4/6/2011

Progeny

Final Report

Shawn Josey
Chad Levesque
Robert Lockyer
Andrew Carter

Executive Summary

Game development in the past few years has seen an explosion in the size of virtual worlds and an explosion in the complexity of characters and other graphical assets. The amount of information required to represent a virtual environment is growing at an astounding rate and so are the requirements for dealing with this level of detail. Video games budgets have ballooned and install sizes have grown rapidly. All this is happening at a time when digital distribution and mobile computing are becoming increasingly popular.

Project Progeny aims to demonstrate how procedural methods can be used in order to combat these developing trends, while still enabling gaming companies to deliver vast, highly detailed game worlds.

Progeny demonstrates these capabilities by generating fully scalable planets which users can rotate at an orbital level, and explore in a free look mode. This sort of scale would be very difficult to accomplish with traditional methods. For large galaxies of these planets it would be completely infeasible.

Two key features of the project are the use of an algorithm for generating a renderable subset of the available surface data, and a procedural technique for generating this surface data for the planet.

In order to generate the surface data procedurally Progeny makes use of a coherent noise library “libnoise”. Coherent noise has the properties of being deterministic, continuous and computable in *n* dimensions. Coherence allows us to use it to generate smooth planet like topologies which are reproducible using a few simple input parameters. The fact that it can be sampled in at least three dimensions means that it is capable of being used to generate topologies which do not have to be wrapped. Thus, distortion and seams are avoided altogether.

Progeny makes use of an algorithm called Real-time Optimally Adapting Meshes (ROAM) for sampling the noise function and generating a manageable set of triangles which are

rendered to the screen to generate a scene that is optimized for visual fidelity and performance. The algorithm tries to distribute triangles in a way that best represents the underlying noise data as a continuous surface. Flat areas require very few triangles to represent accurately, while more turbulent areas require dense meshes.

By implementing the design described in this report we have created a reusable procedural content generation library. This library is capable of randomly generating large scale planets with fine grained details. It also enables users with no prior knowledge to easily generate complex planetary geographies with adjustable attributes. These capabilities demonstrate the power of procedural generation, since it would take highly skilled artists a significant amount of time to generate planets at similar levels of detail using traditional methods.

Table of Figures

Figure 1: Procedurally generated Earth-like planets (Eve Online: Tyrannis) [MANN2010] .. 2
Figure 2: Left: Output of a one-dimensional non-coherent noise function. Right: Output of a one-dimensional smoothed coherent noise function. [LIB2007] 4
Figure 3: First six levels of triangle splitting [DUCH1997] 6
Figure 4: Project Dependency Layout 7
Figure 5: Progeny Interfaces, and their implementations in ProgenyDemo 8
Figure 6: A planet is associated with a color scheme and a terrain generator. 10
Figure 7: The terrain and colors of the planet are accessed through a planet strategy, given to the planet instance at construction. 11
Figure 8: diagram of the class structure supporting Progeny's ROAM implementation. 12
Figure 9: The surface of a triangle mesh as two neighboring triangles are split and merged. 13
Figure 10: A ROAM-split surface and the same surface viewed by a virtual third person camera. 14
Figure 11: Noise / Planet Relationship 16
Figure 12: Class Diagram - How Progeny Modules implement libnoise Module Interface 17
Figure 13: Voronoi Cells, Clamped Cells, Inverted Clamped Edges, Finished Craters 18
Figure 14: Turbulent Craters 18
Figure 15: Perlin Noise, Ridged Multi-fractals 19
Figure 16: Progeny Controls 20
Figure 17: Progeny Xbox Controls 21
Figure 18: Notification Pop-up 21
Figure 19: Information Bar 22
Figure 20: Customization Menu 23
Figure 21: Simplified Camera System Class Diagram 24
Figure 22: Sprint Breakdown 26

Contents

Executive Summary ii

Table of Figures iv

1.0 Introduction 1

1.1 Summary and Project Concept 1

1.1.1 Procedural Generation Background 1

1.1.2 Project Concept 3

1.1.3 The Project Epic 3

1.2 Technical Background and Research 4

1.2.1 Coherent Noise 4

1.2.2 Level of Detail 5

1.2.3 Real-time Optimally Adapting Meshes (ROAM) 5

2.0 Technical Design 6

2.1 System Overview 6

2.1.1 Progeny 7

2.1.2 ProgenyDemo 7

2.1.3 Horizon 8

2.1.4 libnoise 8

2.2 Progeny Technical Design 8

2.2.1 Portability 8

2.2.2 Extensibility 9

3.0 Implementation Details 10

3.1 Progeny 10

3.1.1 Progeny Class Descriptions 10

3.1.2 Continuous Level of Detail - ROAM 12

3.2 Progeny Demo 19

3.2.1 User Interface 19

3.2.2 Controls 19

3.2.3 Notifications and overlays 21

3.2.4 Customizing planets 22

3.2.5 Camera System 23

4.0 Final Project Schedule 25

4.1 Scrum 25

4.1.1 Term Three - Ceres Release 26

4.2 Project Management Post Mortem 29

4.2.1 How the Scrum approach has evolved 30

4.3 Success and Challenges 31

5.0 Project Budget and Resources 33

6.0 Moving Forward 34

7.0 Conclusion 35

Glossary 37

References 38

1.0 Introduction

1.1 Summary and Project Concept

1.1.1 Procedural Generation Background

Often in digital 3D mediums, such as video games, art assets are created manually by a 3D modeler and a texture artist. Although these methods provide a great deal of control over the artwork, that control comes at a substantial human resource cost. Art assets are also the primary factor affecting the distribution size of games. This is a growing concern in areas such as mobile gaming markets and digitally distributed content. Depending on the scale of the game, manual creation of content could be practically impossible. For instance, the game world could be an entire galaxy where any star system could be visited by the player.

Procedural generation is the algorithmic creation of such assets as opposed to manual pre-generation. It allows the user of the algorithms to dynamically create content that meets given criteria in geometry and appearance. This allows for games to be smaller in distribution size by generating areas of the game world as they're needed (e.g. based on the player's actions and exploration patterns.)

Procedural art generation was first used in early computer games as a way of including vast amounts of art assets in a small executable. Storage space and small memory limits of early computing platforms were major limiting factors in early game design. The video game Elite included procedural planet generation. It generated a large universe from a single seed value. This universe contained 8 galaxies including 256 stars each, though the creators claimed the engine to be entirely capable of generating 2^48 galaxies [SPUF2003]. That sort of scale would be impossible to achieve in a game where art assets were designed one by one, as in traditional games.

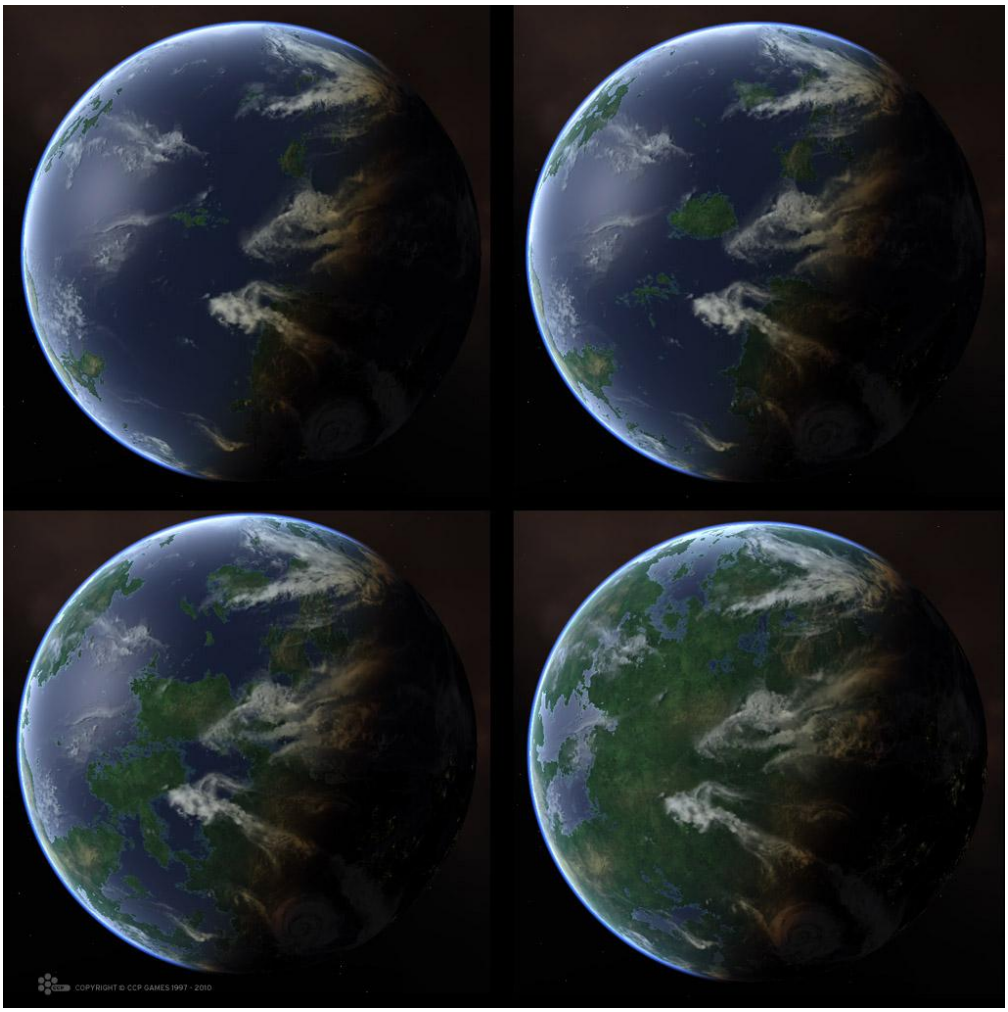

Figure 1: Procedurally generated Earth-like planets (Eve Online: Tyrannis) [MANN2010]

Recently, a new wave of games has tried to extend the concept of Elite using modern computers and techniques. The massively multiplayer game Eve Online previously made all planets in their game universe by hand, but recently started using procedural generation in order to make a vast number of highly detailed plants [MANN2010]. Figure 1 shows how different inputs can create different planets that all have a very convincing and realistic appearance. Yet Another Space Shooter (YASS) by Matthias Dandorff [DAND2008] uses procedural generation to dynamically construct planets and surface textures as needed and present the player with a seamless solar system.

1.1.2 Project Concept

Progeny is designed as a reusable framework for the procedural generation of objects and textures. Its primary focus is on the generation of scalable planetary bodies for real-time rendering systems. To define the scope of the project, an epic was written to illustrate what is desired for the end of project demonstration.

1.1.3 The Project Epic

The demonstration station has a computer running Source Studio's game engine, Horizon. Without user interaction, the system continues to generate and display a planet with random values for size, terrain variation, and class (terrain, desert, lunar, etc.) An on-screen tip also prompts the user to press a key to open the object creation menu.

An intrigued person walks up to the computer and presses a key to open the menu. He or she is presented with a menu to customize the planet parameters. The user can adjust several sliders controlling the values such as ocean height, continent frequency, mountain frequency and planet radius.

After the user customizes the options as desired, he or she clicks generate. This closes the menu and a randomly generated planet matching the given values is presented. The camera is looking at the planet from an orbital height.

To inspect the planet, the user can use the keyboard or mouse controls to navigate the orbital camera. This camera can be spun around the planet or zoomed with a smooth inertial movement. As the user zooms in or out, the geometry and textures of the planet adapt to the appropriate level of detail and the scene maintains a frame rate of at least 30 frames per second. Additionally, the user can press a key to switch to a free look camera for flying over the planet surface.

The user can seamlessly navigate from surface level to extra-orbital heights and back again to any area on the planet's surface. The surface itself is colored at least according to terrain elevation.

1.2 Technical Background and Research

1.2.1 Coherent Noise

To generate a randomly varying procedural height map, an irregular primitive function is needed [EBRT2002]. These primitive functions are often called *noise* functions. White noise, for example, provides random numbers independent of frequency. However, a white noise function is non-deterministic, i.e. it will not produce the same results for a given set of inputs. If used in terrain generation, a planet's appearance could vary greatly from frame to frame, or even with camera position. Clearly something more consistent is desired.

To create a cohesive scene in a real-time graphics system, we need a coherent noise function that meets the following requirements [LIB2007]:

- Passing in the same input value will always return the same output value.
- A small change in the input value will produce a small change in the output value.
- A large change in the input value will produce a random change in the output value.

An illustration of a function that fulfills these requirements is compared with white noise below in Figure 2.

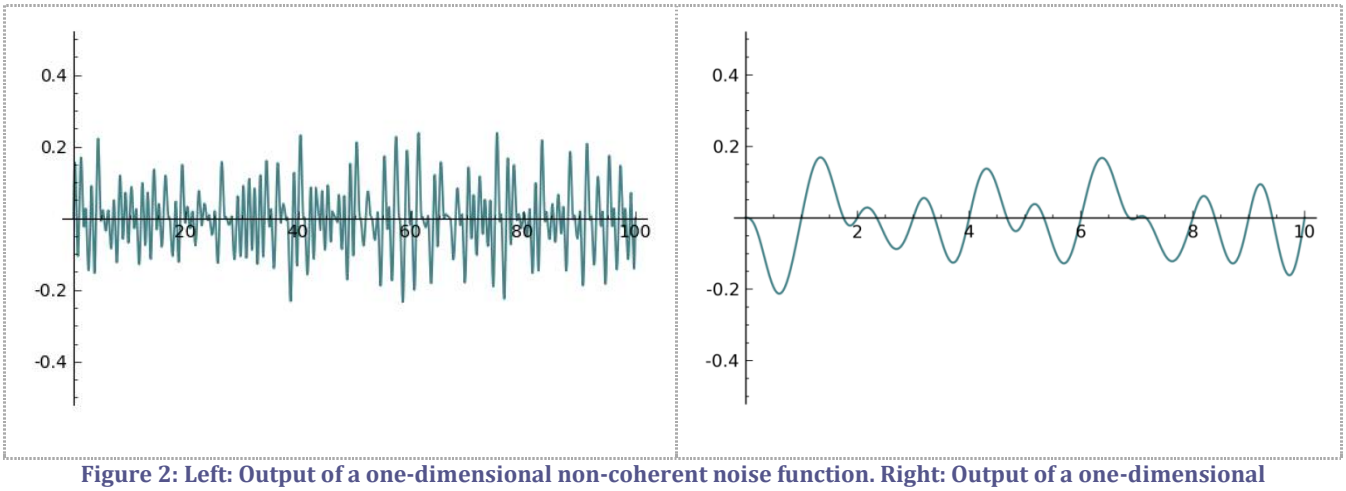

Figure 2: Left: Output of a one-dimensional non-coherent noise function. Right: Output of a one-dimensional smoothed coherent noise function. [LIB2007]

libnoise [LIB2007] is a portable C++ library capable of generating various types of coherent noise such as Perlin noise and ridged multi-fractal. These noise functions are often used in

procedural generation algorithms to create and texture planetary terrain. It was also found that others succeeded in using libnoise for similar applications [DAVX2009].

The libnoise website also contains several tutorials describing methods of using libnoise to create height maps for planar and spherical surfaces. Consequently, libnoise was chosen as the foundation for Progeny's procedural core.

1.2.2 Level of Detail

Progeny is intended to be used in real-time applications such as interactive video games. As an entire planet is naturally composed of a very large data set of geometry and textures, it is prohibitive to store all potentially required data in memory, or to render the object at its maximum sampling frequency and maintain an acceptable frame rate.

It is evident that a dynamic level of detail system is required to realize the project epic wherein the user can seamlessly zoom from orbital distances to any area on the planet's surface.

1.2.3 Real-time Optimally Adapting Meshes (ROAM)

In 1997, Mark Duchaineau et al. at the Los Alamos National Laboratory and the Lawrence Livermore National Laboratory published an algorithm they developed for terrain visualization entitled Real-time Optimally Adapting Meshes, or ROAM [DUCH1997].

ROAM is an algorithm for constructing flexible triangle meshes to render high frequency elevation maps with controllable error bounds at high frame rates. The algorithm uses dual priority queues to manage split (see Figure 3 for illustration of split levels) and merge operations that execute on triangles stored in a binary tree. After the mesh is updated, the renderer is called to draw the leaf nodes of the binary tree, representing the smallest triangles needed to adequately represent the elevation map in a given area.

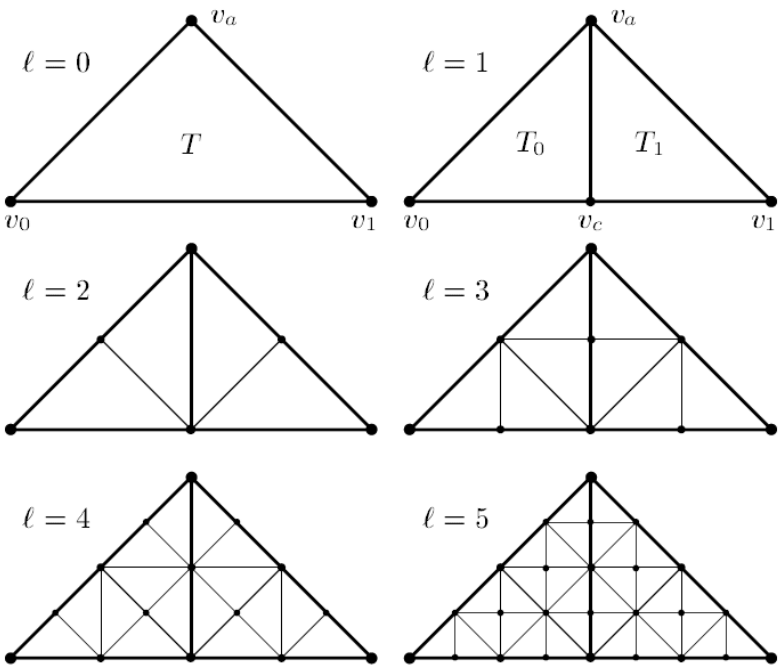

Figure 3: First six levels of triangle splitting [DUCH1997]

This allows the application to represent the scene at a constant number of triangles independent of frame content and camera position.

ROAM has been empirically proven to provide an effective solution to the problem of terrain level of detail and much additional literature can be found on the subject [BTR2000, ONEIL2001, POM2000].

2.0 Technical Design

2.1 System Overview

Project Progeny was designed as the combination of two main components. The first component named "Progeny," acts as the core library and is made independent of any specific graphics library. The second component, named "ProgenyDemo," acts as an example of how the Progeny library could be used in an existing game engine.

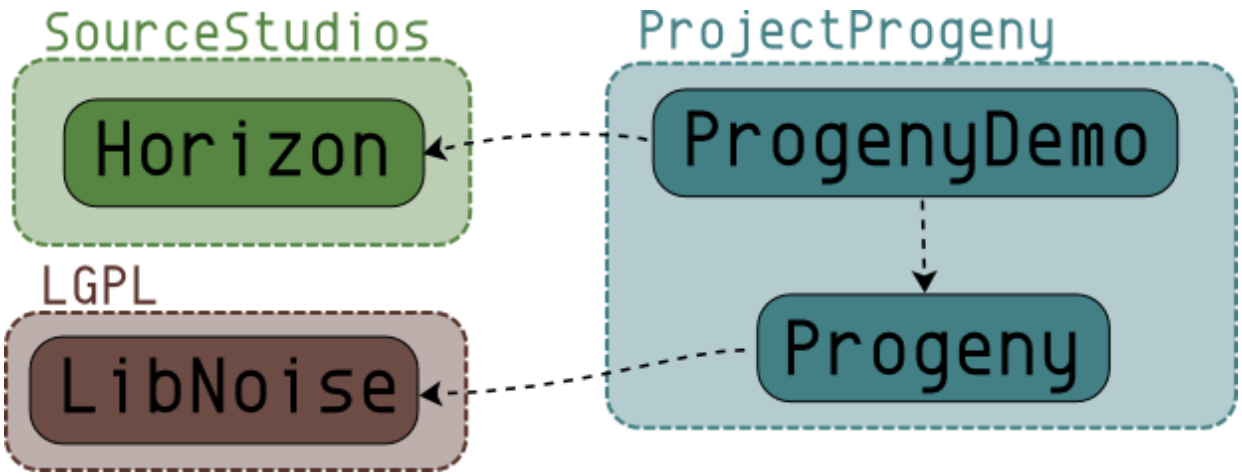

Figure 4: Project Dependency Layout

Progeny has been designed to be easily integrated into the pipeline of an existing game, providing it with a simple interface for procedurally generating complex worlds. Illustrated in Figure 4 are the four main modules contained in the project. Careful consideration has been given to how these modules interact with each other.

2.1.1 Progeny

The Progeny core module is designed to be completely abstracted from any specific rendering system. It is implemented as a C++ library for concerns of speed and fine grained resource control. It uses the free and open-source libnoise project for coherent noise generation.

2.1.2 ProgenyDemo

ProgenyDemo acts as an example of how Progeny could be integrated into a game engine. It uses Source Studio's Horizon engine to render Progeny's outputs in an interactive way. This is the project that builds an executable application; all other modules generate library files. Additionally, ProgenyDemo provides the user with a menu to adjust various planet parameters (such as radius, color, and terrain features), and then explore it using an orbital or free-look camera.

2.1.3 Horizon

Horizon is Source Studio's proprietary game engine; it should have no dependence upon any of the modules created for the project. It currently uses DirectX to provide 3D rendering, but this is abstracted through its own graphics layer, and it is planned that it will eventually offer OpenGL support as well.

2.1.4 libnoise

libnoise is an open source library used directly by Progeny, but abstracted away from other modules such as ProgenyDemo. This allows for changes in the noise implementation without cascading those changes to Progeny's clients.

2.2 Progeny Technical Design

2.2.1 Portability

To facilitate the integration of Progeny with a client's graphics system, two key interface classes are provided: `IPgRenderer` and `IPgCamera`. These are implemented in ProgenyDemo as illustrated below in Figure 5.

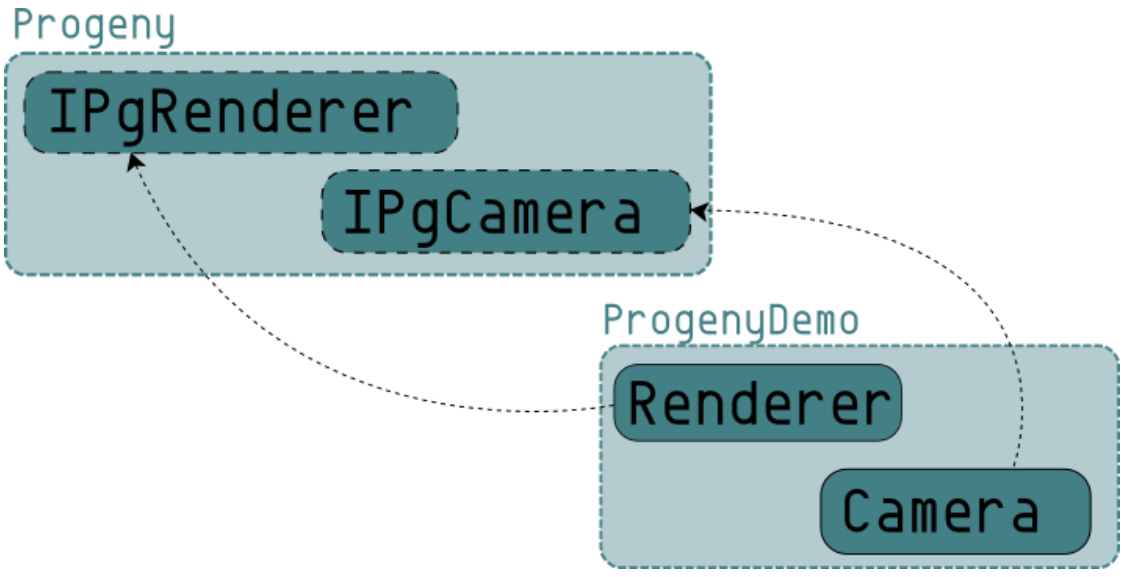

Figure 5: Progeny Interfaces, and their implementations in ProgenyDemo

`IPgRenderer` contracts a render function with parameters for a pointer to an array of vertex indices, `PgVertex` objects, and the number of triangles to be drawn in the current frame. This method of rendering, known as immediate mode, is commonly used in graphics applications for geometry that changes frequently (such as our ROAM planetary surface.) As immediate mode rendering is common to many popular graphics systems, this abstraction layer should allow for easy integration while keeping Progeny independent of any one system.

`IPgCamera` is another small interface class that needs to be implemented by the client application. This allows the ROAM implementation to query the camera state and maintain an optimal mesh without having to maintain its own camera state. It contracts functions for getting the camera coordinates (in world Cartesian space) and the camera heading (as a normalized directional vector in the same space.)

2.2.2 Extensibility

Progeny has been designed with extensibility in mind by allowing the client application to supply custom procedural algorithms or static data to drive surface generation.

An implementation of the `IPgPlanetStrategy` interface class is given to the `PgROAMPlanet` through its constructor. This association allows the planet to position and color its vertices by querying positions on a unit sphere. Although Progeny supplies a robust and highly customizable terrain generation strategy capable of creating an incredible variety of terrestrial type planet surfaces, the use of this interface allows the user of the library to provide his or her own terrain algorithms (that may or may not be procedural or random.)

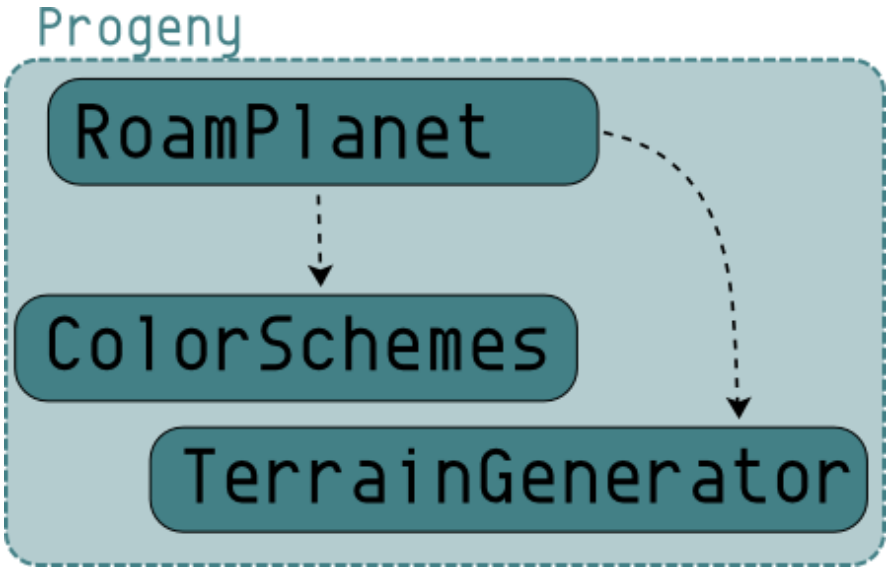

Figure 6: A planet is associated with a color scheme and a terrain generator.

The provided planet strategy, `PgPlanetStrategyRandom`, also takes `PgColorScheme` objects at instantiation. While a selection of color schemes based on the colors of the planets in our solar system are provided, the user is free to implement their own to be used by the planet in assigning colors to vertices as they are created. Figure 6 illustrates an abstracted design of a planet's dependency on a color scheme and a terrain generator.

3.0 Implementation Details

3.1 Progeny

3.1.1 Progeny Class Descriptions

The core class of the Progeny module is `PgROAMPlanet`. This class is instantiated by the client application and passed an object implementing `IPgPlanetStrategy`. While this interface can be implemented by some custom class, Progeny provides a robust implementation called `PgPlanetStrategyRandom`.

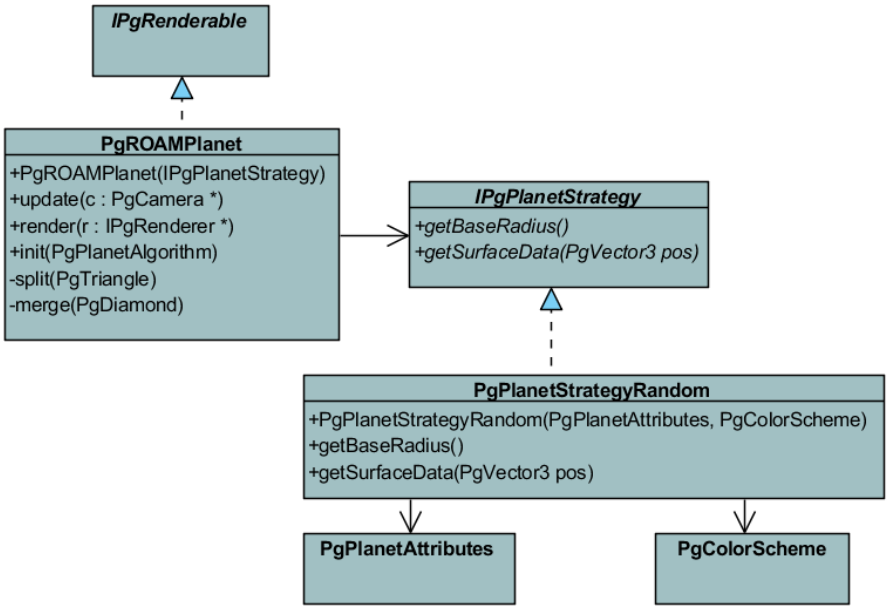

Figure 7: The terrain and colors of the planet are accessed through a planet strategy, given to the planet instance at construction.

`PgPlanetStrategyRandom` implements a terrain generator by combining libnoise modules (see details below under Terrain Generation). We set the parameters for terrain generation by passing in a `PgPlanetAttributes` structure which holds values for seed, radius, ocean height, mountain frequency, continent frequency, noise amplitude, and crater state. As our terrain generator modules are deterministic, this allows us to recreate a desired planet simply by storing these values and recreating the `PgPlanetAttributes` object in a later session. This enables artists and game developers a level of manual control over their planet construction while still maintaining a very small size with regards to game distribution.

Additionally, we provide a `PgColorScheme` that defines the colors of the terrain as specified by normalized elevations (with 0.0 being planetary minimum and 1.0 being planetary maximum.) We can also define a polar region in the `PgColorScheme` that allows us to blend in a color towards the polar zones. This is useful for creating icy polar caps in Earth-scheme planets, as well as a subtle striping effect in other color schemes.

3.1.2 Continuous Level of Detail - ROAM

Algorithm Architecture

Progeny makes use of an algorithm called Real-time Optimally Adapting Meshes, or ROAM, to provide a camera-optimized triangle mesh each frame. This algorithm was proposed by Mark Duchineau et al. to tackle the problem of rendering very large surfaces with high frequency data at an efficient, real-time rate.

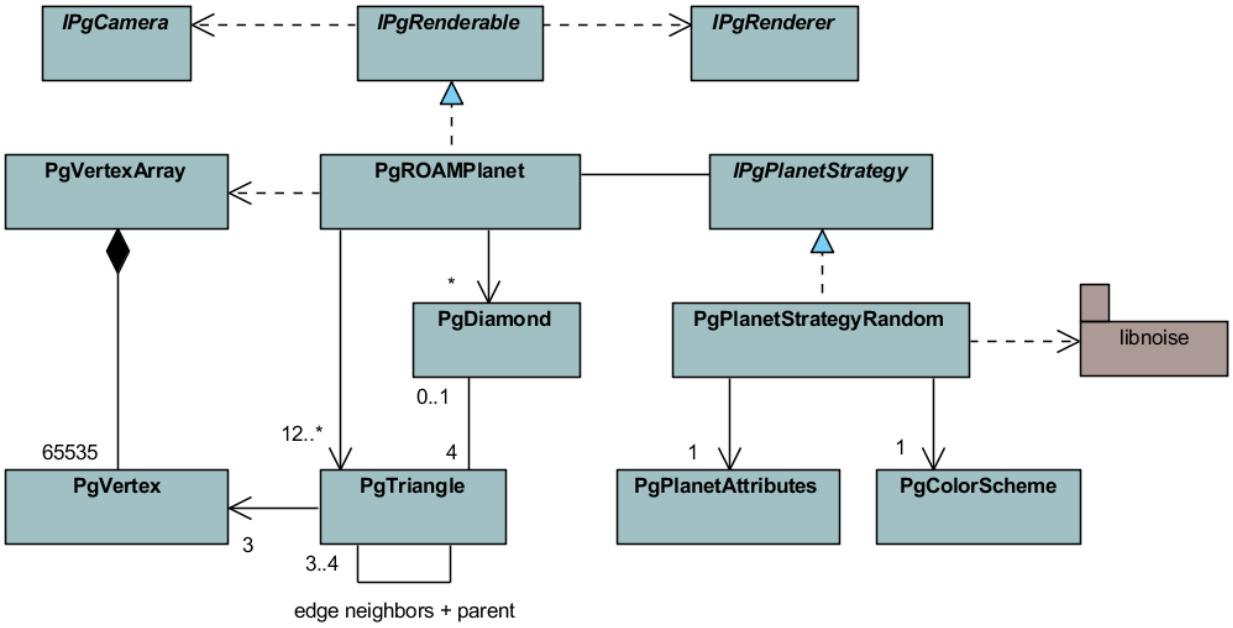

Figure 8: diagram of the class structure supporting Progeny's ROAM implementation.

Figure 8Figure 6 illustrates the design of the object hierarchy within the Progeny library. The central class, `PgROAMPlanet`, uses an implementation of the ROAM algorithm in combination with a given `IPgPlanetStrategy` to create a planetary surface.

Each planet is initialized as a cube, made of eight vertices and twelve triangles. Prior to rendering each frame, the client application (e.g. ProgenyDemo) calls update on the `PgROAMPlanet` through the `IPgRenderable` interface. This method takes an `IPgCamera` as an argument.

Triangle Splitting

Within the update method, the `PgROAMPlanet` will first iterate through all its associated `PgTriangle` objects (stored in a linked list), and get the priority value of each. This priority value is a function of camera position, camera heading, and error offset.

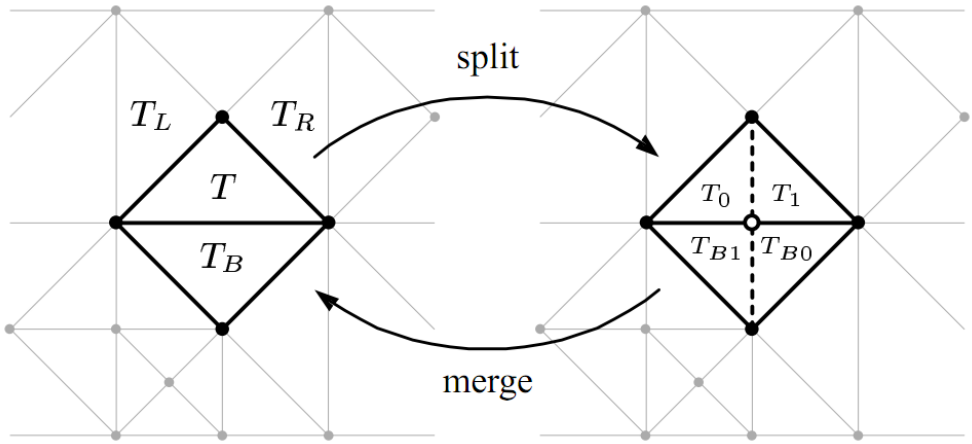

Figure 9: The surface of a triangle mesh as two neighboring triangles are split and merged.

The error offset is the distance between where the midpoint of the long edge of the triangle currently is, and where a vertex would be positioned if this edge were split in two and a new vertex were placed according to a query of the planet's `IPgPlanetStrategy`. As an optimization, the planet also calculates the approximate distance to the planet's horizon, and passes this as an argument to each triangle's `getPriority` function.

If the triangle determines that it is further from the camera than the horizon, its priority value is immediately returned as 0. Similarly, the dot product between a directional vector from the camera to the triangle's midpoint, and the camera's heading is calculated. This is used to determine whether the midpoint is actually in the view frustum. This allows the priority function to quickly simplify the geometry to the sides and behind the camera when it is zoomed in to a low altitude. Otherwise, we must calculate the triangle's actual positive priority value. This is done as a simple sum of the squares of the distance from the camera

to the midpoint, and of the error offset (a sum of squares is used simply because a square root operation costs quite a few more CPU cycles than a multiplication.)

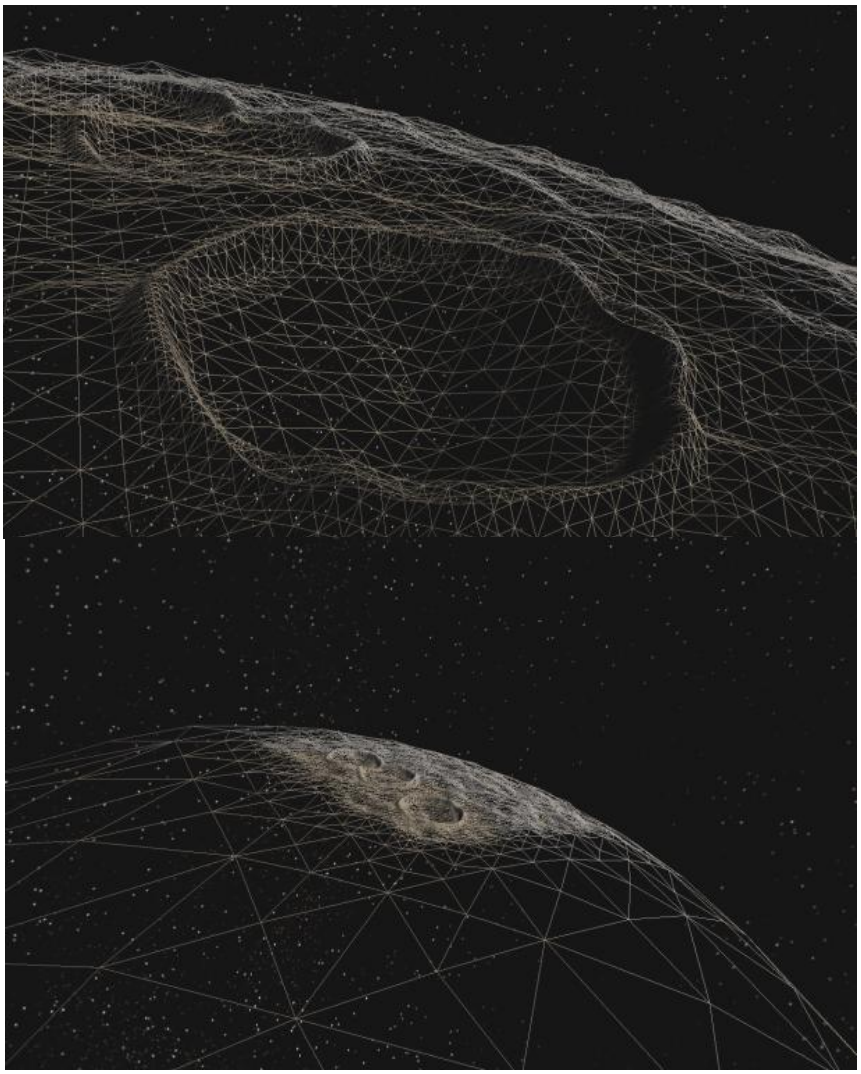

Figure 10: A ROAM-split surface and the same surface viewed by a virtual third person camera.

Each triangle that is determined to be of a given tolerance value for this frame (passed to the planet through the update call) is split. Additionally, we can only split a triangle if its neighbor (the triangle along its longest edge) is also split. This is to avoid surface irregularities and visible seams in the triangle mesh. The split operation itself creates a new vertex by pulling one from the pool of pre-allocated vertices, as managed by

`PgVertexArray`. This new vertex is pushed or pulled out from the planet's base radius by the error offset. The split method then manipulates the edges of the given triangle and its neighbor to turn them from two larger triangles into four small triangles. These four triangles are stored in a new `PgDiamond` object (with the original two triangles being specified as parent triangles, and the two new ones as child triangles), which itself is added to a linked list held by the planet.

Diamond Merging

The other half of the update step is merging. The planet iterates through the mentioned list of diamonds, and calculates the priority value for each. This is calculated identically to how it is calculated for triangles, except the error offset value represents how much the middle vertex of the diamond was pushed or pulled from its parent triangles when the diamond was created. If the priority value is *less than* the given tolerance, the diamond is merged. This operation releases the middle vertex, and recreates the original two triangles by deleting the two child triangles and adjusting the edges of two parents to the outer vertices.

By performing these two stages each frame, the planet quickly becomes a complex surface despite having been initialized as a simple cube, and the surface maintains its excellent fidelity as the camera zooms in close to the surface. Figure 10 shows the results of this ROAM implementation as we can observe the detail in a surface mesh from the perspective of the main camera, and from the perspective of a third person camera at some distance behind that.

Terrain Generation

Terrain generation is the responsibility of a set of classes which `PgPlanetStrategyRandom` owns a reference to. It uses the `PgPlanetAttributes` structure to set various parameters on the objects under its control as shown in Figure 11.

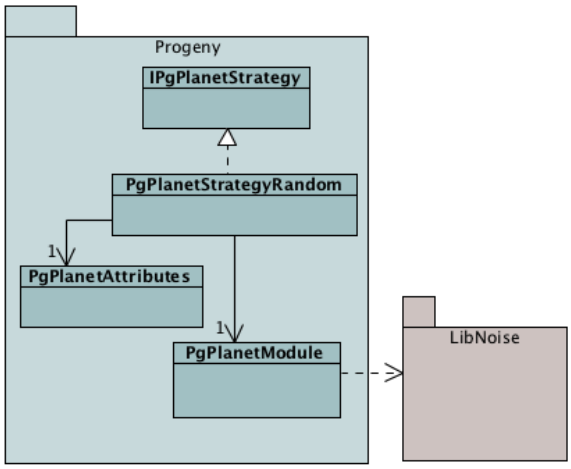

Figure 11: Noise / Planet Relationship

As mentioned earlier, terrain generation is accomplished through the use of coherent noise functions provided through the external library libnoise. Modules provided by libnoise are designed to be chained together in order to generate complex noise based structures. Modules generally implement a collection of common functions as shown below.

```
double GetValue(double X, double Y, double Z)
```
Returns the noise value at this point in the noise volume.

```
void SetSourceModule(int ModuleNumber, Module SourceModule)
```
Sets this module's source module, and the identification number to use for this module.

```
void SetControlModule(Module Module)
```
Some modules require another module to control some kind of behavior such as selecting between two source modules.

Progeny extends the module interface in order to generate its own high level modules as seen below in Figure 12. These modules are `PgTerrainModule`, `PgPlanetModule`, and `PgCraterModule`. Each of these modules uses a collection of libnoise modules in order to

generate a specific type of terrain. Further documentation about libnoise modules can be found on the official libnoise website [LIB2007].

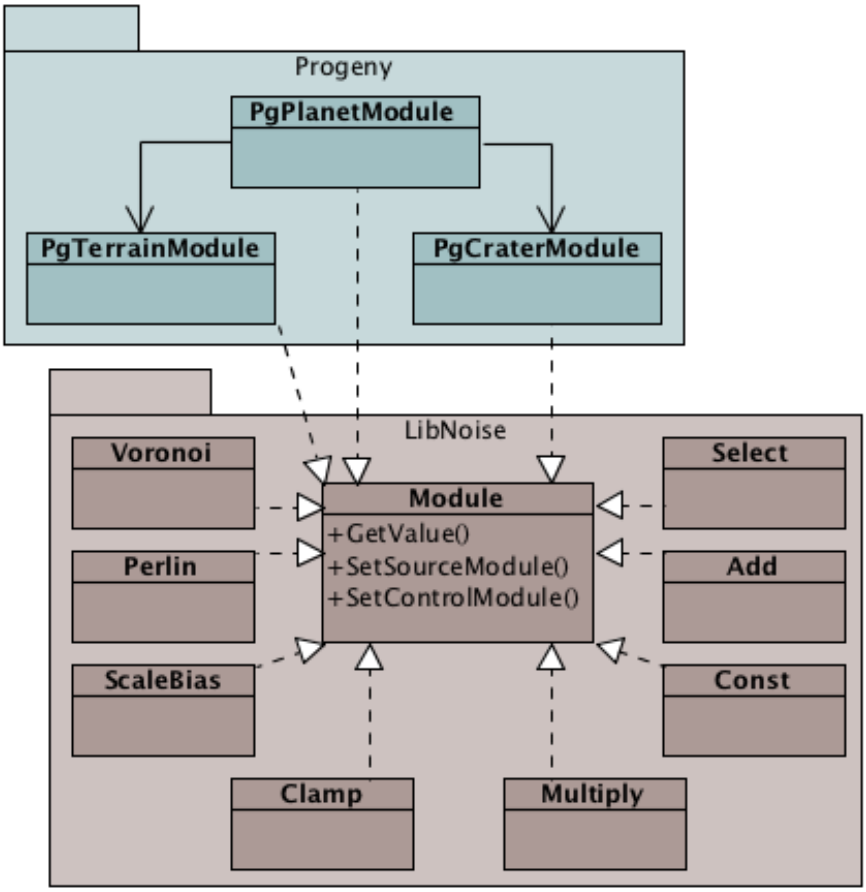

Figure 12: Class Diagram - How Progeny Modules implement the libnoise Module Interface

PgCraterModule

The crater module starts with the Voronoi primitive. It then clamps the Voronoi output off at a specific level in order to isolate a small number of dark spots which can be used to generate craters bowls. These cells are then clamped to a minimum value and the output of that is inverted to generate the craters edges. Finally the entire is scaled so that the flat area is at zero. This is important so that the crater output can later be added to existing terrain. The output of these stages can be seen below in Figure 13.

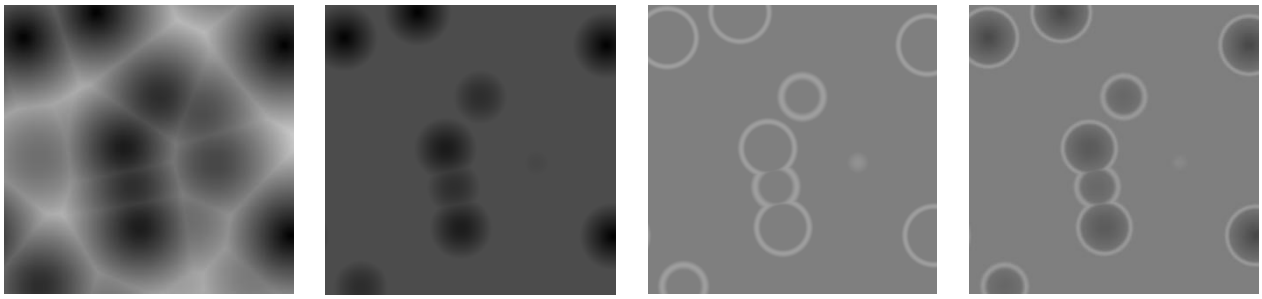

Figure 13:
(a) Voronoi Cells (b) Clamped Cells (c) Inverted Clamped Edges (d) Finished Craters

Initially functionality was added to allow our craters to be less circular and therefore more realistic (see Figure 14 below). However since the turbulence operation inherently uses three Perlin modules [LIB2007] it was found to be far too resource intensive for our application.

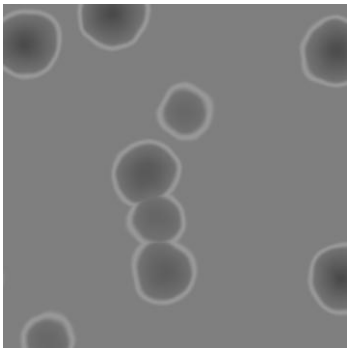

Figure 14: Turbulent Craters

PgTerrainModule

The terrain module makes use of Perlin and Ridged Multi-fractals (see Figure 15) in order to generate the base terrain. The basic continent mask is constructed of low frequency Perlin noise which contains multiple octaves. On top of this another layer of Perlin noise is added. This is the detail noise; it is a much higher frequency, and much lower amplitude. This allows for high level continental detail coupled with very fine details at close zoom levels. In order to add mountainous terrain we add a layer of ridged multi-fractals using a selector module. The height of the mountain terrain is offset by the water in order to allow us to have mountains show up at a reasonable height.

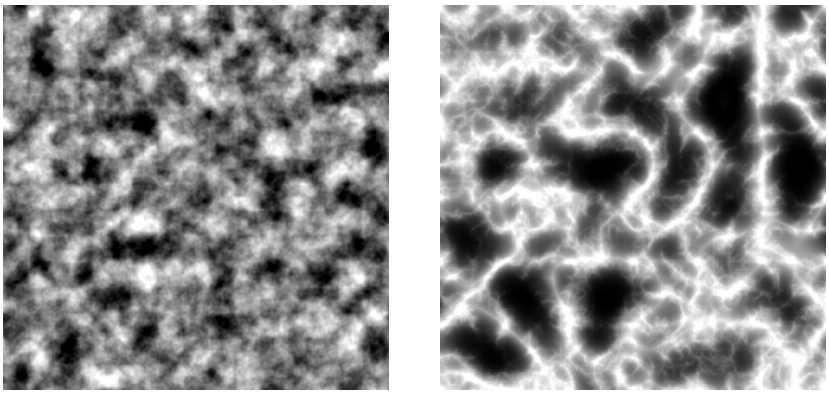

Figure 15:
(a) Perlin Noise **(b) Ridged Multi-fractals**

PgPlanetModule

The planet module adds the terrain and crater modules and applies an ocean height which flattens areas below a specified point. This gives the impression of a flat ocean surface. Coloring the ocean an appropriate color in a `PgColorScheme` module helps enhance this appearance. On planets without blue oceans the effect still generates interesting effects which seem to emulate the appearance of distinct lowland areas.

3.2 Progeny Demo

Progeny Demo is a sample project used to illustrate the project as well as give an example to future users how implement the required interfaces for Progeny.

3.2.1 User Interface

In addition to using Source Studio's Horizon engine for rendering the planet, it has been used to create the user interface. Utilities are available to handle controller input and draw the GUI.

3.2.2 Controls

There are two methods available for a user to interact with Progeny. A keyboard and mouse can be used to control the entire system. A list of the controls is available in Figure

16 below. Controls are based on control schemes that would be similar to users who play a lot of games. WASD are used to move around and the mouse changes the camera direction.

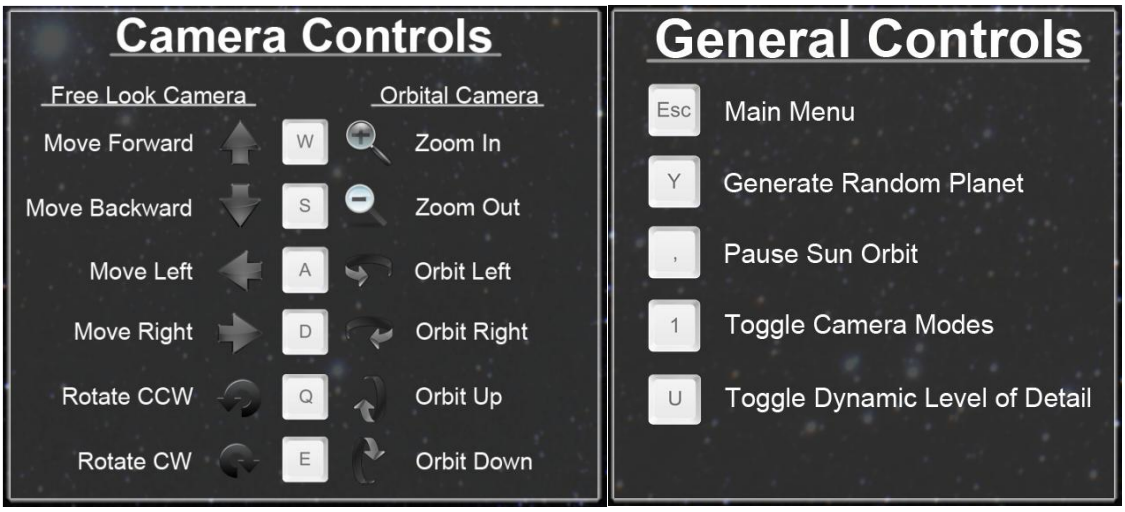

Figure 16: Progeny Controls

The users are also able to use an Xbox 360 controller to operate the camera and generate new random planets. The user is also able to toggle between wireframe, and reset the camera using the Xbox controller (see Figure 17).

Figure 17: Progeny Xbox Controls

3.2.3 Notifications and overlays

During the development of the project, it was decided that a heads-up display (HUD) should be created to display information to the user, as well as to display debug information. The HUD that was created consists of two parts. The first part is the notification pop-up, pictured below in Figure 18.

Figure 18: Notification Pop-up

The second piece of the HUD was the information bar, as seen in Figure 19. This was used internally as a debug tool to display information while we were debugging. As far as the user is concerned, the display bar shows only the triangle count and the vertex count. This was used to illustrate how the ROAM algorithm was functioning in real-time.

Figure 19: Information Bar

3.2.4 Customizing planets

When a user wants to customize the planet that they are viewing and exploring, they can open the planet creation menu (seen in Figure 20). From this menu the user is able to customize a number of parameters of a planet. The user is able to choose the radius, the ocean level, the continent frequency, mountain frequency, whether the planet has craters and the color scheme of the planet. Once these parameters have been selected, the user can then select the seed for the noise functions.

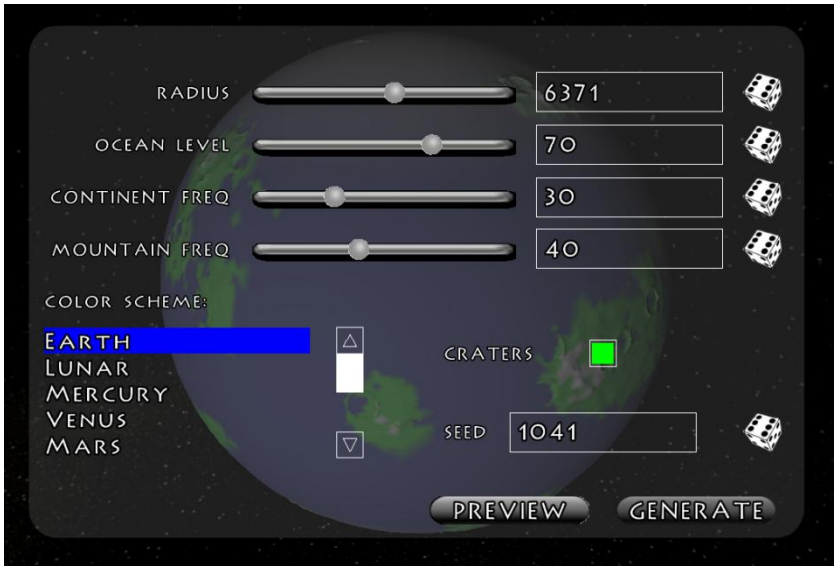

Figure 20: Customization Menu

3.2.5 Camera System

The ProgenyDemo project contains a camera system which allows the user to quickly and conveniently view the procedurally generated art assets. This functionality is vital given the nature of the algorithms being written, as it allows for a quick and highly effective "visual smoke-test" of the algorithms. Initially, an orbital camera system was constructed which allowed viewing of the generated planets in a spherical coordinate system. Eventually, however, the ability to fly along planet surfaces became a requirement. Thus, a 6-degree of freedom free-look camera system was also created. A simplified class diagram illustrating the design of the camera system is presented below.

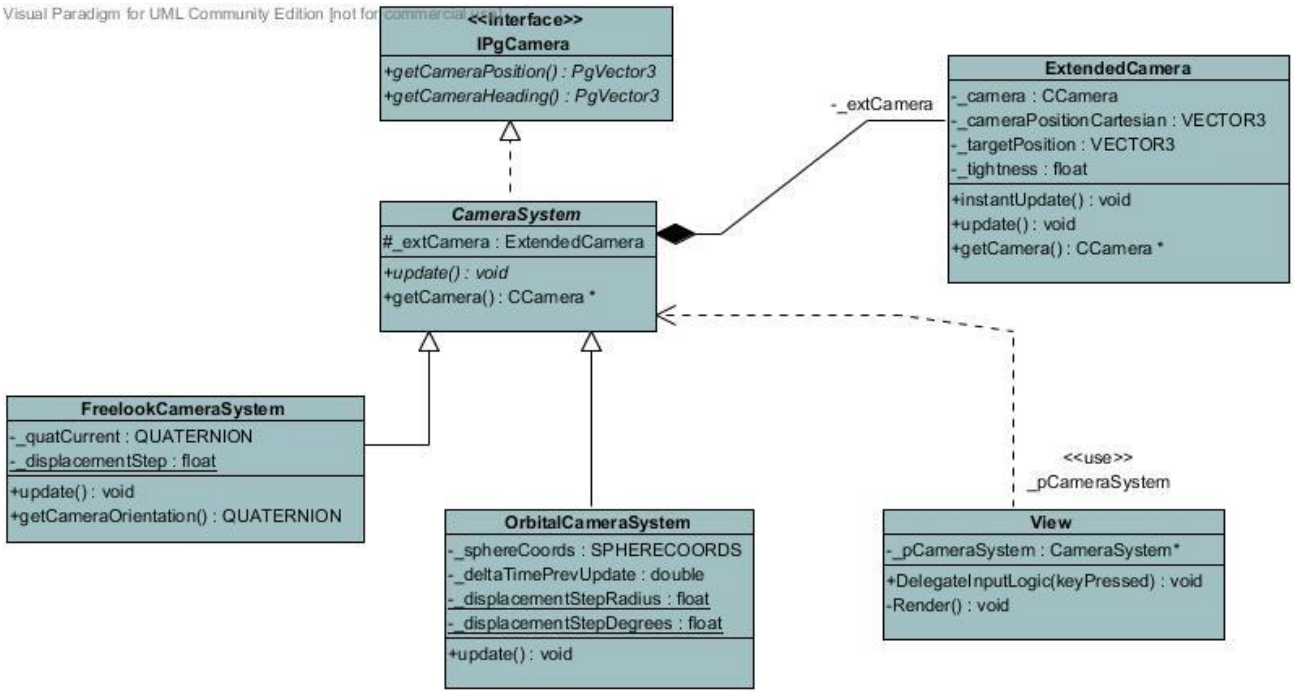

Figure 21: Simplified Camera System Class Diagram

As can be seen in Figure 21 the resulting system is a textbook example of the state pattern in action. The view is able to change its behavior at run time by switching state via the creation of a `CameraSystem` child class. This is advantageous as it allows for the seamless transition of viewing context on the fly, a desired feature of the camera system.

The `ExtendedCamera` class present in the system wraps the Horizon provided Camera and extends is functionality by allowing for advanced movement effects. At the moment an inertial movement effect is implemented using a simple linear equation which smooth's displacement of the camera. It does this by having the movement occurs over several frames instead of instantly (see Equation 1). Additionally, `ExtendedCamera` could support a wide variety of movement effects; they need only be implemented as replacements for the update function in the class (if a wide variety of movement effects is desired the strategy pattern could also be employed).

$$\begin{matrix} x' \\ y' \\ z' \end{matrix} = \begin{matrix} x_o \\ y_o \\ z_o \end{matrix} - \begin{matrix} x_i \\ y_i * tightness * \Delta t \\ z_i \end{matrix}$$

Equation 1: Inertial Camera Movement

The `OrbitalCameraSystem` provides a convenient way to view generated geometries. It contains a representation of camera position in a spherical coordinate system and then converts that representation to a three-space Cartesian representation to allow parameterization of the calls to `ExtendedCamera`.

The `FreelookCameraSystem` provides full control of translation and reorientation in three-dimensional space. It accomplishes reorientation via the use of quaternions, a mathematical construct which are useful for handling orientations in three dimensions. `FreelookCamera` handles the reorientation and translation behaviors but delegates the view matrix construction and storage responsibilities to the `ExtendedCamera` class.

4.0 Final Project Schedule

Due to the inherent volatility of software development, it has been agreed that the waterfall model of software development is rarely effective. As a result, the team opted to use a modified version of the Scrum Agile Development process.

4.1 Scrum

The remainder of the project organization section assumes to reader has some familiarity with the Scrum process for Agile project management. A succinct video describing the process helps to illustrate the basic concepts [HAM2008], while a more in depth guide may be helpful in clarifying some finer points [SCPR2010].

4.1.1 Term Three - Ceres Release

Goals

The goal for this release was to complete the ROAM implementation, terrain generation system and demo application such that our project was ready to be demonstrated at IEEE senior project night.

The sprints completed during this release along with their corresponding start and end dates are shown in Figure 22.

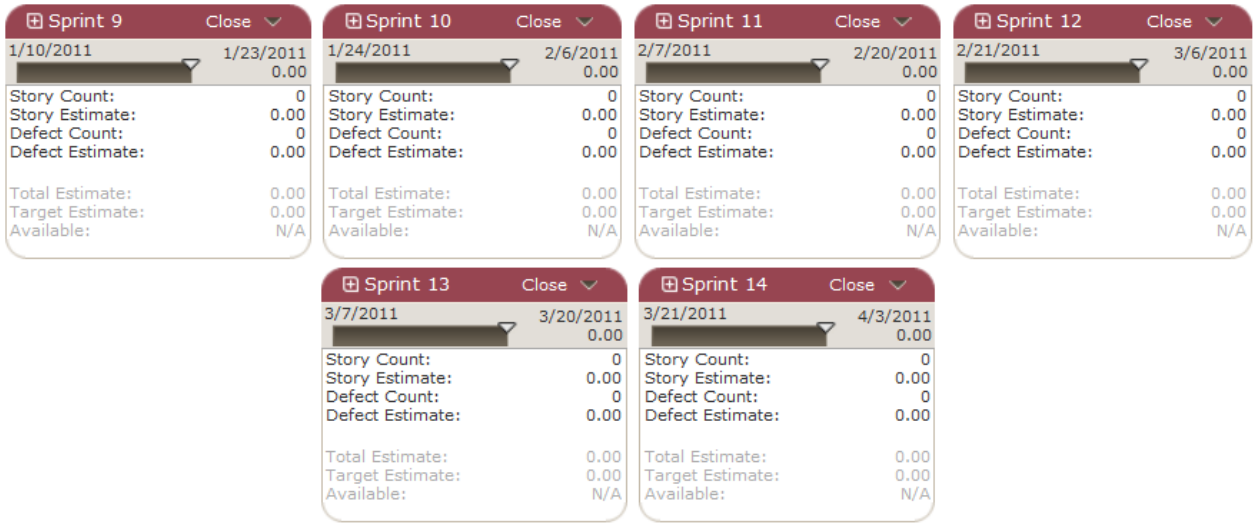

Figure 22: Sprint Breakdown

Release Retrospective

This release went as planned without any major stumbling points. We were able to reach the goals we had in mind for the senior demo night in addition to a couple of "nice to have" stretch goals. The team is quite pleased with the state of the project at the end of this release. The stories which were taken on during the six sprints of this release are presented below Table 1- Table 6.

Table 1: Sprint Nine Completed Stories

Title	Priority
Progeny can Instantiate and render `PgROAMPlanet` as a Cube [ROAM phase 0.5]	High
The user can switch between camera modes via keyboard input	High

Table 2: Sprint Ten Completed Stories

Title	Priority
The Split operation of the ROAM system works to divide cube into sphere given a max triangle count [ROAM phase 1]	High
The ROAM system has support for Splitting and Merging triangles to stay at the max triangle count [ROAM phase 2]	High
The midterm Progress Report must be completed	High

Table 3: Sprint Eleven Completed Stories

Title	Priority
The Free-look camera system must support the reorientation of the camera object via mouse input [via Quaternions]	High
The ROAM system is parameterized by the view frustum for the purposes of intelligent splitting of triangles [ROAM phase 3]	High

An abstraction layer must be created between libnoise and Progeny	High
Graphical overlays must be made which indicate Frames Per Second, # of Triangles, Max Triangles	Medium

Table 4: Sprint Twelve Completed Stories

Title	Priority
User has control of the master seed value	High
The user must be able to control the speed of camera movement	High
Vertex coloring must be applied such that the generated planet has an earth like gradient	High
Terrain generation module must be completed for terrestrial planets	High
We need to be able to properly calculate the normal of the vertices for lighting purposes	Medium

Table 5: Sprint Thirteen Completed Stories

Title	Priority
A GUI must be created to accept parameterization of the ROAM planet	High
Some optimizations must be implemented which improve performance and reduce popping for the ROAM system [ROAM phase 4]	Medium
Implement linear interpolation between orientations when switching between free-look and orbital camera systems	Medium

libnoise must be used to produce more complex terrain using multi-fractals	Low
Optimizations need to be completed for the terrain module	Low

Table 6: Sprint Fourteen Completed Stories

Title	Priority
A demo station must be constructed for displaying the project during IEEE night	High
The Final Project Report must be completed	High
The project presentation must be created and rehearsed	High
Tweaking of control scheme based on user feedback	High
Graphical overlays must be created which indicate the control scheme, and current camera movement speed	Medium
Gamepad Support!	Medium

4.2 Project Management Post Mortem

The management philosophy of Project Progeny has undergone some significant changes since the launch of the project. Our original process was much more constrained and while it was in theory an agile process, it introduced significant overhead for our team size. We have decided to cut activities that produced significant overhead and didn't seem to provide significant advantages for their cost. While our team believes in the main philosophies of Scrum we believe that our team size and working schedule warrants a lighter weight process.

Another important aspect of our design process has been our change of direction regarding our development goals. Although we have adjusted our design process, we have always

tried to maintain an agile mindset. In the world of software development the initially purposed solution often overlooks vitally important details, as such it is wise to allow information acquired during the construction process to feedback into itself.

This is especially true of a project like Progeny; where we have very little previous experience with some of the main techniques and Libraries in use. Therefore, it has often been difficult to make accurate long term projections. Focusing on concrete short term goals has allowed us to avoid the pitfalls of shoehorning an inappropriate solution when better alternatives have presented themselves.

4.2.1 How the Scrum approach has evolved

As the Scrum process was applied over the duration of the project, the retrospective process unveiled some inefficiency with the formal method as it is defined for an environment with multiple teams. As a result, some adjustments to the way Scrum was being applied to the project were made:

- The amount of formal granular planning performed at the beginning of each sprint has been reduced. With a small team of only four people, it was found that there was not much value added as there was already a good understanding of the tasks required to bring stories to completion.
- The amount of detailed time estimation being done was reduced. The team is not working on a fixed schedule week to week; therefore there is less of a need to perform detailed time estimations for story tasks.
- Originally sprints ended on Sundays. As meetings with the supervisor and client were held every Wednesday, it was decided that a more natural time to end sprints would be this day.

Overall the Scrum approach was a unanimous success and using it helped ensure all team members were on the same page heading into each sprint. Additionally, it allowed all members of the team to be aware of, and in agreement on, the direction of the project.

4.3 Success and Challenges

When implementing Progeny, the team encountered a number of challenges to be overcome.

As it was always an important goal to maintain Progeny as a system completely independent of platform and rendering engine, the team had to learn a great deal about graphics interfaces and data structures to implement the output of the ROAM algorithm in a reusable and portable way. The Horizon engine was initially restricted in that it did not offer a rendering method for immediate mode rendering, but consultation with Source Studio revealed that this was an ideal choice for ROAM rendering, and the method was complete in a newer branch of the engine code. When it was brought over to the team's working branch of Horizon, the implementation of `HorizonPgRenderer` was quickly completed.

As ROAM is a complex algorithm, however, there were several instances where there arose bugs that took significant time to find and fix. For several weeks there was a blocking issue where zooming close to the planet would cause a stability problem where triangles would begin to consume other triangles, destroying the surface of the planet until the camera zoomed out enough and the problem triangles would merge and cause the planet to re-stabilize. As the number of triangles was going up in our status window, in addition to the graphical glitches, we wrongfully assumed that it was being caused by a singular problem in our ROAM implementation that was causing vertices to be misplaced. After a great deal of debugging sessions, it was discovered that the issue was caused by three underlying problems:

1. As there is a pool of 2^16 vertices available for use by the `PgROAMPlanet`, we indexed the vertices with the type "unsigned short". This gives us enough space to access every vertex. However, when we ask the planet to render, each triangle is given an array of unsigned shorts to append its three vertex indices into, including an unsigned short as the current index into this array. This caused a visual stability

problem when we hit a number of triangles 2^16/3 ~ 21845 since each triangle is writing three unsigned shorts into the array, the unsigned short index would overflow at this point and the first triangles in the array would have their edge data overwritten. This caused the random visual incoherence of the planet structure.

2. The initial eight vertices that form the cube basis of the planet were being positioned at the planet radius instead of at the position determined by the planet's associated strategy. This could sometimes result in severe surface discontinuities at certain levels, triggering an infinite split operation.
3. An optimization that was introduced to the ROAM split method where the error offset value was taken from one of the parent triangles and stored in the newly created diamond structure without recalculating it. However, this pull was being done after the parent triangles had their edges adjusted and a new error offset value was calculated based on the new longest edge. This resulted in error values that would not converge towards 0 as more splits were doing, also triggering infinite split operations.

Having these three issues in parallel was caused by the difficulty in an iterative implementation of ROAM (i.e. without doing most of it in one large attempt, it is hard to test piece-wise functionality), and assuming the issue was caused by a singular problem lead to great difficulty in debugging.

While procedural generation methods allow for simplistic interfaces to the generation of complex graphical designs, the implementation of such interfaces poses a number of challenges. One of the biggest challenges is setting up the procedural algorithms so that their parameters are easily adjustable, and so that they perform actions which are logical to unskilled users. Our team experimented with a couple of different planetary models before settling on our final choice. We believe our final choice provides an intuitive interface to the generation of fairly believable planetary geographies.

During this development process we found that modules such as Perlin noise, turbulence and Voronoi noise could result in severe performance penalties. We initially focused on a more complicated planetary model in order to make it more adjustable and varied. However, we quickly realized that similar results could be achieved with much fewer modules, at much greater frame rates. These simplifications only resulted in very minor decreases in visual fidelity. We felt that performance was extremely important in our project because ultimately it is designed to be integrated into real-time games running on consumer level hardware. While our team faced many different challenges we are ultimately pleased with the solutions we discovered.

5.0 Project Budget and Resources

Most of the tools and research materials used are free or open source. In order to aid in research, four copies of the book "Texturing and Modeling: a Procedural Approach" have been purchased for forty dollars each. Otherwise, entirely free research sources and tools are being used. There are a large variety of tutorials available regarding the subject matter, and various ACM and IEEE papers are freely available through Memorial University's library services. The remaining budget details can be seen in Table 7.

Table 7: Budget details

Texturing and Modeling: a Procedural Approach (research/book)	$40 x 4
Visual Studio 10 Pro (IDE)	$0 (Free for students)
Version One (scrum project management tool)	$0 (Free Team Edition)
Direct X SDK	$0 (Free)
Libnoise (noise generation library)	$0 (Open Source)
Tortoise SVN (source control)	$0 (Open Source)
Horizon (game engine)	$0 (License Agreement with Source Studio)
Skybox Textures	$20
Demonstration Backboard	$32
Backboard Print	$15
TOTAL	$227

6.0 Moving Forward

There are many directions in which this project could be expanded in the future. Below we elaborate on a few that we think would provide reasonable but challenging capstone projects.

Our current implementation of Progeny makes very little use of the GPU. This provides us with a great deal of portability. However, many of the algorithms used would scale well to parallel counterparts which could be run on the GPU. Therefore, writing a portion of the project in shaders seems like a logical next step. It's conceivable that our noise generation and terrain paging algorithm could be ported over to the GPU and see considerable performance gains at the cost of a modest loss in portability.

Another aspect which our team was unable to explore due to time constraints was the use of a texture paging algorithm. How this would be done on such a dynamic planetary surface it an interesting question that would require considerably more research and development work. There may be good methods of doing this work within shaders as well, though our team has very little experience in that area. Certainly, we have seen examples of procedural textures generated within shaders, running at real time speeds. So we know that such techniques are at least feasible. Perhaps this is an area that Source Studio would be interested in exploring in future projects.

Finally, we believe the power of procedural methods could be made even more apparent our demo project had the capability of generating entire solar systems or even explorable galaxies of planets. Such a project would have to devise procedural methods for the statistical distribution of planets and solar systems. Other planet types could be added to our library such as gas giants, and stars could also be procedurally generated. These are just a few of the potential future directions our team finds intriguing.

7.0 Conclusion

Implementing the design that has been shown in this report allowed us to build a reusable C++ library for procedural generation of art assets for games. Focusing our project on the generation of large scale planets has allowed us to demonstrate the strengths of procedural methods. Our library and demo show how easy it would be to generate a massive universe of planets impossible through traditional game development methods. Furthermore, it

demonstrates how complex art assets can be created through very simplistic interfaces, enabling unskilled users to create highly realistic and detailed art assets within seconds. Such technology could have powerful implications in the fields of user generated content and game editing tools.

Glossary

Coherent Noise - A class of noise function whose output is proportional to any change in its input. Coherent noise does not contain discontinuities.

GUI - Graphical User Interface

HUD - Heads Up Display

Multi-Fractal - A function that uses primitive fractal functions to build a more complex a varied fractal pattern. Perlin noise can be used in functions with itself to generate complex multi-fractal patterns.

Perlin Noise - A form of coherent noise that is often used as a primitive in the generation of procedural textures. Developed by Ken Perlin and introduced in his 1985 paper “An Image Synthesizer” and later expanded upon in his paper “Improving Noise”. Pure Perlin noise produces an effect similar to blurred random noise.

ROAM (Real-time Optimally Adapting Meshes) - A mesh adapting algorithm introduced in this paper “ROAMing Terrain - Real Time Optimally Adapting Meshes”. Designed to display dense mesh data where it is needed and simplify mesh structure where it is less important. Useful when applied to dense large scale terrain meshes which need to be rendered in real-time.

References

[SPUF2003] Francis Spufford. *The Guardian. Masters of their universe.* (2003, October 18). Retrieved from: http://www.guardian.co.uk/books/2003/oct/18/features.weekend

[MANN2010] CCP Mannapi. *Awesome Looking Planets*. (2010, January 19). Retrieved from http://www.eveonline.com/devblog.asp?a=blog&bid=724

[DUCH1997] M. Duchaineau, M. Wolinsky, D.E. Siegeti, M.C. Miller, C. Aldrich, M.B. Mineev-Weinstein. *"ROAMing Terain: Real-Time Optimally Adapting Meshes"*. Retrieved from https://graphics.llnl.gov/ROAM/roam.pdf (n.d)

[DAND2008] Christoph Matthias Dandorff. *An interactive rendering system for real-scale planets combining static with procedural data*. (2008, November 7). Retrieved from VTerrain.org: http://www.vterrain.org/LOD/spherical.html

[EBRT2002] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley Texturing and Modeling: A Procedural Approach, 3rd Edition. (2003)

[LIB2007] Jason Bevins. "libnoise: a portable, open-source, coherent noise-generating library for C++". 2007). Retrieved from sourceforge: http://libnoise.sourceforge.net.

[DAVX2009] Davlex Design. *ROAM Planet rendering.* (2009, April 3). Retrieved from Ogre Forums: http://www.ogre3d.org/forums/viewtopic.php?f=11&t=49849

[BTR2000] Bryan Turner. *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*. (2000, April 3). Retrieved from Gamasutra: http://www.gamasutra.com/view/feature/3188/realtime_dynamic_level_of_detail_.php

[ONEIL2001] Sean O'Neil. *A Real-Time Procedural Universe, Part Two: Rendering Planetary Bodies*. (2001, August 10). Retrieved from Gamasutra: http://www.gamasutra.com/view/feature/3042/a_realtime_procedural_universe_.php

[POM2000] Alex Pomeranz. *ROAM using Triangle Clusters (RUSTiC)*. Master's thesis, University of California Davis Computer Sciences Department. (June 2000)

[HAM2008] (December 09, 2008). SCRUM in Under 10 Minutes. Retrieved May 31, 2010, from You Tube: http://www.youtube.com/watch?v=Q5k7a9YEoUI

[SCPR2010] (2010). Retrieved May 31, 2010, from Scrum Training Institute: http://scrumtraininginstitute.com/home/stream_download/scrumprimer