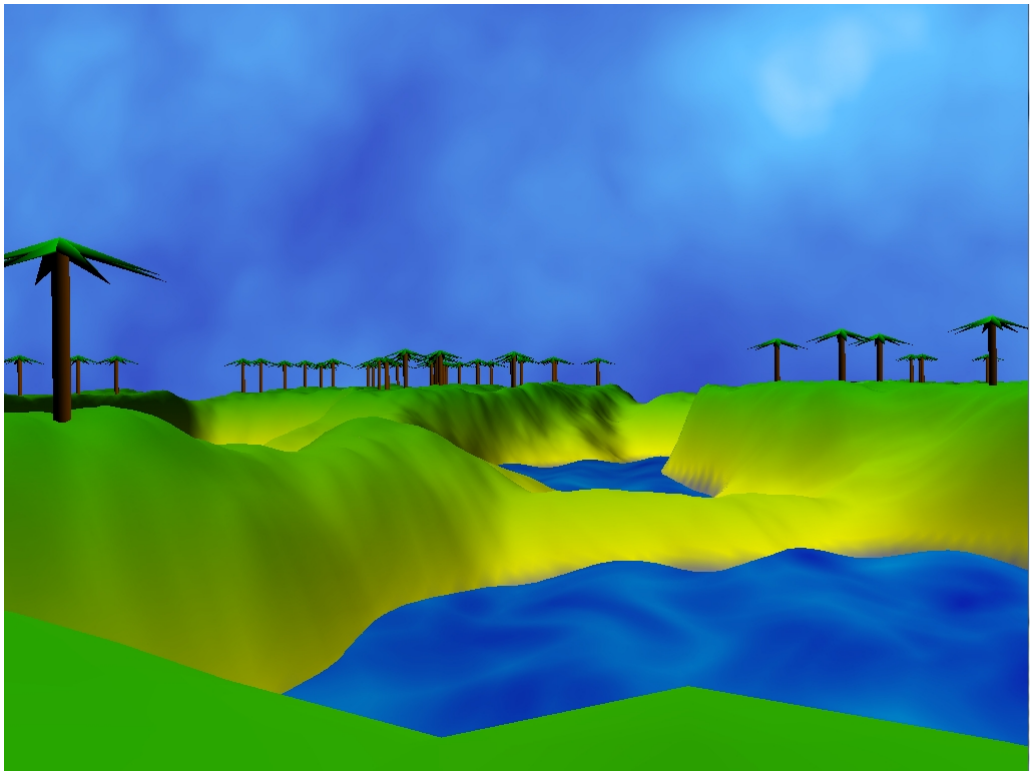
Conclusion

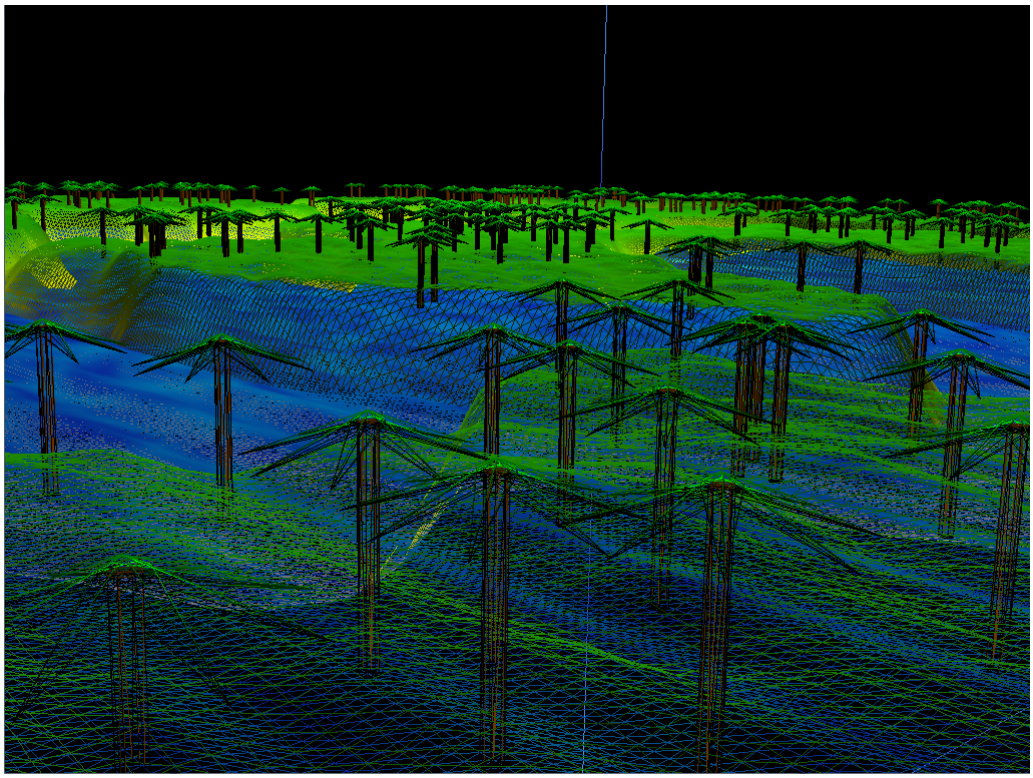
The development of the TIG application proved to be a rewarding learning experience. The exposure to procedural generation algorithms allowed for the investigation, development and testing of various methods used in many of the cutting edge 3D rendering applications found in today's games, interactive applications, film and other visual media.

The exposure the OpenGL 3.3 also proved to be a worthwhile experience, as in comparison to OpenGL 2.0 and 1.2, it's almost a completely different graphics API. The use of OpenGL 3.3 has opened the doors for the use of OpenGL ES 2.0 for future personal 3D graphics research and programming.









References

Ebert, S. D et al. 1998. *Texturing & Modeling: A Procedural Approach, Second Edition*. AP Professional.

Wright, R. S, Jr. Lipchak et al. 2011. *OpenGL SuperBible, Fifth Edition*. Addison-Wesley Publishing Company.

Rost, J. R et al. 2008. *OpenGL Shading Language, Second Edition*. Addison-Wesley Publishing Company.

Angel, E. 2000. *Interactive Computer Graphics: A Top-Down Approach With OpenGL*. Addison-Wesley Publishing Company.

DeLoura, M et al. 2000. *Game Programming Gems*. Charles River Media.

DeLoura, M et al. 2001. *Game Programming Gems 2*. Charles River Media.

Van Verth, M. J. Bishop, M. L. 2008. *Essential Mathematics for Games & Interactive Applications: A Programmer's Guide, Second Edition.* Morgan Kaufmann.

Akenine-Moller, T. Haines, E. Hoffman, N. 2008. *Real-Time Rendering - Third Edition*. A K Peters, Ltd.

Hearn, D. Baker , P. M. 1994. *Computer Graphics – Second Edition*. Prentice Hall.

Perlin, K. 1999. *Making Noise.* Available from: http://www.noisemachine.com/talk1/ [Accessed 05 May 2011].