5-11-2002

A Faster Technique for Rendering Meshes in Multiple Display Systems

Randall Eugene Hand

A FASTER TECHNIQUE FOR RENDERING MESHES IN MULTIPLE DISPLAY SYSTEMS

By

Randall Eugene Hand

A Thesis
Submitted to the Faculty of
Mississippi State University
In Partial Fulfillment of the Requirements
for the Degree of Master of Science
In Computer Engineering
In the Department of Computer Engineering

Mississippi State, Mississippi

May 2002

A FASTER TECHNIQUE FOR RENDERING MESHES IN

MULTIPLE DISPLAY SYSTEMS

By

Randall Eugene Hand

Approved:

_____	_____
Robert J. Moorhead Professor of Electrical and Computer Engineering (Major Professor)	Lori M. Bruce Assistant Professor of Electrical and Computer Engineering (Committee Member)
_____	_____
Joerg Meyer Assistant Professor of Computer Science (Committee Member)	James C. Harden Professor of Computer Engineering (Graduate Coordinator)
_____	
A. Wayne Bennet Dean of the College of Engineering	

:

Name: Randall Eugene Hand

Date of Degree: May 11, 2002

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. Robert J. Moorhead

Title of Study: A FASTER TECHNIQUE FOR RENDERING MESHES IN MULTIPLE DISPLAY SYSTEMS

Pages in Study: 132

Candidate for Degree of Master of Science

Level of detail algorithms have widely been implemented in architectural VR walkthroughs and video games, but have not had widespread use in VR terrain visualization systems. This thesis explains a set of optimizations to allow most current level of detail algorithms run in the types of multiple display systems used in VR. It improves both the visual quality of the system through use of graphics hardware acceleration, and improves the framerate and running time through modifications to the computations that drive the algorithms. Using ROAM as a testbed, results show improvements between 10% and 100% on varying machines.

DEDICATION

To my wife Laura, for standing by me during the last 6 months

TABLE OF CONTENTS

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Terrain visualization is a core part of most geospatial scientific visualization systems today. Whether rendered for context information or actual data, it is a valuable part of most visualization. Rendering terrains at high framerates, however, has proven to be a difficult problem with many solutions.

In multiple display systems, such as head mounted displays (HMD's) or the CAVE™ makes the problem more difficult through the added requirement of multiple renderings. Existing algorithms can rarely achieve the triangle counts or speeds required for such complex renderings. My thesis shows that both the tessellation and rendering stages of mesh algorithms can be improved for use in multiple display systems.

CHAPTER II

RELATED WORK

In 1993, Funkhouser and Sequin [1] developed a method for simplifying building walkthroughs using multiple levels of detail for each object. The different levels were swapped out based on distance to the viewer. They implemented a benefit and cost heuristic, where the goal was to maximize benefit (detail) while minimizing cost (rendering time). They achieved good results, but their implementation was somewhat restricted by the requirement of several independent objects. Terrains are one large object, and therefore this approach is not directly applicable to terrain visualization.

Silva, Mitchell, and Kaufman used these heuristics in a different approach [2]. They designed a system capable of generating triangular meshes from a uniform rectilinear heightfield at any detail level. Given an error threshold, they could simplify the mesh greatly into several irregular triangles. Unfortunately, the method was slow (anywhere from 3 – 30 seconds) so it had to be implemented out-of-core.

All the previous methods used simple tricks to swap out different levels of detail based on metrics like distance and screen space. The goal was to swap them out when the difference between the two models mapped to approximately one pixel of screen space, making the difference almost unnoticeable. Unfortunately, this was rarely the case and popping became a serious issue. Daniel Cohen-Or and Yishay Levanoni developed a

method of overcoming this using “geomorphing” [3]. Taking two different meshes, generated through delaunay triangulation, they could slowly insert and remove triangles to give the impression of a smooth transition. Unfortunately, this still required a lengthy out-of-core process to generate the triangulated meshes.

That same year, Hoppe [4] released his progressive meshes algorithm. Through a lengthy preprocessing stage, he could generate a list of vertex inserts or removals that could be rendered to display the mesh at any resolution. Through use of geomorphing, he could smoothly transition between them all. For terrains, this meant he could render exactly the number of triangles he wanted, and add or remove triangles to change the rendering time. Unfortunately, this preprocessing stage took several minutes, up to an hour on difficult meshes, and had problems “stitching” patches together. The problems could be overcome, but it was not a trivial task.

Also that year, Lindstrom and others released a real-time method for calculating levels of detail [5]. Using quadtrees and geomorphing, they could start with a coarse mesh and quickly add detail until a specified time or triangle count was hit. Then, on successive frames, they could slowly add and remove triangles until they reached a near optimal solution. The results were great, but the algorithm was complicated. The data was broken down into square blocks, and each block was represented as a quadtree. Each branch of the quadtree represented an edge of a triangle. Maintaining a crack-less surface was difficult, both within a block and between adjacent blocks.

The next year, Duchaineau and others released their edge-bisection algorithm called real-time optimimally adapting meshes algorithm, or the ROAM algorithm [6]. It

was a combination of the tree topology of Lindstrom's algorithm with the single triangle resolution of Hoppe's progressive meshes algorithm. By using a binary tree instead of a quadtree, Duchaineau was able to set error metrics based on the exact number of triangles desired. Other metrics were used to determine when to break a triangle, but since they were always broken in two, instead of into four, an increase of only a single triangle was generated. Combined with geomorphing, the results were faster and smoother than Lindstrom's. Lindstrom also proposed maintaining two FIFO queues for split and merge operations, to limit how much the image can vary from frame to frame. Unfortunately, this is requires a lot of code to implement and can be very costly in computing cycles, almost to the point of eliminating the benefit.

Hoppe later published an extension to his progressive meshes algorithm [15]. He found he could further improve his performance by using a view-dependant refinement method. By focusing the triangulations in the area near the viewing direction, he could focus more detail in a smaller area, using the same triangle count but improving the visual clarity. Several others took this work and put view dependant refinements in their own algorithms [14, 16] with great success.

ROAM's problem of the two priority queues still prevented widespread use and several people have attempted to implement better methods. John Blow found a way to eliminate the queues and still improve performance by using a hierarchical tree of spheres at each point, instead of the simple biased 1-dimensional system [11]. Although not explicitly stated, he implied this method offered significant increase in the frame rates and a reduction in the tesselation times [11].

Another group from Switzerland used quadtrees to implement some of ROAM's functionality [13]. They also implemented the ability to generate triangle strips at run-time. This makes rendering easier on hardware by reducing the number of vertices to send to the video hardware. They were able to implement some of Hoppe's progressive meshes ideas and some of ROAM's patch-stitching ideas.

ROAM and Progressive Meshes are widely used by many industries today. Scientific visualization systems use them for quick rendering of context information, such as terrain or bathymetry. Gaming companies use them for quick rendering of terrain information for their games. Improvements are constantly being made. Scientific systems tend to use Hoppe's progressive meshes algorithms, because they offer more accurate visuals at the cost of a longer pre-processing time.

ROAM is used widely in the gaming industry today in games such as TreadMarks [7], where the terrain is an important factor in the strategy of the game and the simplicity of the algorithm shortens development time. ROAM's unique ability to dynamically change the amount of detail makes it ideal for generating the terrains when a widely varying number of other objects may also be rendered in that frame (such as explosions, other users, or data points). Also, the preprocessing calculations (the variance tree generation) can be done very quickly and localized to a small area of the terrain, making it ideal for terrains that change during runtime.

CHAPTER III

PROBLEM STATEMENT

In recent years, virtual reality systems and virtual environments have become more popular. Systems such as head mounted displays, Immersadesks™, and CAVEs ™ are becoming more widespread. These systems have special requirements for software developers, the most noted being the multiple displays. Either the programmer must render two images on a single wall (for stereo), or render matching images on multiple walls to create immersion. It's been proven that virtual reality tasks require a high frame rate, about 20 frames per second (fps), to maintain interactivity, and they are sensitive to frame rate variations. Any frame rate variation of more than 10% begins to interfere with the users ability to interact [8].

Martin Reddy observed that level-of-detail (LOD) algorithms didn't seem to be propagating into the virtual reality industry as fast as in other industries. He tested several commercial packages and found that while most packages supported programmer-implemented distance-based LOD, only three packages supported actually generating these meshes, and none of them at run time [9]. Perhaps this comes from most method's reliance on a screen-space error metric at the cost of processing time.

Although Reddy's research is several years old, the information is still mostly true. Systems usually only implement model-based LOD, and these detail-reduced

models must be generated by the programmer beforehand. Systems like SGI's Performer [12] don't natively support any run-time generated LOD's, but user components do exist to implement some algorithms, or the programmer can develop their own.

In a four-walled CAVE™, eight images must be rendered (one for each wall, one for each eye). This means that eight different images must be rendered to generate a single frame. To maintain 20 fps, the system must be capable of generating and displaying the scene 160 times a second, or in about 6 milliseconds. Today's state-of-the-art algorithms run optimally around 30 fps and occasionally sacrifice image fidelity and add extensive artifacts at 60 fps [4,5,6].

This means that modifications must be made to existing mesh generation algorithms. The chosen algorithm must be modified to support the following:

1) A more drastic reduction in triangle count with no noticeable increase in execution time.
2) A steady frame rate, varying no more than 10% frame to frame.
3) A method of managing several sets of context-specific data, one for each display.

My thesis is that existing real-time mesh generation algorithms can be modified to make them feasible on a multiple display system. ROAM's exact control over the triangle count and top-down design can be used to help maintain a steady frame rate by controlling the number of triangles rendered per frame which will help the rendering process. Careful management and allocation of patches allows the data to be paged in and out as needed to help the mesh generation process.

CHAPTER IV

DYNAMIC DATA LOADING

As machines become faster and faster, scientists are finding they can enlarge and improve their models. Models like NCOM [18] are updated each year to include more layers of data and higher resolution data, but with these larger datasets there are difficulties in rendering. ROAM divides a dataset into square patches, and these patches are the minimum resolution of the data. Because of ROAM's recursive nature, patch sizes are usually powers of two. Common patch sizes are 64x64 or 32x32, but not every dataset is evenly divisible by these numbers. This creates the problem of what to do with these leftover rows and columns after patches are created. The most common solution is to add extra rows and columns of a special "empty" data value that will not be rendered or will be rendered fully transparent. This requires the machine to have enough memory to contain the entire dataset at once. Some larger datasets won't fit in memory and other methods must be used.

One could simply load and render it all, but this is a poor and naïve method. With a dataset of 1024x640 points, like the NCOM ¼ degree global data [18], and a patch size of 64x64, you would be able to exactly load a 16x10 grid of patches, or 160 patches. This means you would have to render a minimum of 320 triangles for each display, and in a CAVE™ where one unit equals one foot, you could be rendering data almost 1000

feet away. With a larger dataset of 6400x4000 points, like a tile from GTOPO30 [17], and a 64x64 patch size, the data must be padded out to 6400x4032 and loaded into a 100x63 patch grid. This grid would require a minimum of up to 6300 triangles to render in each display at possibly over a mile away. View frustum culling could cut this number down somewhat, but not consistently. Also, if each number is a four-byte floating point number, the data itself would consume 102M of RAM, and other temporary data structures (such as textures and variance trees) still need to be generated. Generating a complete bank of full-resolution, one byte per data point textures for this dataset would consume almost 25M of video RAM, per display. This number becomes significantly higher if features like mip-mapping are used. Modern desktop cards like Nvidia's Geforce 2 and Geforce 3 usually support 32M or 64M of RAM, which could only hold one or two sets of textures. Most high-performance machines, like SGI's hold 128M or more of RAM, which would allow for significantly more textures to be loaded simulataneously.

In our four-walled CAVE™-like device, the target number of triangles is 5000. This number must be low because it renders the scene eight times (twice for stereo, and four walls), which brings the actual number of triangles up to a more respectable 40,000 triangles per frame, or 1,200,000 triangles per second at an optimal 30 fps. Using a standard patch size of 64, there are too many patches to get good data resolution where it is needed because of the required two triangles for the distant patches. The GTOPO30 dataset requires more than 5000 triangles for a minimum rendering on a single display, and could require over 50,000 triangles in our CAVE to complete a single frame at

minimum resolution. NCOM requires 10% of our allocated triangles to simply render the minimum resolution.

Attempting to use larger patch sizes is also an option, but not a good one. Larger patch sizes increase the time needed to compute variance trees and increase the computational load for each patch. Each increase of the patch size by a power of two quadruples the size of the variance tree and quadruples the amount data it must store. This leads to increased lag when it must recompute the variance tree or load in new data. Also since the values in the variance tree are scaled by the values in lower levels, the variance figures become less detailed with larger trees. Larger patch sizes make view-frustum culling less effective since there are fewer patches to eliminate, and shifts the triangulation to be almost exclusively in the near-area, leaving only two triangles to represent an entire 256x256 area. Also, increasing the patch size only solves the problem when dealing with large datasets. Small datasets work fine with smaller patch sizes, so this solution would require custom patch sizes for each dataset size. Increasing the patch size does little but increase the amount of work and frustration.

Another approach would be to only load patches in a square around the user. An 8x8 square of patches around the user generates data several feet out, till it disappears into darkness with almost no tessellation on the border triangles. This would require a minimum of only 128 triangles, which is only 2.5% of our allocated number. This number will probably be slightly lower in practice, due to view-frustum culling, and it doesn't change between datasets because of the fixed size of the patch grid. This approach proves much faster, although it has problems with the user moving. Whenever

the user crosses the edge of the patch, the application has to load more data in the distance, which can only be achieved by reloading all of the patches at once. This creates a short jerk in the smoothness of the environment that is easily perceivable and very annoying, but it greatly improves the memory consumption.

The solution to this problem is in the user interface. The user has no way to move about the data but in small linear steps. There is no way to "teleport" or "jump" around in the data. Therefore, it is sufficient to only load the edge data in the direction the user is moving. The rest of the data simply needs to be shifted in the opposite direction. This means that in a 7x7 set of patches, only 7 or 13 patches need to be loaded instead of the full 81, as in Figure 1. Implementing this in a fast way involves building another array of patches, in which each element points to a patch in the actual array of patches. Then, one doesn't need to actually move any data at all. Simply adjust all the pointers in the new array, and reload the necessary patches.

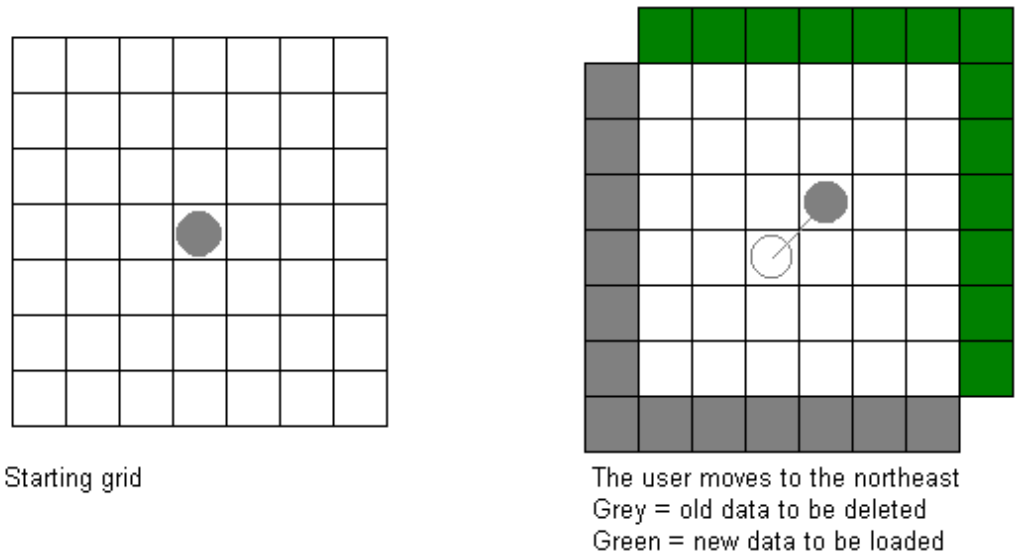

Figure 4.1: How the Patch Grid looks in use

This modification eliminates or reduces the perceivable pause between patch boundaries, and increases the speed of the entire system by eliminating data at a distance. Unfortunately, there is a noticeable "edge" to the data if the grid size is too small. This can easily be overcome by increasing the grid size, at the expense of visual detail, or by enabling fog.

CHAPTER V

SINGLE TREE RECURSION

ROAM works by splitting the data into a binary tree based on the perceived error between the parent triangle's hypotenuse and the two child triangles. This binary tree is maintained for each square patch of data and recursively rendered in the display thread, as seen in Figure 2.

ROAM starts by dividing the data into square patches. Each patch is a square piece of the dataset, one more than a power of two on each side (9x9, 17x17, 33x33). One row and one column overlap with two adjacent patches, so they are usually considered 16x16 or 32x32 patches since the extra row and column are redundant. This facilitates the "stitching" of adjacent patches. This patch is split along the diagonal into two triangles. At an absolute minimum, these triangles will be rendered. So a 1024x1024 dataset split into 64x64 patches will contain 256 patches and at least 512 triangles.

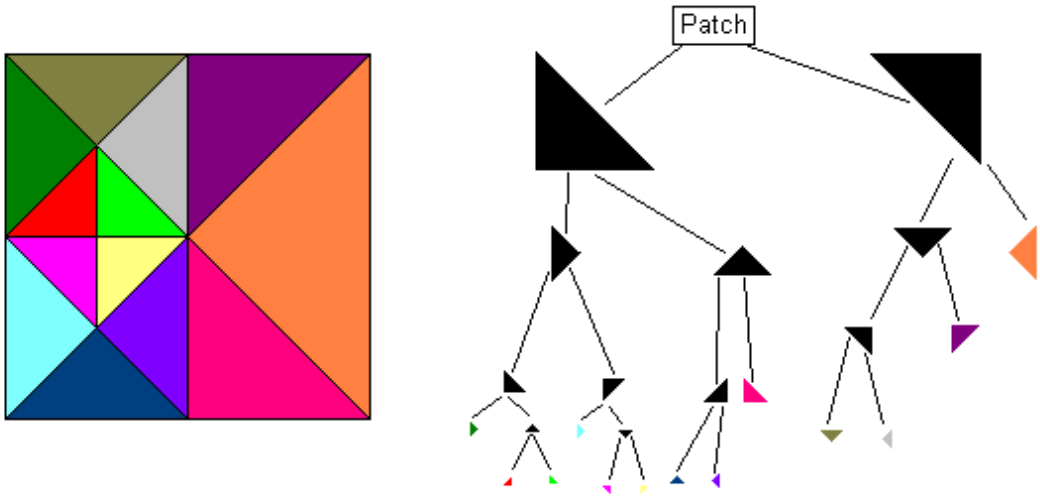

Figure 5.1: Example of a tesselation & the corresponding binary tree

Then ROAM generates a variance tree. The variance tree contains a calculation of the amount of error at each point if it is not rendered, as shown in Figure 3. It is stored as a binary tree, such that each entry corresponds to the hypotenuse of the parent triangle. Each error is also scaled by the amount of error from all the child triangles, meaning a quantitatively large feature that is spatially small will raise the variance for all the parent triangles, hopefully forcing it to split more often. After the variance tree has been built, it never needs to be updated. This is done once for each patch.

Finally, during the render phase, the two parent triangles are recursively split in two until the variance at the current level is below some threshol, as shown in Figure 4. If the desired number of triangles is reached, tessellation stops and the variance threshold is raised. This means fewer triangles should result in the next frame. If tessellation

completes, and there are triangles left, then the threshold is dropped to allow for more detail in future frames.

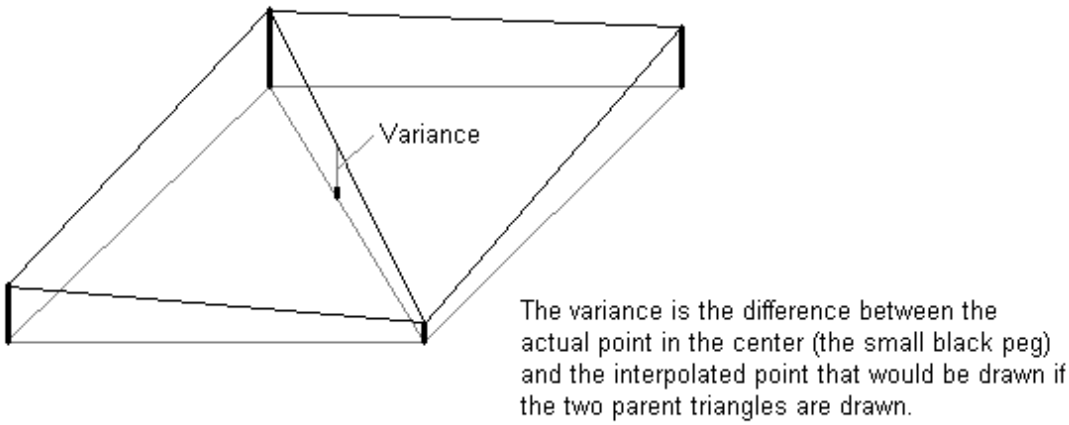

Figure 5.2: Example of Variance

During this tessellation, triangles are either rendered immediately when it's discovered that they will not be split, or, more commonly, they are stored in a symbolic tree. The tree doesn't store any vertex information, simply pointers to child triangle nodes. This is because the vertices can be calculated by dividing the triangle edge in two at each recursive call to the render routine until the binary tree stops at a leaf node [6]. The final result is a dynamically built mesh generated from a grid of data points, as shown in figure 5.

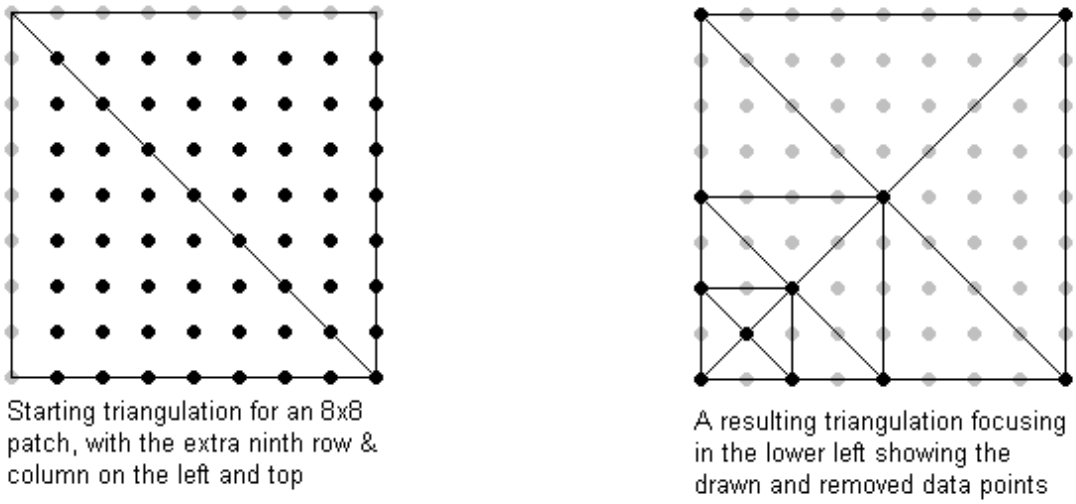

Figure 5.3: Example Tesselation

This works well for single displays, but multiple displays can cause significant problems. While the code for this recursive rendering routine looks deceptively simple, a lot of assembly code is generated automatically during compilation to handle pushing and popping of registers. All of this code can add up quickly, especially when it must be run multiple times per frame. A large dataset can actually cause the stack to overflow if it recurses too deep. In a single display environment, like a desktop, the recursion cannot be avoided. In a multiple display system, like a CAVE™ or HMD, there is a much better method. My thesis is loading the data into a single linear array before rendering starts would greatly improve performance. Then the array can be drawn to the screen as many times as necessary in a single non-recursive function.

In order to easily implement other speed-up techniques, each patch should maintain its own list of triangles for the frame. This way, view frustum culling can eliminate entire patches at a time, improving performance even more.

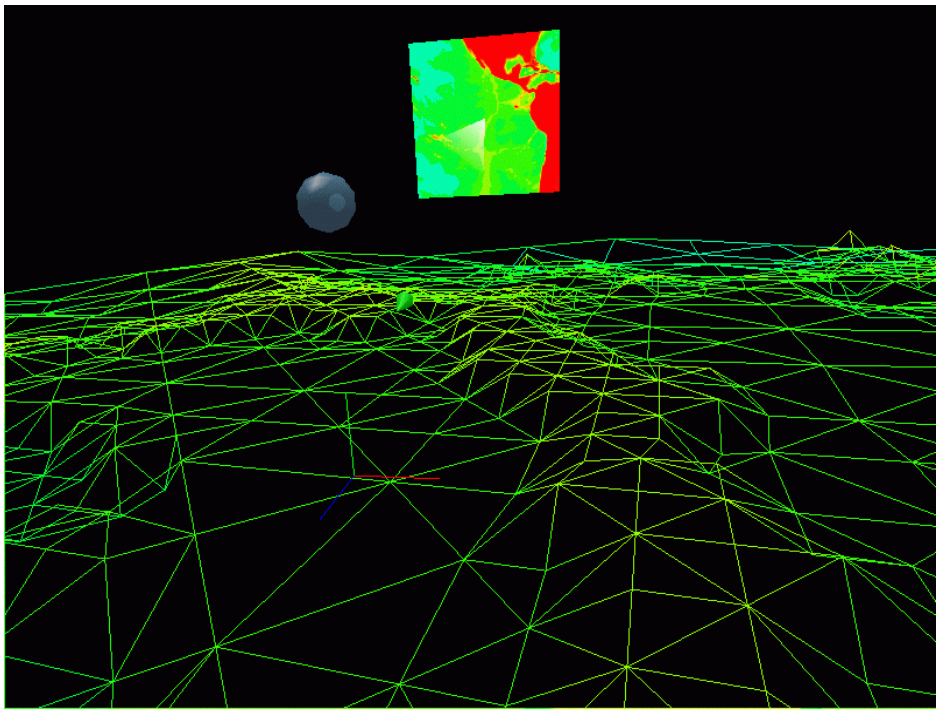

Figure 5.4: A tessellated scene

CHAPTER VI

FULL RESOLUTION COLOR MAPPING

Another way to increase the perceived detail is through the color mapping. Previous implementations of ROAM used a single randomized texture to add color, or used per-vertex coloring. Most scientific applications, however, want a color mapped to a data value on top of the surface. With previous ROAM implementations, this would cause a lot of flashing and blinking as colors appeared and disappeared as vertices were added or removed, and features would appear to fade away into the distance as the tolerable error increased far away from the user. In Figure 6, notice how the lighter green ridge seems to disappear toward the left of the user, and the light blue to the right is almost unnoticeable.

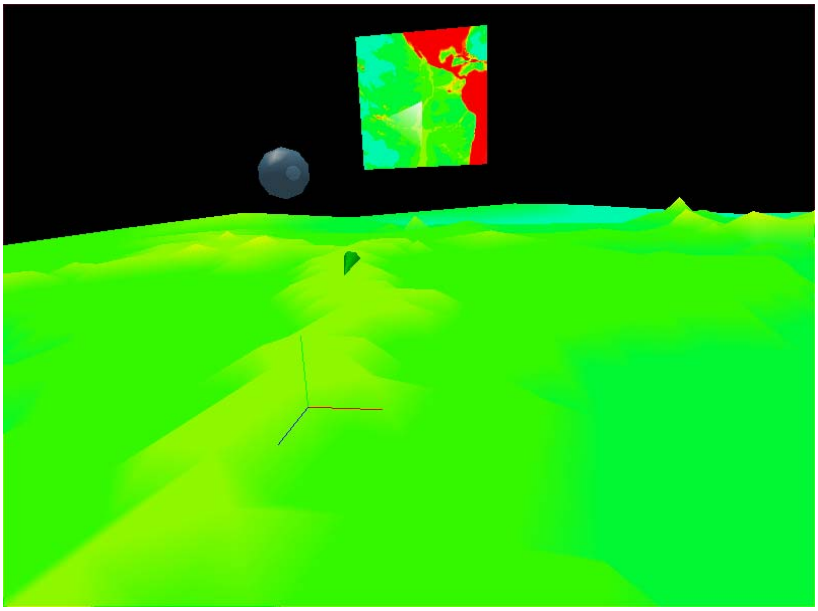

Figure 6.1: Example scene with Per-Vertex Colors

My solution is to add texture support to my implementation. When a new patch is loaded, a texture is generated for it. The texture has the same dimensions as the data, and uses a user-specified RGB colormap. The texture is flagged as new and is bound on each display process as it is used. It is not bound immediately to prevent the possibility that it never appears on that certain display and it is uselessly sent across the memory bus to the video system.

In the binary tree reader routine, discussed in chapter IV, texture coordinates are generated. Since each patch contains its own texture, the texture coordinates simply start at (1,1) and are divided by two at each division. This way, no matter what the level of geometric detail, the color will be accurate. This works well for areas with frequent slight changes, like a large area filled with small hills and valleys. These geometric

details will likely be removed, but the colors will show the rises and falls of the region. In figure 6, notice how the same scene which was rendered in figure 7 shows the light green ridge much more prominently, and how it continues all the way to the edge of the viewable dataset. The lighter blue area to the right is much larger and well defined, and the small red portion straight ahead of the user has appeared, which was not shown in figure 6.

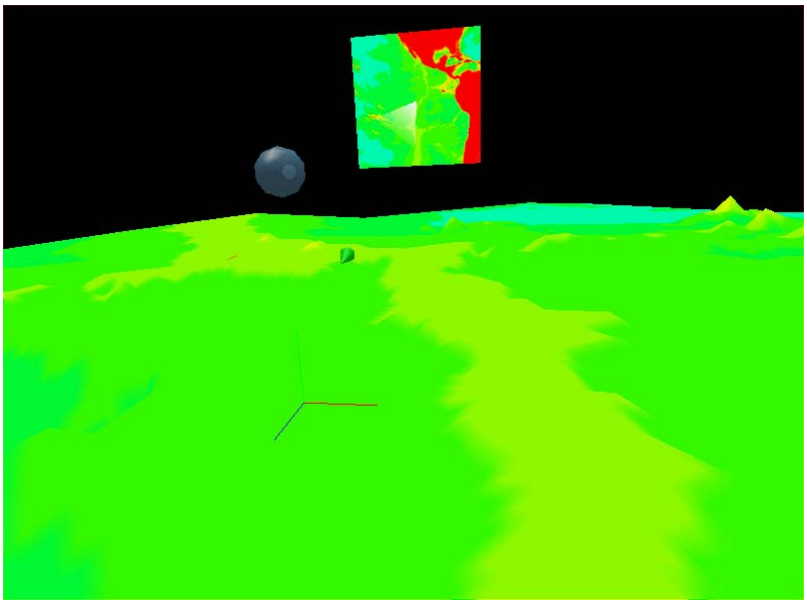

Figure 6.2: Example scene rendered with texture maps

CHAPTER VII

MULTIPLE DISPLAY BASICS & VRJUGGLER

One of the newest systems for multiple display virtual environments is VRJuggler [10]. Developed by Iowa State University's VRAC (Virtual Reality Applications Center), it supports all current display methods, runs on most operating systems, and supports several input methods. VRJuggler is C++ based and makes heavy use of templates and dynamic allocation and inheritance, but maintains a speed comparable to other systems such as CAVELibs [19].

VRJuggler works by abstracting inputs into several proxies that the user can configure based on their individual system. The programmer simply uses a single proxy that returns the correct analog or digital values, with no knowledge to what exactly that proxy is connected. This means that a single program can be run with a keyboard, a wand, a joystick, or a cyberglove, all with no change in the code.

The same is done for displays. VRJuggler configures the display and perspective matrices before calling the user's rendering function. A program starts with no information on the window size, how many windows there are, to which window imagery is being drawn, or for which eye imagery is being rendered (in a stereo system). All of these parameters are set before the render function is executed. This way, a program

does not need to make special considerations for each display or each eye, and a single display function can render to all walls.

Sometimes, however, there are operations that need to be executed once per frame, and not once per render. Processes such as loading data, calculating data, or handling user interaction can be done in the application's pre-frame calculation function.

All of these functions (pre-frame, display, etc) are specified through overriding functions specified in the vjGLApp class, provided by VRJuggler. There are also functions to spawn parallel threads to the display process (to run short calculations while the rendering continues), to run code after all the displays are done (good for calculating framerate), and to handle reconfiguration at run-time.

However, all of this flexibility does have some drawbacks. Things like display lists and textures must be bound and accessed by each display window individually, and there is no guarantee that a texture's ID for one window is the same for another window. This is handled through careful use of VRJuggler's vjglContextData template class. It creates and manages pointers to data on a per-display basis, so display list ID's and texture ID's must be put into structures that can be allocated and accessed through this template [10].

To use the vjglContextData structures requires a careful management of state flags. In the pre-frame function the textures could be built, because the same texture would exist for all displays. However, the pre-frame function cannot access context specific data because no context is active at that time, and there is no way of knowing how many OpenGL contexts are active. To accomplish this one has to create two types of flags for each such occurrence:

1) Global state flag – a single Boolean value to indicate if new data is ready. This is set to true in the preframe if new textures are available for binding. It is cleared in the postframe.

2) vjglContextData texture flag – a Boolean value that exists for each frame, indicating if this texture ID is valid and loaded. If the global state flag is false then this is forced to false also, indicating that this texture is outdated and needs to be rebound.

These two flags can be used to dynamically load and bind textures as necessary. They will be loaded a single time each frame and bound to each display context as they are rendered. The view frustum culling will also help out by not binding textures for patches that are not seen.

A few changes must be made to the dynamic data loader to optimize it also. By forcing it to use an odd number of patches (9x9,7x7, etc) surrounding the user, there is always a "center" patch that the user currently occupies. If the user ever leaves that patch, the patches must be moved to make the one the user is currently in the center. Pseudocode for these algorithms is listed below in Table 1 and Table 2.

Table 7.1

Data Moving Algorithm

```
01  New_Center_Patch = Patch containing User_Location;
02  Border_Left = Patch.Left – (PatchGrid_Size/2)*Patch.Width;
03  Border_Right = Border_Left + PatchGrid_Size * Patch.Width;
04  Border_Top = Patch.Top – (PatchGrid_Size/2)*Patch.Width;
05  Border_Bottom = Border_Top + PatchGrid_Size * Patch.Width;
06  Current_Patch = PatchGrid[0,0];
07  While (Current_Patch) {
08        If (Current_Patch.Left < Border_Left OR
09              Current_Patch.Top < Border_Top OR
10              Current_Patch.Left > Border_Right OR
11              Current_Patch.Top > Border_Bottom) {
12                    // This Patch is Invalid (Outside the grid)
13                    Current_Patch.Valid = FALSE;
14                    Current_Patch++;
15        } else {
16              // This Patch is Valid
17              Current_Patch.Valid = TRUE;
18              Grid_X, Grid_Y = Current_Patch.NewGridLocation;
19              Destination_Patch = PatchGrid[Grid_X, Grid_Y];
20              Swap (Destination_Patch, Current_Patch);
21              Current_Patch = PatchGrid[0,0];  // Restart
22        }
23  }
```

Table 7.2

Data Reloading Algorithm

```
01  Current_Patch = PatchGrid[0,0];
02  While (Current_Patch) {
03        If (Current_Patch.Valid == FALSE) {
04              Current_Patch.Valid = Current_Patch.Reload();
05        }
06        Current_Patch++;
07  }
08
```

While this sounds lengthy, especially the restart in the data moving algorithm, the pause is unnoticeable. Unfortunately, there is no easy way to prevent the restart with

every swap. The restart is required in the event that a data patch is swapped behind the current patch.

The final modification is made in the ROAM algorithm itself. The recursive render function was replaced with a function to simply store the triangle in an array, along with its color and texture coordinates. One is able to use an array because ROAM works on a target triangle count. There is always a maximum number of triangles to render, so one can simply use an array. Another function reads this array and draws it to the screen. The previous recursive render code is called right after the tessellation code, in the pre-frame calculations. The new render function is called inside the display function.

CHAPTER VIII

RESULTS

The final program was tested on several machines, both with and without the listed modifications. The test machines are listed in table 3.

Table 8.1

Test Platforms

Machine #	Type	Video	CPU
1	SGI Octane	V8	Two R12k, 400Mhz
2	SGI Onyx	Infinite Reality 3	Four R12k, 400Mhz
3	SGI Onyx2	Two Infinite Reality 2	Eight R10k, 250Mhz

The tests were run on the 1024x640 NCOM Global ¼-degree bathymetry data [18]. The data was stored locally on Machine #3 and accessed across the network through an NFS mount for machines 1 & 2. Each rendering consisted of approximately 700 frames, rendered through a pregenerated flight path to ensure each algorithm was tested identically.

Each platform was tested with three different algorithms:

1) Algorithm A used a basic ROAM implementation, with per-vertex coloring. It did not contain the dynamic data loading code.
2) Algorithm B used the same ROAM algorithm as Algorithm A, but also contained the dynamic data loading code.

3) Algorithm C contained the single tree recursion and texture mapping, as well as the dynamic data loading code.

All three algorithms were view independent, but did implement view frustrum culling on a per-patch basis. All accessed the same dataset in the same manner, except algorithm A loaded the entire dataset for each patch change. None of the processes were multi-threaded because of the varying number of processors on each machine.

As seen in table 4, in a single window the algorithm B achieves a slightly higher framerate on single display systems, but algorithm C fares better with multiple displays. This is as expected, since the single tree recursion offers no improvement in a single display system, as the tree must be recursed one time anyway. Qualitatively, however, Algorithm C rendered much better results because of the use of texturing. Colors were more consistent frame-to-frame, and small details and distant features were easier to locate.

The three algorithms were also tested in our four-walled CAVE™, run by Machine #3.

As seen in table 5, Algorithm C runs faster here also because of the savings from the single-tree recursion. In the CAVE™, eight renderings of the scene must be performed each frame. It should be noted that VRJuggler automatically multi-threads programs in the CAVE to place computation and rendering in different threads, but all three algorithms were threaded similarly.

Table 8.2

Benchmark Comparison (in fps)

Machine	Single Display			Multiple (5) Display		
	A	B	C	A	B	C
1	36.5fps	46.4fps	41.5fps	8.96fps	11.8fps	12.4fps
2	39.2fps	52.9fps	46.5fps	10.12fps	13.38fps	27.2fps
3	31.3fps	41.4fps	37.1fps	8.44fps	13.12fps	14.63fps

Table 8.3

CAVE™ Benchmarks

Algorithm A	Algorithm B	Algorithm C
10.24 fps	10.63 fps	13.88 fps

From the frame-rate plot for the CAVE™ (Figure 7), one can easily see that Algorithm C has a higher frame-rate than the other two algorithms throughout the entire rendering. Algorithm A and B are closely matched, but A consistently drops below as it slowly reloads the data.

The similarity is even more pronounced in a single display. On Machine #2, one can see (Figure 8) that algorithms A and B are similar in performance, except that A's performance consistently drops to 20fps while it loads data. When using multiple displays [Figure 9], the similarities are still present, but algorithm C shows far better performance.

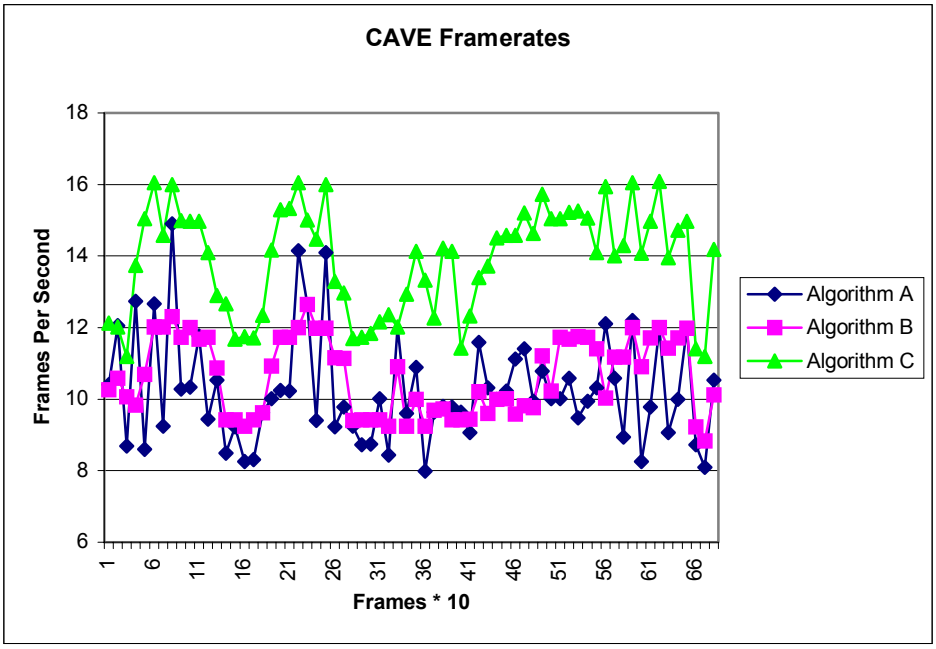

Figure 8.1: CAVE Framerates

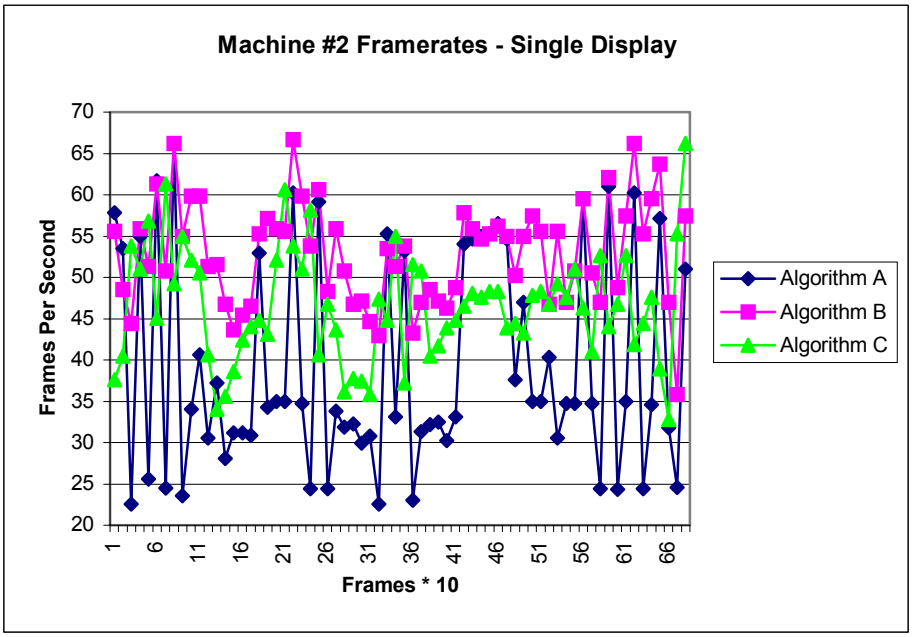

Figure 8.2: Machine #2 Framerates - Single Display

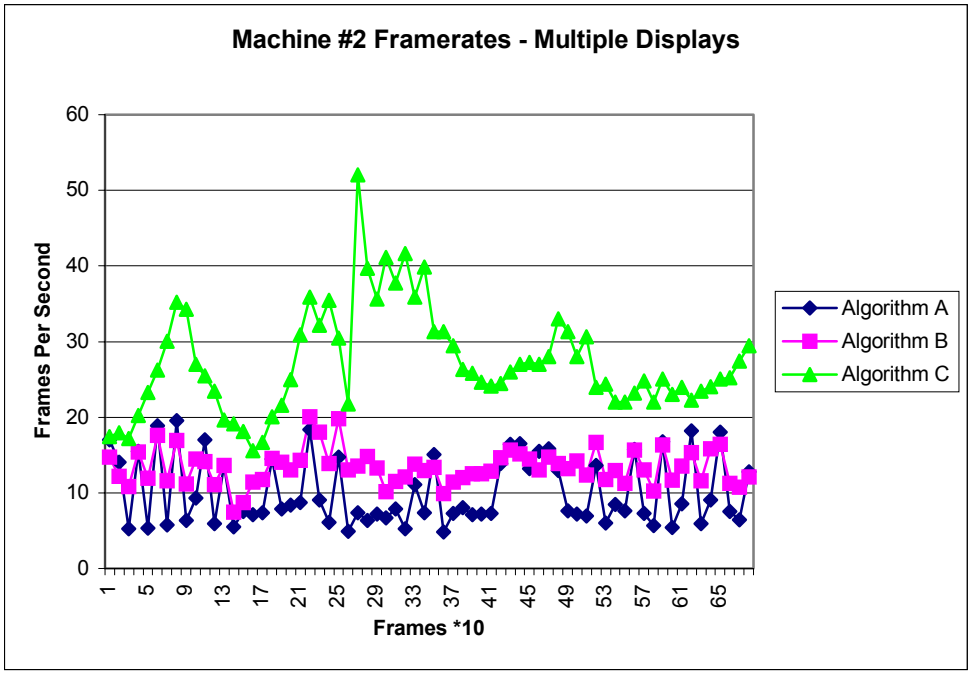

Figure 8.3: Machine #2 Framerates - Multiple Displays

CHAPTER IX

CONCLUSIONS

The addition of dynamic data loading greatly stabilized and improved the framerate. The intermittent jerks that were caused by the program loading new data were significantly reduced, and the average framerate was improved between 20% and 70% on various platforms.

The single tree recursion improved multiple display performance by 5% to 100% on the test platforms. The results varied widely because of the different speed processors and the different memory bus speeds. This optimization is mostly bound by the processor speed, since it has little to do with the video subsystems.

These two points prove that my thesis is correct. I was able to stabilize the framerate, increase the performance, and preserve the visual quality of the renderings by adding single tree recursion, dynamic data loading, and texture mapping.

These results could be improved further by adding the other extensions discussed in Duchaineau's ROAM paper, such as dual priority queues and incremental optimization [6]. Geomorphing would improve visual quality, while the split and merge queues would maintain a more consistent framerate. View dependant tessellation could improve visual clarity, while John Blow's sphere representation [11] could also significantly improve performance.

It is also important to note that while ROAM provided a convenient method for testing and implementing my thesis, very little of these additions are specific to ROAM. The single tree recursion could be added to any recursive algorithm such as Lindstrom's [5] or any quadtree algorithm. The dynamic data loading can also be added to several systems. Any system that uses a square tile-based data format, like quadtrees, could benefit from this improvement.

REFERENCES

[1] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH `93 Proc.),* pages 247-254, 1993.

[2] Claudio T. Silva, Joseph S. B. Mitchell, and Arie E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *Proc. Visualization `95,* pages 201-208, IEEE Computing Society Press, 1995.

[3] Daniel Cohen-Or and Yishay Levononi. Temporal continuity of levels of detail in delaunay triangulated terrain. In *Proc. Visualization `96*, pages 37-42. IEEE Computing Society Press, 1996.

[4] Hugues Hoppe. Progressive meshes. In *SIGGRAPH `96 Proc.*, pages 99-108, August 1996.

[5] Peter Lindstrom, David Killer, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH `96 Proc.*, pages 109-118, August 1996.

[6] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineed-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization'97*, pages 81-88, 1997.

[7] TreadMarks. LongBow Digital Arts. http://www.treadmarks.com.

[8] Benjamin Watson, Victoria Spaulding, Neff Walker, and William Ribarsky. Evaluation of the Effects of Frame Time Variation on VR Task Performance. In *Proc. IEEE Virtual Reality Annual Symposium*, pages 38-44, April 1997.

[9] M. Reddy. "A Survey of Level of Detail Support in Current Virtual Reality Solutions". In *Virtual Reality: Research, Development and Application*, pages:95-98, 1995.

[10] VRJuggler Programmer's Guide. Iowa State University, April, 2001. http://www.vrjuggler.org.

[11] J. Blow. Terrain Rendering at High Levels of Detail. *Proceedings of the 2000 Game Developers Conference*. March 2000.

[12] OpenGL Performer. Silicon Graphics Inc. http://www.sgi.com/software/performer/

[13] R. B. Pajarola. Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation. *IEEE Visualization `98*. pages 19-26, October 1998.

[14] S. Rottger, W. Heidrich, P. Slussalek, and H.-P. Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, pages 315-322, February 1998.

[15] Hughes Hoppe. Smooth View-Dependant Level-of-Detail control and its Application to Terrain Rendering *IEEE Visualization `98,* pages 35-42, October 1998.

[16] Peter Lindstrom and Valerio Pascucci. Visualization of Large Terrains Made Easy. Proceedings *IEEE Visualization 2001,* pages 363-370, October 2001.

[17] GTOPO30. U.S. Geological Survey EROS Data Center. http://edcdaac.usgs.gov/gtopo30/gtopo30.html July 2001.

[18] Martin, P. J., 2000. Description of the Navy Coastal Ocean Model Version 1.0, NRL Formal Report NRL/FR/7322-00-9962, Naval Research Laboratory, Stennis Space Center, MS, 39529.

[19] D. Pape, "A Hardware-Independent Virtual Reality Development System," in IEEE Computer Graphics and Applications, vol. 16, no. 4, pages 44-47, Jul. 1996.

APPENDIX A

SOURCE LISTINGS FOR ALGORITHM A

1. roamApp.h

```
/*
 * VRJuggler
 *   Copyright (C) 1997,1998,1999,2000
 *   Iowa State University Research Foundation, Inc.
 *   All Rights Reserved
 *
 * Original Authors:
 *   Allen Bierbaum, Christopher Just,
 *   Patrick Hartling, Kevin Meinert,
 *   Carolina Cruz-Neira, Albert Baker
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Library General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Library General Public License for more details.
 *
 * You should have received a copy of the GNU Library General Public
 * License along with this library; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 *
 * -----------------------------------------------------------------
 * File:          $RCSfile: roamApp.h,v $
 * Date modified: $Date: 2000/01/25 23:01:11 $
 * Version:       $Revision: 1.27 $
 * -----------------------------------------------------------------
 */

#ifndef _ROAM_APP_
#define _ROAM_APP_

#define SPEED 0.5

#define STARTX 400
#define STARTY 400

#define R_DATATYPE float
#include <vjConfig.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <vector>

#include <Kernel/GL/vjGlApp.h>
#include <Kernel/GL/vjGlContextData.h>
#include <Kernel/vjDebug.h>
```

```
#include <Math/vjMatrix.h>
#include <Math/vjVec3.h>
#include <Kernel/vjDebug.h>

#include <Input/InputManager/vjPosInterface.h>
#include <Input/InputManager/vjAnalogInterface.h>
#include <Input/InputManager/vjDigitalInterface.h>

#include <Kernel/vjUser.h>
#include <time.h>
#include <iostream>
#include <string>
using namespace std;


class roamApp;

#include "Utility.h"
#include "VisualComponent.h"
#include "UserData.h"
#include "data-mgr.h"
// Class to hold all data for a specific user

class BooleanData
{
      public:
            bool Enabled;
};

//-------------------------------------------------
// Demonstration OpenGL application class
//
// This application simply renders a field of roam.
//-------------------------------------------------
class roamApp : public vjGlApp
{
public:
   int Death;
   roamApp(vjKernel* kern) : vjGlApp(kern)
   {
      configFileName = "config.tile";
      Record = false;
      Playback = false;
      PathFileName = "";
   }

   virtual ~roamApp() {}

   // Execute any initialization needed before the API is started
   virtual void init();
   // Execute any initialization needed <b>after</b> API is started
   //  but before the drawManager starts the drawing loops.
```

```
   virtual void apiInit()
   {
      vjDEBUG(vjDBG_ALL,0) << "---- roamApp::apiInit() ----\n" <<
vjDEBUG_FLUSH;
   }

   // Called immediately upon opening a new OpenGL context
   void contextInit();

   /** Function to draw the scene
    * PRE: OpenGL state has correct transformation and buffer selected
    * POST: The current scene has been drawn
    */
   void draw()
   {
      initGLState();    // This should really be in another function

      myDraw(vjGlDrawManager::instance()->currentUserData()-
>getUser());
   }

   /**   name Drawing Loop Functions
    *
    *  The drawing loop will look similar to this:
    *
    *  while (drawing)
    *  {
    *        preFrame();
    *       draw();
    *        intraFrame();      // Drawing is happening while here
    *       sync();
    *        postFrame();      // Drawing is now done
    *
    *       UpdateTrackers();
     *  }
    *
    */

   /// Function called after tracker update but before start of drawing
   virtual void preFrame();

   /// Function called after drawing has been triggered but BEFORE it
completes
   virtual void intraFrame() {};

   /// Function called before updating trackers but after the frame is
drawn
   virtual void postFrame();

   //: Make sure that all our dependencies are satisfied
   // Make sure that there are vjUsers registered with the system
   virtual bool depSatisfied()
   {
```

```
      // We can't start until there are users registered wth the system
      // We rely upon users to keep track of the multi-user data
structure
      int num_users = vjKernel::instance()->getUsers().size();
      return (num_users > 0);
   }

   void LoadComponentList(void);
private:

   //-----------------------------------------------
   //  Draw the scene.  A bunch of boxes of differing color and stuff
   //-----------------------------------------------
   int datastatus, context_count;
   void myDraw(vjUser* user);
   void initGLState();
   float altitude;
   VisualComponentList VCs;
   DataManager DM;
      long frames;
      clock_t time_ticks;
public:
   std::vector<UserData*>        mUserData;     // All the users in the
program
   vjGlContextData<BooleanData>  InitFinished;
   string                configFileName;
   bool                        Record, Playback;
   string                PathFileName;
   //Kinda like a semaphore, but not really.
};

#endif
```

1. roamApp.cpp

```
#include "roamApp.h"

#ifdef CVS
#include <iostream>
#else
#include <iostream.h>
#endif
using namespace std;
#include <math.h>
#include <unistd.h>
#include <stdlib.h>

// Foundation classes for important stuff

// Include Visual Components
#include "Landscape.h"

// ------------------------------------------------------------------------
--------
// roamApp methods.
// ------------------------------------------------------------------------
--------

// Execute any initialization needed before the API is started
void roamApp::init()
{
   vjDEBUG(vjDBG_ALL,0) << "---------- roam:App:init() ---------------"
                        << endl << vjDEBUG_FLUSH;
   std::vector<vjUser*> users = kernel->getUsers();
   int num_users = users.size();
   vjASSERT(num_users > 0);         // Make sure that we actually have
users defined
   context_count=0;
   UserData* new_user=NULL;
   mUserData = std::vector<UserData*>(num_users);

   switch (num_users)
   {
   case (2):

      mUserData[1] = new_user;
      new_user->Death = Death;
      vjASSERT(users[1]->getId() == 1);
   case (1):
      new_user = new UserData(configFileName, users[0],
                        "VJHead", "VJWand", "VJButton0", "VJButton1",
                                  "VJButton2", "VJButton3",
"VJAnalog0", "VJAnalog1");
      mUserData[0] = new_user;
```

```
      new_user->Death = Death;
      vjASSERT(users[0]->getId() == 0);
      break;
   default:
      vjDEBUG(vjDBG_ALL,0) << "ERROR: Bad number of users." <<
vjDEBUG_FLUSH;
      exit();
      break;
   }

   if (Record) {
      mUserData[0]->SetOutputFile(PathFileName);
   } else if (Playback) {
      mUserData[0]->SetInputFile(PathFileName);
   }

   DM.Init(configFileName);
   LoadComponentList();
      frames = 0;
}

void roamApp::LoadComponentList()
{
   int ComponentCount = ReadIntFromFile(configFileName, "SYSTEM",
"ComponentCount", 0);
   string header;
   string type;
   char hdr[]="AddComponent100";
   VisualComponent *VC;

   VC = new VCRoamSurfaceFloat();
   VC->Init(&DM,mUserData[0], configFileName, "MAINROAM");
   VCs.AddComponent(VC);
}


// Called immediately upon opening a new OpenGL context
void roamApp::contextInit(void)
{
   InitFinished->Enabled = false;
   // Create display list
   vjDEBUG(vjDBG_ALL,2) << "---------- roam:App:contextInit() ---------
------"
                        << endl << vjDEBUG_FLUSH;
   VCs.InitGL();
   initGLState();
   InitFinished->Enabled = true;
      // Initialize the Environment's Contexts State
}

void roamApp::preFrame(void)
{
      float xpos, ypos, zpos;
```

```
      float seconds;
      static start_time=0;
      mUserData[0]->getPosition(&xpos, &ypos, &zpos);

        if (InitFinished->Enabled == false) {
                vjDEBUG(vjDBG_ALL,2) << "roamApp::preFrame() Canceled"
<< endl << vjDEBUG_FLUSH;
                return;
        }
        for(unsigned int i=0;i<mUserData.size();i++){
             mUserData[i]->updateNavigation();       // Update the
navigation matrix
        }

      VCs.PerFrameCalculations();
      if (frames==0) {
            time_ticks = time(NULL);
            start_time = glutGet(GLUT_ELAPSED_TIME);
      } else {
            if (frames == 10) {
                  ofstream fout("framerate.txt", ios::app);
                  float elapsed = (glutGet(GLUT_ELAPSED_TIME) -
start_time) * 0.001f;
                  fout << frames / elapsed << endl;
                  frames = 0;
                  start_time = glutGet(GLUT_ELAPSED_TIME);
                  fout.close();
            }
      }
      frames++;

}

//---------------------------------------------
// Draw the scene.  A bunch of boxes of
// differing color and stuff.
//---------------------------------------------
void roamApp::myDraw(vjUser* user)
{

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   if (InitFinished->Enabled == false) {
      vjDEBUG(vjDBG_ALL,2) << "\nContext Init not complete yet\n" <<
vjDEBUG_FLUSH;
      return;
   }
   glPushMatrix();
      glScalef(2.0, 1.0, 2.0);
         // --- Push on Navigation matrix for the user --- //
      vjMatrix nav_matrix = mUserData[user->getId()]->mNavMatrix;
      glMultMatrixf(nav_matrix.getFloatPtr());
```

```
//      glTranslatef(0, -10, 0);
      VCs.Render();

   glPopMatrix();

}

void roamApp::postFrame()
{
      VCs.PostRender();
}

void roamApp::initGLState()
{
   glEnable(GL_DEPTH_TEST);
   glDepthFunc(GL_LEQUAL);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   glMatrixMode(GL_MODELVIEW);
   glColor4f(1, 1, 1, 1);
   glEnable(GL_ALPHA_TEST);
   glAlphaFunc(GL_GREATER, 0.1);

// ----------- LIGHTING SETUP -----------
        // Light values and coordinates
        GLfloat  whiteLight[]    = { 0.45f,  0.45f, 0.45f, 1.0f };
        GLfloat  ambientLight[]  = { 0.25f,  0.25f, 0.25f, 1.0f };
        GLfloat  diffuseLight[]  = { 0.50f,  0.50f, 0.50f, 1.0f };
        GLfloat  lightPos[]      = { 0.00f,300.00f, 0.00f, 0.0f };

        // Setup and enable light 0
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
        glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
        glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
        glEnable(GL_LIGHT0);

        // Enable color tracking
        glEnable(GL_COLOR_MATERIAL);

        // Set Material properties to follow glColor values
        glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

        // Set the color for the landscape
        glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, whiteLight );

//        glFrontFace(GL_CCW);            // Counter clock-wise
polygons face out
//        glEnable(GL_CULL_FACE);         // Cull back-facing triangles
      glEnable(GL_TEXTURE_2D);
```

```
/*    glEnable(GL_FOG);
      glFogi(GL_FOG_MODE, GL_LINEAR);
      glFogi(GL_FOG_DENSITY, 0.8);
      glFogi(GL_FOG_START, 128.0);
      glFogi(GL_FOG_END, 256.0);
      glHint(GL_FOG_HINT, GL_FASTEST);
*/
}
```

2. Landscape.h

```
#ifndef _LANDSCAPE_H_
#define _LANDSCAPE_H_
#include <ioconn.h>
#include <tile.h>
#include <Kernel/GL/vjGlContextData.h>
#include <vector>
#include "Colormap.h"
#include "UserData.h"

// Patch & Landscape are so interconnected that all datatypes
//  pretty much need to be pre-declared
struct TriTreeNode
{
        TriTreeNode *LeftChild;
        TriTreeNode *RightChild;
        TriTreeNode *BaseNeighbor;
        TriTreeNode *LeftNeighbor;
        TriTreeNode *RightNeighbor;
};
template <class T>
class MetaData;
template <class T>
class ROAMLandScape;

#include "Patch.h"
#include "Utility.h"
#include "VisualComponent.h"

#define TRUE -1
#define FALSE 0

template <class T>
class MetaData {
public:
      int _patchsize, _viewsquare, variance_depth;
      int max_nodes;
      int map_size;

      float gFrameVariance;
      float _scale;
      int gDesiredTris;
      int gNumTrisRendered;
      int gDrawMode;
      int AutoLOD;
      ROAMLandScape<T> *land;
        int m_NextTriNode;
        TriTreeNode *m_TriPool;
      float Spread, MinValue, MinVariance;
      float height_dataMin, height_dataMax, height_dataSpread;
```

```
      float color_dataMin, color_dataMax, color_dataSpread;
      vjGlContextData<glstuff> Context;
      float gViewPosition[3];
      float TriVarianceList[500];
      Colormap cmap;
      int layer;
      UserData *user;
};

template <class T>
class Patch_Info {
public:
      Patch<T> *the_patch;
      int valid, updated;
};

template <class T>
class ROAMLandScape {
protected:
        datasource *TileManager;
      datasource *color_data;
        Patch<T> *PatchPool;
      Patch_Info<T> *m_Patches;

        int GetNextTriNode() { return info.m_NextTriNode; }
        void SetNextTriNode( int nNextNode )
            { info.m_NextTriNode = nNextNode; }
      void LoadConfigFile(string filename, string Header);
      void SetDrawModeContext();
public:
      ROAMLandScape(DataManager *DM, string filename, string Header);
      ~ROAMLandScape() {};

      float GetAltitude(float x, float y);
        TriTreeNode *AllocateTri();

        virtual void Init();
      virtual void InitGL();
        virtual void Reset();
        virtual void Tessellate();
        virtual void Render();
      virtual void PostRender();

      float getData(int x, int y);
      void MakeVisible(int x, int y);

      MetaData<T> info;
private:
      void DrawMap();
};

class VCRoamSurfaceFloat:public VisualComponent
{
```

```
    public:
        VCRoamSurfaceFloat(void);
        virtual ~VCRoamSurfaceFloat(void);

        virtual void Draw(void);
        virtual void PerFrameCalculations(void);
        virtual void PostRender(void);
        virtual int Status(void);
      virtual void InitGL(void);

      virtual void Init(DataManager *DM, UserData *userdata, string
filename, string header);
    private:
      UserData *user;
      ROAMLandScape<float> *myplot;
      string CfgFileName;
      string CfgHeader;
};

#endif
```

3. Landscape.cpp

```
#include <iostream>
#include <GL/glu.h>
#include <fstream>
#include <string>
#include <math.h>

using namespace std;

#include "Landscape.h"

VCRoamSurfaceFloat::VCRoamSurfaceFloat(void)
{
      myplot = NULL;
}

VCRoamSurfaceFloat::~VCRoamSurfaceFloat(void)
{
      if (myplot)
            delete (myplot);
}

void VCRoamSurfaceFloat::Draw(void)
{
        glPushMatrix();
        myplot->Render();

        glPopMatrix();
      glBindTexture(GL_TEXTURE_2D, 0);
}

void VCRoamSurfaceFloat::PerFrameCalculations(void)
{
      vjMatrix dummy1;
      vjVec3 location;

        dummy1 = user->getPosition();
        location=dummy1.getTrans();
        myplot->info.gViewPosition[0]=location[0];
        myplot->info.gViewPosition[1]=location[1];
        myplot->info.gViewPosition[2]=location[2];

        //myplot->info.gClipAngle = -gAnimateAngle;

        myplot->Reset();
        myplot->Tessellate();
}

void VCRoamSurfaceFloat::PostRender(void)
{
      myplot->PostRender();
```

```
}

int VCRoamSurfaceFloat::Status(void)
{
      return 0; // This component never deletes itself
}

void VCRoamSurfaceFloat::InitGL(void)
{
      myplot->InitGL();

}

void VCRoamSurfaceFloat::Init(DataManager *DM, UserData *userdata,
string filename, string header)
{
   CfgHeader = header;
   CfgFileName = filename;
   user = userdata;
   myplot = new ROAMLandScape<float>(DM , CfgFileName, header);
   myplot->info.gViewPosition[0] = ReadIntFromFile(CfgFileName,
"SYSTEM", "startx", 0);
   myplot->info.gViewPosition[1] = 0;
   myplot->info.gViewPosition[2] = ReadIntFromFile(CfgFileName,
"SYSTEM", "starty", 0);
   myplot->info.cmap.LoadColormap(ReadStringFromFile(CfgFileName,
header, "colormap", ""));
   myplot->info.cmap.SetLand(0, 1, 0, 1);
   myplot->info.user = user;
   myplot->Init();

}


template <class T>
ROAMLandScape<T>::ROAMLandScape(DataManager *DM, string filename,
string Header)
{
       LoadConfigFile(filename, Header);

      info.m_TriPool = new TriTreeNode[info.max_nodes];

      cout << "ROAM:constructor --> Building new Tile Manager" << endl;
      TileManager = DM->GetDataBase(ReadStringFromFile(filename,
Header, "data", ""));
      TileManager->setLayer(ReadIntFromFile(filename, Header,
"data_layer", 1));
      info.layer = TileManager->getLayer();

      info.land = this;
      if(ReadStringFromFile(filename, Header, "color", "") == "shared")
{
            cout << "--> Using Shared Colormap/Heightmap data" << endl;
```

```
            color_data = TileManager;
      } else {
            cout << "--> Not Using Shared Colormap/Heightmap data" <<
endl;
            color_data = DM->GetDataBase(ReadStringFromFile(filename,
Header, "color", ""));
            color_data->setLayer(ReadIntFromFile(filename, Header,
"color_layer", 1));
      }


      for(int index=0;index<VAR_MAX;index++)
            info.TriVarianceList[index] = 1.0 / (float)(index);
      m_Patches = NULL;
}

template <class T>
void ROAMLandScape<T>::LoadConfigFile(string filename, string Header)
{
      info._patchsize = ReadIntFromFile(filename, Header, "patchsize",
0);
      info._viewsquare = ReadIntFromFile(filename, Header,
"viewsquare", 0);
      info.variance_depth = ReadIntFromFile(filename, Header,
"variance_depth", 0);
      info.max_nodes = ReadIntFromFile(filename, Header, "max_nodes",
0);
      info.gDesiredTris = ReadIntFromFile(filename, Header,
"gDesiredTris", 0);
      info.map_size = ReadIntFromFile(filename, Header, "map_size", 0);
      info.gDrawMode = ReadIntFromFile(filename, Header, "render", 0);
      info.gFrameVariance = ReadIntFromFile(filename, Header,
"FrameVariance", 0);
      info.AutoLOD = ReadIntFromFile(filename, Header, "AutoLOD", 0);
      info.MinVariance = ReadFloatFromFile(filename, Header,
"MinVariance", 0.0);
}

template <class T>
void ROAMLandScape<T>::InitGL(void)
{
      // This used to create the land texture
      // Obviously now removed
}


// ------------------------------------------------------------------
-----------------------------
//      LANDSCAPE CLASS
// ------------------------------------------------------------------
-----------------------------
```

```
// ------------------------------------------------------------------
-
// Allocate a TriTreeNode from the pool.
//
template <class T>
TriTreeNode *ROAMLandScape<T>::AllocateTri()
{
        TriTreeNode *pTri;

        // IF we've run out of TriTreeNodes, just return NULL (this is
handled gracefully)
        if ( info.m_NextTriNode >= info.max_nodes )
                return NULL;

        pTri = &(info.m_TriPool[info.m_NextTriNode++]);
        pTri->LeftChild = pTri->RightChild = NULL;

        return pTri;
}

// ------------------------------------------------------------------
-
// Initialize all patches
//
template<class T>
void ROAMLandScape<T>::Init(void)
{
        Patch<T> *patch;
      Patch_Info<T> *ppatch;
        int X, Y;
      int diff;
      int sx_offset, sy_offset;
      int halfdist;
      TileManager->setLayer(info.layer);

      // Make sure we load the data centered around the user, so go
      // half of the viewable distance to the left,right,front, and
back of the user
      halfdist = ((info._viewsquare-1) * (info._patchsize)) >>1 ;

      diff = ((int)info.gViewPosition[0] % (info._patchsize));
      sx_offset = (info.gViewPosition[0]-diff) - halfdist;

      diff = ((int)info.gViewPosition[2] % (info._patchsize));
      sy_offset = (info.gViewPosition[2]-diff) - halfdist;

      // Now this is fucked up, I can't explain this
      //  But it doesn't work without it
//    if (sx_offset < 0) sx_offset--;
//    if (sy_offset < 0) sy_offset--;

        // Initialize all terrain patches
        // Store the Height Field array
```

```
      if (m_Patches == NULL) {
            // The patches haven't been allocated yet.. This means we
need
            // to allocate memory & load all the patches.
            m_Patches = new
Patch_Info<T>[info._viewsquare*info._viewsquare];
            PatchPool = new
Patch<T>[info._viewsquare*info._viewsquare];
            patch = &(PatchPool[0]);
            ppatch = &(m_Patches[0]);
            for ( Y=0; Y < info._viewsquare; Y++)
                    for ( X=0; X < info._viewsquare; X++ )
                    {
                  ppatch->the_patch = patch;
                  ppatch->valid = TRUE;
                  ppatch->updated = TRUE;
                        patch->Init(sx_offset + X*(info._patchsize),
                              sy_offset+ Y*(info._patchsize),
                              info._patchsize,
                              &info);
                  // Load data now.. If no data is found, then this
patch is
                  // Invalid (outside data-able area)
                  if(TileManager->loadArea(sx_offset +
X*(info._patchsize),
                                    sy_offset + Y*(info._patchsize),
                                          info._patchsize+1,
                                    info._patchsize+1,
                                          patch->height_data) == -1 )
                        ppatch->valid = FALSE;
                  else  {
                        color_data->loadArea(sx_offset +
X*(info._patchsize),
                                                sy_offset +
Y*(info._patchsize),
                                                info._patchsize+1,
                                                info._patchsize+1,
                                                patch->colormap_data);
                              patch->ComputeVariance();
                  }
                  patch++;
                  ppatch++;
                    }
      } else {
            // The patches have been previously allocated, so they
contain data.
            // Find the valid data to preserve, and load the rest.
            // Algorithm:
            // First search all the data tiles.  If one is "Valid",
swap it to it's correct
            //    location and mark it valid.
            // Then pass through all "invalid" tiles and load them.
            int left, right, top, bottom;
```

```
            int patch_x, patch_y;
            int p2x, p2y;
            Patch<T> *patch_temp;
            Patch_Info<T> *dpatch; // Destination Patch

            left = sx_offset;
            top = sy_offset;
            right = left + (info._viewsquare-1)*(info._patchsize);
            bottom = top + (info._viewsquare-1)*(info._patchsize);

            // Start by marking them all invalid for later checks
            ppatch = &(m_Patches[0]);
            for ( X=0; X < info._viewsquare*info._viewsquare; X++,
ppatch++)  {
                  ppatch->updated = FALSE;
                  ppatch->valid = FALSE;
            }

            // Now we know which ones are valid/invalid..
            // So reload the invalid ones.
                ppatch = &(m_Patches[0]);
                for ( Y=0; Y < info._viewsquare; Y++)
                    for ( X=0; X < info._viewsquare; X++, ppatch++ )
                  if(ppatch->valid == FALSE) {
                        // This patch isn't valid.. so reload.
                        ppatch->updated = TRUE;
                              (ppatch->the_patch)->Init(sx_offset +
X*(info._patchsize),
                                        sy_offset+ Y*(info._patchsize),
                                        info._patchsize,
                                        &info);
                              if(TileManager->loadArea(sx_offset +
X*(info._patchsize),
                                              sy_offset +
Y*(info._patchsize),
                                              info._patchsize+1,
                                              info._patchsize+1,
                                              (ppatch->the_patch)-
>height_data) == -1) {
                              ppatch->valid = FALSE;
                        } else  {
                              color_data->loadArea(sx_offset +
X*(info._patchsize),
                                              sy_offset +
Y*(info._patchsize),
                                              info._patchsize+1,
                                              info._patchsize+1,
                                              (ppatch->the_patch)-
>colormap_data);
                                    (ppatch->the_patch)-
>ComputeVariance();
                              ppatch->valid = TRUE;
                        }
```

```
                }

        }
        info.height_dataMin = TileManager->getMin();
        info.height_dataMax = TileManager->getMax();
        if (info.height_dataMax < info.height_dataMin) {
              T tmp = info.height_dataMin;
              info.height_dataMin = info.height_dataMax;
              info.height_dataMax = tmp;
        }
        info.color_dataMin = TileManager->getMin();
        info.color_dataMax = TileManager->getMax();
        if (info.color_dataMax < info.color_dataMin) {
              T tmp = info.color_dataMin;
              info.color_dataMin = info.color_dataMax;
              info.color_dataMax = tmp;
        }

        info.cmap.SetMaxMin(info.color_dataMax, info.color_dataMin);
        ppatch = &(m_Patches[0]);
          for ( Y=0; Y < info._viewsquare*info._viewsquare; Y++){
              if((ppatch->valid == TRUE) && (ppatch->updated == TRUE))
                    (ppatch->the_patch)->GenerateTexture();
              ppatch++;
        }
        info.height_dataSpread = info.height_dataMax -
info.height_dataMin;
        info.height_dataSpread = info.height_dataMax -
info.height_dataMin;
}

template <class T>
void ROAMLandScape<T>::Reset(void)
{
          //
          // Perform simple visibility culling on entire patches.
          //   - Define a triangle set back from the camera by one patch
size, following
          //     the angle of the frustum.
          //   - A patch is visible if any of it's 4 corners are within
the viewable area
        //          -- uses gluUnProject to figure out
          //   - This visibility test is only accurate if the camera
cannot look up or down significantly.
          //
          int X, Y, size;
        int startx, starty;
          Patch_Info<T> *patch;
        size = info._viewsquare;
        Patch_Info<T> *pLeft, *pRight, *pTop, *pBottom;

          // Set the next free triangle pointer back to the beginning
          SetNextTriNode(0);
```

```
      // See if we are outside the center patch
      patch = &m_Patches[(size * size) >> 1]; // set Patch to center
patch
                                    // NOTE: Only works when size is an
odd number
      if (patch->valid == TRUE) {
          (patch->the_patch)->getLocation(&startx, &starty);
          if ((info.gViewPosition[0] < startx) ||
(info.gViewPosition[0] > startx+info._patchsize+1) ||
            (info.gViewPosition[2] < starty) || (info.gViewPosition[2]
> starty+info._patchsize+1)) {
            // We are outside the center patch, so reload data to put
us inside the center patch
            vjDEBUG(vjDBG_ALL, 1) << "User at: " <<
info.gViewPosition[0] << "," <<
                  info.gViewPosition[2] << endl << vjDEBUG_FLUSH;
            vjDEBUG(vjDBG_ALL, 1) << "Center tile at: " << startx <<
"," << starty << endl << vjDEBUG_FLUSH;
            vjDEBUG(vjDBG_ALL, 1) << "Not Centered. Reloading\n" <<
endl << vjDEBUG_FLUSH;
            // Insert code to swap stuff out HERE
            Init();
          }
      }
        // Reset rendered triangle count.
        info.gNumTrisRendered = 0;
      patch=&(m_Patches[0]);
        // Go through the patches performing resets, compute variances,
and linking.
        for ( Y=0; Y < info._viewsquare; Y++ )
                for ( X=0; X < info._viewsquare; X++)
                {
                        // Reset the patch
                if (patch->valid == TRUE) { // If patch is invalid,
then don't bother rebuilding it
                                    // Prevents "edge of the world" bug
                                    // Edge of world is now darkness..
Just like the europeans thought :)
                        (patch->the_patch)->Reset();

                        // Check to see if this patch has been deformed
since last frame.
                        // If so, recompute the varience tree for it.
                        if ( (patch->the_patch)->isDirty() )
                                (patch->the_patch)->ComputeVariance();
                  pTop = &(m_Patches[(Y-1)*size + X]);
                  pBottom = &(m_Patches[(Y+1)*size +X]);
                  pLeft = &(m_Patches[(Y*size) + X-1]);
                  pRight = &(m_Patches[(Y*size) + X+1]);

                        // Link all the patches together.
                        if ((X > 0) && (pLeft->valid == TRUE) )
```

```
                                (patch->the_patch)->GetBaseLeft()-
>LeftNeighbor = pLeft->the_patch->GetBaseRight();
                        else
                                (patch->the_patch)->GetBaseLeft()-
>LeftNeighbor = NULL;              // Link to bordering Landscape
here..

                        if (( X < (size-1)) && (pRight->valid == TRUE)
)
                                (patch->the_patch)->GetBaseRight()-
>LeftNeighbor = pRight->the_patch->GetBaseLeft();
                        else
                                (patch->the_patch)->GetBaseRight()-
>LeftNeighbor = NULL;              // Link to bordering Landscape here..

                        if (( Y > 0) && (pTop->valid == TRUE) )
                                (patch->the_patch)->GetBaseLeft()-
>RightNeighbor = pTop->the_patch->GetBaseRight();
                        else
                                (patch->the_patch)->GetBaseLeft()-
>RightNeighbor = NULL;              // Link to bordering Landscape
here..

                        if (( Y < (size-1)) && (pBottom->valid == TRUE)
)
                                (patch->the_patch)->GetBaseRight()-
>RightNeighbor = pBottom->the_patch->GetBaseLeft();
                        else
                                (patch->the_patch)->GetBaseRight()-
>RightNeighbor = NULL;     // Link to bordering Landscape here..
                }
                patch++;
                }
}


// ------------------------------------------------------------------------
-
// Create an approximate mesh of the landscape.
//
template<class T>
void ROAMLandScape<T>::Tessellate()
{
        // Perform Tessellation
      int x, count;
        Patch_Info<T> *patch = &(m_Patches[0]);
      count = info._viewsquare * info._viewsquare;

        for(x=0;x<count; x++, patch++){
            if (patch->valid == TRUE)
                  (patch->the_patch)->Tessellate( );
      }
        for(patch = &(m_Patches[0]),x=0;x<count; x++, patch++){
```

```
            if (patch->valid == TRUE)
                  (patch->the_patch)->BuildRender();
        }
}

template <class T>
float ROAMLandScape<T>::GetAltitude(float x, float y)
{
        float corners[2][2], topApprox, botApprox;
        float ix, iy;
      ix = ftrunc(x);
      iy = ftrunc(y);
        corners[0][0] = getData(ix, iy);
        corners[0][1] = getData(ix+1, iy);
        corners[1][0] = getData(ix, iy+1);
        corners[1][1] = getData(ix+1, iy+1);

        topApprox = corners[0][0] +
                (x - ix)*(corners[0][1] - corners[0][0]);
        botApprox = corners[1][0] +
                (x - ix)*(corners[1][1] - corners[1][0]);
        return(topApprox +
                (y - iy)*(botApprox - topApprox));

}

// ------------------------------------------------------------------------
-
// Render each patch of the landscape & adjust the frame variance.
//
template <class T>
void ROAMLandScape<T>::Render()
{
        int x,y ;
        Patch_Info<T> *patch = &(m_Patches[0]);
      GLdouble projection[16], model[16];
      GLint viewport[4];

      glDisable(GL_LIGHTING);
      glGetDoublev(GL_PROJECTION_MATRIX,projection);
      glGetDoublev(GL_MODELVIEW_MATRIX, model);
      glGetIntegerv(GL_VIEWPORT,viewport);
      // Set visibility on patches (View Frustrum Culling)

for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare; x++,
patch++)
                if (patch->valid == TRUE)
                  (patch->the_patch)->SetVisibility(projection, model,
viewport);
      // Force nearby patches to be visible (reduce blicker);
      //  (for the unknowing.. blicker = flicker + blink )
      //  Blicker (c) Randall Hand 2001
      for(x=-1;x<=1; x++) {
```

```
            for(y=-1; y<=1; y++) {

      MakeVisible(info.gViewPosition[0]+(x*(info._patchsize)),
                        info.gViewPosition[2]+(y*(info._patchsize)));
            }
      }
      // Render Visible Patches
      if ((info.user)->Render_Wireframe) {
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            glEnable(GL_CULL_FACE);
      } else {
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      }

for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare; x++,
patch++){
            if ((patch->valid == TRUE) && ((patch->the_patch)-
>isVisible()))   {
                  (patch->the_patch)->Render();
            }
        }

      glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      glDisable(GL_CULL_FACE);
      DrawMap();
}

template <class T>
void ROAMLandScape<T>::DrawMap(void)
{
      vjVec3 angles;
      int x;
        Patch_Info<T> *patch = &(m_Patches[0]);

        glDisable(GL_DEPTH_TEST);
      for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare
; x++, patch++){
            if(patch->valid == TRUE) {
                  glPushMatrix();
                  glLoadIdentity();
                  glTranslatef(0, 5, 0);
                  glScalef(0.5,0.5,0.5);
                  glTranslatef((x % info._viewsquare), 10-
int(x/info._viewsquare),-15);
                  (patch->the_patch)->RenderTexPanel();
                  glPopMatrix();
            }
      }

      glPushMatrix();
      glLoadIdentity();
      glTranslatef(0,5,0);
      glScalef(0.5,0.5,0.5);
```

```
      glTranslatef((float)info._viewsquare/2.0, 11.0 -
((float)info._viewsquare/2.0), -15);
      angles = (info.user)->GetUserOrientation();
      if (angles[0] == -180) {
            glRotatef(-(180-angles[1]), 0,0,1);
      } else {
            glRotatef(-angles[1],0,0,1);
      }
      glBindTexture(GL_TEXTURE_2D, 0);
      glColor4f(1,1,1,1);
      glBegin(GL_TRIANGLES);
         glVertex3f(0,0,0);
         glColor4f(1,1,1,0);
         glVertex3f(-2, 3, 0);
         glVertex3f(2, 3, 0);
      glEnd();
      glPopMatrix();


        glEnable(GL_DEPTH_TEST);

}



template <class T>
void ROAMLandScape<T>::PostRender(void)
{
        // Check to see if we got close to the desired number of
triangles.
        // Adjust the frame variance to a better value.
      int x;
      Patch<T> *patch;
      if (info.AutoLOD == 0)
            return;
        if ( GetNextTriNode() != info.gDesiredTris )
                info.gFrameVariance += ((float)GetNextTriNode() -
(float)info.gDesiredTris) / (float)info.gDesiredTris;

        // Bounds checking.
        if ( info.gFrameVariance < 0 )
                info.gFrameVariance = 0;


for(x=0,patch=&(PatchPool[0]);x<info._viewsquare*info._viewsquare; x++,
patch++){
              patch->ClearFlags();
      }
}

template <class T>
void ROAMLandScape<T>::SetDrawModeContext()
{
```

```
        switch (info.gDrawMode)
        {
        case DRAW_USE_TEXTURE:
                glDisable(GL_LIGHTING);
                glEnable(GL_TEXTURE_2D);
        //        glPolygonMode(GL_FRONT, GL_FILL);
                break;

        case DRAW_USE_LIGHTING:
                glEnable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_FILL);
                break;

        case DRAW_USE_FILL_ONLY:
                glDisable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_FILL);
                break;

        default:
        case DRAW_USE_WIREFRAME:
                glDisable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_LINE);
                break;
        }
}

template <class T>
float ROAMLandScape<T>::getData(int x, int y)
{
      int size;
      int startx, starty;
      int px, py;
        Patch_Info<T> *patch = &(m_Patches[0]);
      size = info._patchsize;

      (patch->the_patch)->getLocation(&startx, &starty);
      px = (x - startx) / size;
      py = (y - starty) / size;

      patch = &(m_Patches[(py * info._viewsquare) + px]);
      (patch->the_patch)->getLocation(&startx, &starty);

      return (patch->the_patch)->getData(x-startx, y-starty);
}

template <class T>
void ROAMLandScape<T>::MakeVisible(int x, int y)
{
      int size;
      int startx, starty;
```

```
        int px, py;
          Patch_Info<T> *patch = &(m_Patches[0]);
        size = info._patchsize;

        (patch->the_patch)->getLocation(&startx, &starty);
        px = (x - startx) / size;
        py = (y - starty) / size;

        patch = &(m_Patches[(py * info._viewsquare) + px]);
        (patch->the_patch)->ForceVisibility();
}
```

4. Patch.h

```
#ifndef _PATCH_H_
#define _PATCH_H_
#include <tile-mgr.h>
#include <ioconn.h>
#include <tile.h>
#include <Kernel/GL/vjGlContextData.h>
#include <vector>

struct glstuff {
      GLuint texLand;
};
struct patchstuff {
      unsigned char m_isVisible;
      GLuint tex_id;
      int x_pos, y_pos;
};

typedef struct TriangleInfo {
      GLfloat vertex[3][3];
      GLfloat texture[3][2];
      GLfloat normal[3];
      GLfloat color[3][3];
} TriangleInfo;
template <class T>
class Patch;

#include "Landscape.h"

#define DRAW_USE_TEXTURE   (0)
#define DRAW_USE_LIGHTING  (1)
#define DRAW_USE_FILL_ONLY (2)
#define DRAW_USE_WIREFRAME (3)

#define VAR_MAX 300.0

template <class T>
class Patch{
public:
      Patch(void); //  Simple constructor
      TriTreeNode *GetBaseLeft() {return &m_BaseLeft;}
      TriTreeNode *GetBaseRight() { return &m_BaseRight; }
        virtual void Init( int worldX, int worldY,
                     int size, MetaData<T> *ls_info );
        virtual void Reset();
        virtual void Tessellate();
        virtual void Render();
      virtual void BuildRender();
        virtual void ComputeVariance();
        // The recursive half of the Patch Class
```

```
        virtual void Split( TriTreeNode *tri);
        virtual void RecursTessellate( TriTreeNode *tri,
                  int leftX, int leftY,
                  int rightX, int rightY,
                  int apexX, int apexY,
                  int node );
        virtual void RecursRender( TriTreeNode *tri,
                  int leftX, int leftY,
                  int rightX, int rightY,
                  int apexX, int apexY);
        virtual float RecursComputeVariance(
                  int leftX,  int leftY,  T leftZ,
                        int rightX, int rightY, T rightZ,
                        int apexX,  int apexY,  T apexZ,
                        int node);
        char isDirty()     { return m_VarianceDirty; }
        int  isVisible()  { return stuff->m_isVisible; }
      void getLocation(int *x, int *y) { *x=m_WorldX; *y=m_WorldY; };
      float getData(int x, int y);
      void SetVisibility( GLdouble *proj_matrix, GLdouble
*model_matrix, GLint *viewport );
      void ForceVisibility() { stuff->m_isVisible = 1;};
            // Added 1-9-2k1 REH
      void ClearFlags(void);
      void GenerateTexture(void);
      void BindNewTexture(void);
      void RenderTexPanel(void);
      T *height_data;   // This is stupid, I know. I'll fix it later.
      T *colormap_data;
private:
      int m_WorldX, m_WorldY;
      TriTreeNode m_BaseLeft, m_BaseRight;
      float  *m_VarianceLeft, *m_VarianceRight;
      float  *m_CurrentVariance;
        unsigned char m_VarianceDirty;
      vjGlContextData<struct patchstuff> stuff;
        int width, map_size;
      MetaData<T> *info;
      TriangleInfo *Triangle;
      long TriangleIndex;
      unsigned char *pTexture;
      int flg_NewTexture;
};

#endif
```

5. Patch.cpp

```
#include <iostream>
#include <GL/glu.h>
#include <fstream>
#include <string>
#include <math.h>
#include "Utility.h"
using namespace std;

#include "Patch.h"

// Added 1-9-2k1 REH
template <class T>
Patch<T>::Patch(void)
{
      // Simple Constructor to initialize a few vars
      flg_NewTexture = 0;
      pTexture = NULL;
      height_data = NULL;
      colormap_data = NULL;
}

template <class T>
void Patch<T>::ClearFlags(void)
{
      // This function is called after the Draw routines to reset any
flags
      // Namely, the "There is a new texture" flag (flg_NewTexture)
      // Called per-frame, not per-context
      flg_NewTexture = 0;
}

template <class T>
void Patch<T>::GenerateTexture(void)
{
      // Generate the texture for this tile,
      // Memory is allocated in Init.
      // This routine is called from the Patch's Init, whenever new
data is loaded
      // The texture size will match the patch's size
      //   (The patch size is already ^2's, so this should work nicely.
      //    Let the video system handle color smoothing and such)
      // NOTE: At a later date, add in extra pixels (use 2x2 or 4x4
squares)
      //       and support software based lighting calculations.)
      T *d;
      unsigned char *tex;
      RGBA color;

      d = &colormap_data[width+1]; // skip the top row & left column
      tex = pTexture;
```

```
      for(int x=0; x<(width-1)*(width-1); x++) {
            info->cmap.GetColorRGBA((double)(*d), &color);
            *(tex+0) = color[0];
            *(tex+1) = color[1];
            *(tex+2) = color[2];
            *(tex+3) = color[3];
            tex+=4;
            d++;
            if ((d-colormap_data) % width == 0) d++; // continue to
skip the left column of data
      }
      flg_NewTexture = 1;
}

template <class T>
void Patch<T>::BindNewTexture(void)
{
      // Bind this texture, deleting/unbinding old texture if necessary
      // This routine is called once for each GL context
      // Delete any textures if there are any
      if (glIsTexture(stuff->tex_id) == GL_TRUE) {
            glDeleteTextures(1, &(stuff->tex_id));
      }

      // Now bind the new texture
        glEnable(GL_TEXTURE_2D);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

                    // Generate OpenGL texture IDs.
        glGenTextures(1, &(stuff->tex_id));

        glBindTexture(GL_TEXTURE_2D, stuff->tex_id);

        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, // RGBA textures.
             width-1, width-1, 0, GL_RGBA,
             GL_UNSIGNED_BYTE,
             pTexture);

}
// End of 1-9-2k1 changes

template <class T>
void Patch<T>::Split(TriTreeNode *tri)
{
      //We are already split, so there is no need to split again
```

```
      if (tri->LeftChild)
            return;
      //If this triangle is not in a proper diamond,
      //    force split the base neighbor.
      // This means, if this triangle's base is not shared as the base
of
      // another triangle (This triangle leans on the side of another,
      // not on the bottom), then if we split this one, we must split
all
      // the way back to a base.
      if (tri->BaseNeighbor && (tri->BaseNeighbor->BaseNeighbor !=
tri))
            Split(tri->BaseNeighbor);

      // Setup the two childrenfor the split
      tri->LeftChild = info->land->AllocateTri();
      tri->RightChild = info->land->AllocateTri();

      // If there was no available nodes for the children, then
      // exit gracefully.
      if (!tri->LeftChild)
            return;
      if (!tri->RightChild)
            return;
      // Point the new child to the current base & neighbors
      tri->LeftChild->BaseNeighbor  = tri->LeftNeighbor;
        tri->LeftChild->LeftNeighbor  = tri->RightChild;

        tri->RightChild->BaseNeighbor  = tri->RightNeighbor;
        tri->RightChild->RightNeighbor = tri->LeftChild;

      // Point the left neighbor directly to the newly created children
      if (tri->LeftNeighbor != NULL) {
            // One of the left's neighbor's neighbor-pointers should
            // point to me.. Find out which it is, and point it at
            // the new child
            if (tri->LeftNeighbor->BaseNeighbor == tri)
                        tri->LeftNeighbor->BaseNeighbor = tri-
>LeftChild;
                else if (tri->LeftNeighbor->LeftNeighbor == tri)
                        tri->LeftNeighbor->LeftNeighbor = tri-
>LeftChild;
                else if (tri->LeftNeighbor->RightNeighbor == tri)
                        tri->LeftNeighbor->RightNeighbor = tri-
>LeftChild;
                else
                        ;// Illegal Left Neighbor!
        }
        // Link our Right Neighbor to the new children
        if (tri->RightNeighbor != NULL)
        {
                if (tri->RightNeighbor->BaseNeighbor == tri)
```

```
                        tri->RightNeighbor->BaseNeighbor = tri-
>RightChild;
                else if (tri->RightNeighbor->RightNeighbor == tri)
                        tri->RightNeighbor->RightNeighbor = tri-
>RightChild;
                else if (tri->RightNeighbor->LeftNeighbor == tri)
                        tri->RightNeighbor->LeftNeighbor = tri-
>RightChild;
                else
                        ;// Illegal Right Neighbor!
        }

        // Link our Base Neighbor to the new children
        if (tri->BaseNeighbor != NULL)
        {
                if ( tri->BaseNeighbor->LeftChild )
                {
                        tri->BaseNeighbor->LeftChild->RightNeighbor =
tri->RightChild;
                        tri->BaseNeighbor->RightChild->LeftNeighbor =
tri->LeftChild;
                        tri->LeftChild->RightNeighbor = tri-
>BaseNeighbor->RightChild;
                        tri->RightChild->LeftNeighbor = tri-
>BaseNeighbor->LeftChild;
                }
                else
                        Split( tri->BaseNeighbor);  // Base Neighbor
(in a diamond with us) was not split yet, so do that now.
        }
        else
        {
                // An edge triangle, trivial case.
                tri->LeftChild->RightNeighbor = NULL;
                tri->RightChild->LeftNeighbor = NULL;
        }
}

// ---------------------------------------------------------------------
-
// Tessellate a Patch.
// Will continue to split until the variance metric is met.
//
template <class T>
void Patch<T>::RecursTessellate( TriTreeNode *tri,
                                                                int leftX,
int leftY,
                                                                int rightX,
int rightY,
                                                                int apexX,
int apexY,
                                                                int node )
{
```

```
        float TriVariance, d_num;
        int centerX = (leftX + rightX)>>1; // Compute X coordinate of
center of Hypotenuse
        int centerY = (leftY + rightY)>>1; // Compute Y coord...

        if ( node < (1<<info->variance_depth) )
        {
                // Extremely slow distance metric (sqrt is used).
                // Replace this with a faster one!
                float distance = 1.0f + ( SQR((float)centerX - info-
>gViewPosition[0]) +
                                                SQR((float)centerY -
info->gViewPosition[2]) );
            if (distance>=VAR_MAX)
                  d_num = 1.0/distance; //info-
>TriVarianceList[(int)VAR_MAX-1];
            else
                  d_num = info->TriVarianceList[(int)distance];

                // Egads!  A division too?  What's this world coming
to!
                // This should also be replaced with a faster
operation.
                TriVariance = ((float)m_CurrentVariance[node] * info-
>map_size * 2) * d_num;;
                  //* d_num; // Take both distance and variance into
consideration
        }

        if ( (node >= (1<<info->variance_depth)) || (TriVariance >
info->gFrameVariance))
        {
                Split(tri); // Split this triangle.

                if (tri->LeftChild &&  // If this triangle was split,
try to split it's children as well.
                          ((abs(leftX - rightX) >= 3) || (abs(leftY -
rightY) >= 3)))   {
                        RecursTessellate( tri->LeftChild,   apexX,
apexY, leftX, leftY, centerX, centerY,    node<<1  );
                        RecursTessellate( tri->RightChild, rightX,
rightY, apexX, apexY, centerX, centerY, 1+(node<<1) );
                }
        }
}

// ----------------------------------------------------------------------
-
// Render the tree.  Simple no-fan method.
//
template <class T>
void Patch<T>::RecursRender( TriTreeNode *tri, int leftX, int leftY,
int rightX, int rightY, int apexX, int apexY )
```

```
{
      TriangleInfo *t;
        if ( tri->LeftChild )                                        // All
non-leaf nodes have both children, so just check for one
        {
                int centerX = (leftX + rightX)>>1;      // Compute X
coordinate of center of Hypotenuse
                int centerY = (leftY + rightY)>>1;      // Compute Y
coord...

                RecursRender( tri->LeftChild,  apexX,   apexY, leftX,
leftY, centerX, centerY );
                RecursRender( tri->RightChild, rightX, rightY, apexX,
apexY, centerX, centerY );
        }
        else
// A leaf node!  Output a triangle to be rendered.
        {
                // Actual number of rendered triangles...
                RGBA color;
                info->gNumTrisRendered++;
                GLfloat leftZ  = height_data[((leftY) *width)+leftX ];
                GLfloat rightZ = height_data[((rightY)*width)+rightX];
                GLfloat apexZ  = height_data[((apexY) *width)+apexX ];

                // Perform polygon coloring based on a height sample
                glBegin(GL_TRIANGLES);
                        info->cmap.GetColorRGBA(leftZ, &color);
                        glColor3ub( color[0], color[1], color[2] );
                        glVertex3f(leftX, leftZ, leftY);

                        info->cmap.GetColorRGBA(rightZ, &color);
                        glColor3ub( color[0], color[1], color[2] );
                        glVertex3f(rightX, rightZ, rightY);

                        info->cmap.GetColorRGBA(apexZ, &color);
                        glColor3ub( color[0], color[1], color[2] );
                        glVertex3f(apexX, apexZ, apexY);
                glEnd();

        }
}

// ----------------------------------------------------------------------
-
// Computes Variance over the entire tree.  Does not examine node
relationships.
// NOTE: left, right & APex are in LOCAL coordinates (locations within
the patch)
template <class T>
float Patch<T>::RecursComputeVariance( int leftX,  int leftY,  T leftZ,
                                       int rightX, int rightY, T rightZ,
                                       int apexX,  int apexY,  T apexZ,
```

```
                                        int node)
{
        //        /|\
        //      /  |  \
        //    /    |    \
        //  /      |      \
        //  ~~~~~~~*~~~~~~~  <-- Compute the X and Y coordinates of '*'
        //
        int centerX = (leftX + rightX) >>1;             // Compute X
coordinate of center of Hypotenuse
        int centerY = (leftY + rightY) >>1;             // Compute Y
coord...
        float myVariance;
      float tempvar;

        // Get the height value at the middle of the Hypotenuse
        T centerZ  = height_data[((centerY) * width) + centerX];

        // Variance of this triangle is the actual height at it's
hypotenuse midpoint minus the interpolated height.
        // Use values passed on the stack instead of re-accessing the
Height Field.
        myVariance = abs(centerZ - ((int)(leftZ + rightZ)>>1));
        // Since we're after speed and not perfect representations,
        //    only calculate variance down to an 8x8 block
        if ( (abs(leftX - rightX) >= 8) ||
                  (abs(leftY - rightY) >= 8) )
        {
                // Final Variance for this node is the max of it's own
variance and that of it's children.
            tempvar = RecursComputeVariance( apexX,   apexY,  apexZ,
                                        leftX, leftY, leftZ,
                                        centerX, centerY, centerZ,
                                        node<<1 );
                myVariance = MAX( myVariance,tempvar);
      //Left Child Node

            tempvar = RecursComputeVariance( rightX, rightY, rightZ,
                                        apexX, apexY, apexZ,
                                        centerX, centerY, centerZ,
                                        1+(node<<1));
                myVariance = MAX( myVariance,tempvar);
      //Right Child Node
        }
        // Store the final variance for this node.  Note Variance is
never zero.
        if (node < (1<<info->variance_depth)) {
                m_CurrentVariance[node] = info->MinVariance +
myVariance;
      }
        return myVariance;
}
```

```
// ------------------------------------------------------------------------
----------------------------
//      PATCH CLASS
// ------------------------------------------------------------------------
----------------------------

// ------------------------------------------------------------------------
-
// Initialize a patch.
//
template <class T>
void Patch<T>::Init(int worldX, int worldY, int size, MetaData<T>
*ls_info )
{
        // Clear all the relationships
        m_BaseLeft.RightNeighbor = m_BaseLeft.LeftNeighbor =
m_BaseRight.RightNeighbor = m_BaseRight.LeftNeighbor =
            m_BaseLeft.LeftChild = m_BaseLeft.RightChild =
m_BaseRight.LeftChild = m_BaseLeft.LeftChild = NULL;

        // Attach the two m_Base triangles together
        m_BaseLeft.BaseNeighbor = &m_BaseRight;
        m_BaseRight.BaseNeighbor = &m_BaseLeft;

      info = ls_info;
        // Store Patch offsets for the world and heightmap.
        m_WorldX = worldX;
        m_WorldY = worldY;
      if (width != size) {
              width = size+1;
              // Store pointer to first byte of the height data for
this patch.
            if (height_data) {
                  delete height_data;
            }
            height_data=new T[(size+1) * (size+1)];
            if (colormap_data) {
                  delete colormap_data;
            }
            colormap_data=new T[(size+1) * (size+1)];
      }

      Triangle = new TriangleInfo[(size+1) * (size+1)*2];
      TriangleIndex = 0;

      m_VarianceLeft = new float[1<<info->variance_depth];
        m_VarianceRight = new float[1<<info->variance_depth];

        // Initialize flags
        m_VarianceDirty = 1;

      // Allocate Texture
```

```
	if (pTexture!=NULL) {
		delete pTexture;
	}
	pTexture = new unsigned char[size * size *4]; // 4 bytes for RGBA
	if (pTexture == NULL) {
		cout << "ERROR: Unable to allocate RAM" << endl;
	}
}

template <class T>
float Patch<T>::getData(int x, int y)
{
	return height_data[(y * width) + x];
}

// ---------------------------------------------------------------------
-
// Reset the patch.
//
template <class T>
void Patch<T>::Reset()
{
	// Reset the important relationships
	m_BaseLeft.LeftChild = m_BaseLeft.RightChild =
m_BaseRight.LeftChild = m_BaseLeft.LeftChild = NULL;

	// Attach the two m_Base triangles together
	m_BaseLeft.BaseNeighbor = &m_BaseRight;
	m_BaseRight.BaseNeighbor = &m_BaseLeft;

	// Clear the other relationships.
	m_BaseLeft.RightNeighbor = m_BaseLeft.LeftNeighbor =
m_BaseRight.RightNeighbor = m_BaseRight.LeftNeighbor = NULL;
	TriangleIndex = 0;
}

// ---------------------------------------------------------------------
-
// Compute the variance tree for each of the Binary Triangles in this
patch.
//
template <class T>
void Patch<T>:: ComputeVariance()
{
	// Compute variance on each of the base triangles...

	m_CurrentVariance = m_VarianceLeft;
	RecursComputeVariance(  0,          width-1,
height_data[(width) * (width-1) ],
                            width-1,      0,
height_data[width-1],
                            0,          0,          height_data[0],
                            1);
```

```
        m_CurrentVariance = m_VarianceRight;
        RecursComputeVariance(  width-1,      0,          height_data[
width-1],
                                0,          width-1,      height_data[
(width) * (width-1)  ],
                                width-1,      width-1,
height_data[(width * width)-1],
                                1);

        // Clear the dirty flag for this patch
        m_VarianceDirty = 0;
}

// ------------------------------------------------------------------------
-
// ------------------------------------------------------------------------
-
// Set patch's visibility flag.
//
template <class T>
void Patch<T>::SetVisibility( GLdouble *proj_matrix, GLdouble
*model_matrix, GLint *viewport )
{
        // Get patch's center point
      if ((IsSeen(m_WorldX, m_WorldY, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX+width, m_WorldY, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX, m_WorldY+width, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX+width, m_WorldY+width, proj_matrix,
model_matrix, viewport) == 1)) {
            stuff->m_isVisible = 1;
      } else {
            stuff->m_isVisible = 0;
      }
}

// ------------------------------------------------------------------------
-
// Create an approximate mesh.
//
template <class T>
void Patch<T>::Tessellate()
{
        // Split each of the base triangles
        m_CurrentVariance = m_VarianceLeft;
        RecursTessellate(&m_BaseLeft,
                         m_WorldX,        m_WorldY+width-1,
                         m_WorldX+width-1,  m_WorldY,
                         m_WorldX,        m_WorldY,
                         1 );
```

```
        m_CurrentVariance = m_VarianceRight;
        RecursTessellate(&m_BaseRight,
                         m_WorldX+width-1,m_WorldY,
                         m_WorldX,      m_WorldY+width-1,
                         m_WorldX+width-1,m_WorldY+width-1,
                         1 );
}

// ------------------------------------------------------------------------
-
// Render the mesh.
//
template <class T>
void Patch<T>::BuildRender()
{
}

template <class T>
void Patch<T>::Render()
{
      long index;
      TriangleInfo *t;
        glPushMatrix();
      if ((flg_NewTexture==1) || (m_WorldY != stuff->y_pos) ||
(m_WorldX != stuff->x_pos)) {
            BindNewTexture();
            stuff->y_pos = m_WorldY;
            stuff->x_pos = m_WorldX;
      }
      glColor4f(1,1,1,1);
      glBindTexture(GL_TEXTURE_2D, 0);
        // Translate the patch to the proper world coordinates
        glTranslatef( (GLfloat)m_WorldX, 0, (GLfloat)m_WorldY );
        RecursRender (  &m_BaseLeft,
                0,     width-1,
                width-1, 0,
                0,     0);

        RecursRender(   &m_BaseRight,
                width-1, 0,
                0,     width-1,
                width-1, width-1);


        // Restore the matrix
        glPopMatrix();
}

template <class T>
void Patch<T>::RenderTexPanel()
{
        if ((m_WorldY != stuff->y_pos) || (m_WorldX != stuff->x_pos)) {
```

```
                BindNewTexture();
                stuff->y_pos = m_WorldY;
                stuff->x_pos = m_WorldX;
          }
        glBindTexture(GL_TEXTURE_2D, stuff->tex_id);
        glBegin(GL_QUADS);
        glTexCoord2f(0,1);
        glVertex2f(0,0);

        glTexCoord2f(1,1);
        glVertex2f(1,0);

        glTexCoord2f(1,0);
        glVertex2f(1,1);

        glTexCoord2f(0,0);


        glVertex2f(0,1);
        glEnd();
}
```

APPENDIX B

SOURCE LISTINGS FOR ALGORITHM B

Only changed files are listed.

1. Patch.cpp

```
#include <iostream>
#include <GL/glu.h>
#include <fstream>
#include <string>
#include <math.h>
#include "Utility.h"
using namespace std;

#include "Patch.h"

// Added 1-9-2k1 REH
template <class T>
Patch<T>::Patch(void)
{
	// Simple Constructor to initialize a few vars
	flg_NewTexture = 0;
	pTexture = NULL;
	height_data = NULL;
	colormap_data = NULL;
}

template <class T>
void Patch<T>::ClearFlags(void)
{
	// This function is called after the Draw routines to reset any
flags
	// Namely, the "There is a new texture" flag (flg_NewTexture)
	// Called per-frame, not per-context
	flg_NewTexture = 0;
}

template <class T>
void Patch<T>::GenerateTexture(void)
{
	// Generate the texture for this tile,
	// Memory is allocated in Init.
	// This routine is called from the Patch's Init, whenever new
data is loaded
	// The texture size will match the patch's size
	//   (The patch size is already ^2's, so this should work nicely.
	//    Let the video system handle color smoothing and such)
	// NOTE: At a later date, add in extra pixels (use 2x2 or 4x4
squares)
	//       and support software based lighting calculations.)
	T *d;
	unsigned char *tex;
	RGBA color;
```

```
      d = &colormap_data[width+1]; // skip the top row & left column
      tex = pTexture;
      for(int x=0; x<(width-1)*(width-1); x++) {
            info->cmap.GetColorRGBA((double)(*d), &color);
            *(tex+0) = color[0];
            *(tex+1) = color[1];
            *(tex+2) = color[2];
            *(tex+3) = color[3];
            tex+=4;
            d++;
            if ((d-colormap_data) % width == 0) d++; // continue to
skip the left column of data
      }
      flg_NewTexture = 1;
}

template <class T>
void Patch<T>::BindNewTexture(void)
{
      // Bind this texture, deleting/unbinding old texture if necessary
      // This routine is called once for each GL context
      // Delete any textures if there are any
      if (glIsTexture(stuff->tex_id) == GL_TRUE) {
            glDeleteTextures(1, &(stuff->tex_id));
      }

      // Now bind the new texture
        glEnable(GL_TEXTURE_2D);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

                    // Generate OpenGL texture IDs.
        glGenTextures(1, &(stuff->tex_id));

        glBindTexture(GL_TEXTURE_2D, stuff->tex_id);

        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, // RGBA textures.
             width-1, width-1, 0, GL_RGBA,
             GL_UNSIGNED_BYTE,
             pTexture);

}
// End of 1-9-2k1 changes

template <class T>
void Patch<T>::Split(TriTreeNode *tri)
```

```
{
      //We are already split, so there is no need to split again
      if (tri->LeftChild)
            return;
      //If this triangle is not in a proper diamond,
      //    force split the base neighbor.
      // This means, if this triangle's base is not shared as the base
of
      // another triangle (This triangle leans on the side of another,
      // not on the bottom), then if we split this one, we must split
all
      // the way back to a base.
      if (tri->BaseNeighbor && (tri->BaseNeighbor->BaseNeighbor !=
tri))
            Split(tri->BaseNeighbor);

      // Setup the two childrenfor the split
      tri->LeftChild = info->land->AllocateTri();
      tri->RightChild = info->land->AllocateTri();

      // If there was no available nodes for the children, then
      // exit gracefully.
      if (!tri->LeftChild)
            return;
      if (!tri->RightChild)
            return;
      // Point the new child to the current base & neighbors
      tri->LeftChild->BaseNeighbor  = tri->LeftNeighbor;
        tri->LeftChild->LeftNeighbor  = tri->RightChild;

        tri->RightChild->BaseNeighbor  = tri->RightNeighbor;
        tri->RightChild->RightNeighbor = tri->LeftChild;

      // Point the left neighbor directly to the newly created children
      if (tri->LeftNeighbor != NULL) {
            // One of the left's neighbor's neighbor-pointers should
            // point to me.. Find out which it is, and point it at
            // the new child
            if (tri->LeftNeighbor->BaseNeighbor == tri)
                        tri->LeftNeighbor->BaseNeighbor = tri-
>LeftChild;
                else if (tri->LeftNeighbor->LeftNeighbor == tri)
                        tri->LeftNeighbor->LeftNeighbor = tri-
>LeftChild;
                else if (tri->LeftNeighbor->RightNeighbor == tri)
                        tri->LeftNeighbor->RightNeighbor = tri-
>LeftChild;
                else
                        ;// Illegal Left Neighbor!
        }
        // Link our Right Neighbor to the new children
        if (tri->RightNeighbor != NULL)
        {
```

```
                if (tri->RightNeighbor->BaseNeighbor == tri)
                        tri->RightNeighbor->BaseNeighbor = tri-
>RightChild;
                else if (tri->RightNeighbor->RightNeighbor == tri)
                        tri->RightNeighbor->RightNeighbor = tri-
>RightChild;
                else if (tri->RightNeighbor->LeftNeighbor == tri)
                        tri->RightNeighbor->LeftNeighbor = tri-
>RightChild;
                else
                        ;// Illegal Right Neighbor!
        }

        // Link our Base Neighbor to the new children
        if (tri->BaseNeighbor != NULL)
        {
                if ( tri->BaseNeighbor->LeftChild )
                {
                        tri->BaseNeighbor->LeftChild->RightNeighbor =
tri->RightChild;
                        tri->BaseNeighbor->RightChild->LeftNeighbor =
tri->LeftChild;
                        tri->LeftChild->RightNeighbor = tri-
>BaseNeighbor->RightChild;
                        tri->RightChild->LeftNeighbor = tri-
>BaseNeighbor->LeftChild;
                }
                else
                        Split( tri->BaseNeighbor);  // Base Neighbor
(in a diamond with us) was not split yet, so do that now.
        }
        else
        {
                // An edge triangle, trivial case.
                tri->LeftChild->RightNeighbor = NULL;
                tri->RightChild->LeftNeighbor = NULL;
        }
}

// ----------------------------------------------------------------------
-
// Tessellate a Patch.
// Will continue to split until the variance metric is met.
//
template <class T>
void Patch<T>::RecursTessellate( TriTreeNode *tri,
                                                                int leftX,
int leftY,
                                                                int rightX,
int rightY,
                                                                int apexX,
int apexY,
                                                                int node )
```

```
{
        float TriVariance, d_num;
        int centerX = (leftX + rightX)>>1; // Compute X coordinate of
center of Hypotenuse
        int centerY = (leftY + rightY)>>1; // Compute Y coord...

        if ( node < (1<<info->variance_depth) )
        {
                // Extremely slow distance metric (sqrt is used).
                // Replace this with a faster one!
                float distance = 1.0f + ( SQR((float)centerX - info-
>gViewPosition[0]) +
                                                SQR((float)centerY -
info->gViewPosition[2]) );
            if (distance>=VAR_MAX)
                  d_num = 1.0/distance; //info-
>TriVarianceList[(int)VAR_MAX-1];
            else
                  d_num = info->TriVarianceList[(int)distance];

                // Egads!  A division too?  What's this world coming
to!
                // This should also be replaced with a faster
operation.
                TriVariance = ((float)m_CurrentVariance[node] * info-
>map_size * 2) * d_num;;
                  //* d_num; // Take both distance and variance into
consideration
        }

        if ( (node >= (1<<info->variance_depth)) || (TriVariance >
info->gFrameVariance))
        {
                Split(tri); // Split this triangle.

                if (tri->LeftChild &&  // If this triangle was split,
try to split it's children as well.
                          ((abs(leftX - rightX) >= 3) || (abs(leftY -
rightY) >= 3)))   {
                        RecursTessellate( tri->LeftChild,   apexX,
apexY, leftX, leftY, centerX, centerY,    node<<1  );
                        RecursTessellate( tri->RightChild, rightX,
rightY, apexX, apexY, centerX, centerY, 1+(node<<1) );
                }
        }
}

// ---------------------------------------------------------------------
-
// Render the tree.  Simple no-fan method.
//
template <class T>
```

```
void Patch<T>::RecursRender( TriTreeNode *tri, int leftX, int leftY,
int rightX, int rightY, int apexX, int apexY )
{
      TriangleInfo *t;
        if ( tri->LeftChild )                                    // All
non-leaf nodes have both children, so just check for one
        {
                int centerX = (leftX + rightX)>>1;      // Compute X
coordinate of center of Hypotenuse
                int centerY = (leftY + rightY)>>1;      // Compute Y
coord...

                RecursRender( tri->LeftChild,  apexX,   apexY, leftX,
leftY, centerX, centerY );
                RecursRender( tri->RightChild, rightX, rightY, apexX,
apexY, centerX, centerY );
        }
        else
// A leaf node!  Output a triangle to be rendered.
        {
                // Actual number of rendered triangles...
                RGBA color;
                info->gNumTrisRendered++;
                GLfloat leftZ  = height_data[((leftY) *width)+leftX ];
                GLfloat rightZ = height_data[((rightY)*width)+rightX];
                GLfloat apexZ  = height_data[((apexY) *width)+apexX ];

                // Perform polygon coloring based on a height sample
                glBegin(GL_TRIANGLES);
                        info->cmap.GetColorRGBA(leftZ, &color);
                        glColor3ub( color[0], color[1], color[2] );
                        glVertex3f(leftX, leftZ, leftY);

                        info->cmap.GetColorRGBA(rightZ, &color);
                        glColor3ub( color[0], color[1], color[2] );
                        glVertex3f(rightX, rightZ, rightY);

                        info->cmap.GetColorRGBA(apexZ, &color);
                        glColor3ub( color[0], color[1], color[2] );
                        glVertex3f(apexX, apexZ, apexY);
                glEnd();

        }
}

// ----------------------------------------------------------------------
-
// Computes Variance over the entire tree.  Does not examine node
relationships.
// NOTE: left, right & APex are in LOCAL coordinates (locations within
the patch)
template <class T>
float Patch<T>::RecursComputeVariance( int leftX,  int leftY,  T leftZ,
```

```
                                        int rightX, int rightY, T rightZ,
                                        int apexX,  int apexY,  T apexZ,
                                        int node)
{
        //        /|\
        //      /  |  \
        //    /    |    \
        //  /      |      \
        //  ~~~~~~~*~~~~~~~  <-- Compute the X and Y coordinates of '*'
        //
        int centerX = (leftX + rightX) >>1;             // Compute X
coordinate of center of Hypotenuse
        int centerY = (leftY + rightY) >>1;             // Compute Y
coord...
        float myVariance;
      float tempvar;

        // Get the height value at the middle of the Hypotenuse
        T centerZ  = height_data[((centerY) * width) + centerX];

        // Variance of this triangle is the actual height at it's
hypotenuse midpoint minus the interpolated height.
        // Use values passed on the stack instead of re-accessing the
Height Field.
        myVariance = abs(centerZ - ((int)(leftZ + rightZ)>>1));
        // Since we're after speed and not perfect representations,
        //    only calculate variance down to an 8x8 block
        if ( (abs(leftX - rightX) >= 8) ||
                  (abs(leftY - rightY) >= 8) )
        {
                // Final Variance for this node is the max of it's own
variance and that of it's children.
            tempvar = RecursComputeVariance( apexX,   apexY,  apexZ,
                                        leftX, leftY, leftZ,
                                        centerX, centerY, centerZ,
                                        node<<1 );
                myVariance = MAX( myVariance,tempvar);
      //Left Child Node

            tempvar = RecursComputeVariance( rightX, rightY, rightZ,
                                        apexX, apexY, apexZ,
                                        centerX, centerY, centerZ,
                                        1+(node<<1));
                myVariance = MAX( myVariance,tempvar);
      //Right Child Node
        }
        // Store the final variance for this node.  Note Variance is
never zero.
        if (node < (1<<info->variance_depth)) {
                m_CurrentVariance[node] = info->MinVariance +
myVariance;
      }
        return myVariance;
```

```
}


// ------------------------------------------------------------------------
----------------------------
//      PATCH CLASS
// ------------------------------------------------------------------------
----------------------------

// ------------------------------------------------------------------------
-
// Initialize a patch.
//
template <class T>
void Patch<T>::Init(int worldX, int worldY, int size, MetaData<T>
*ls_info )
{
        // Clear all the relationships
        m_BaseLeft.RightNeighbor = m_BaseLeft.LeftNeighbor =
m_BaseRight.RightNeighbor = m_BaseRight.LeftNeighbor =
            m_BaseLeft.LeftChild = m_BaseLeft.RightChild =
m_BaseRight.LeftChild = m_BaseLeft.LeftChild = NULL;

        // Attach the two m_Base triangles together
        m_BaseLeft.BaseNeighbor = &m_BaseRight;
        m_BaseRight.BaseNeighbor = &m_BaseLeft;

      info = ls_info;
        // Store Patch offsets for the world and heightmap.
        m_WorldX = worldX;
        m_WorldY = worldY;
      if (width != size) {
              width = size+1;
              // Store pointer to first byte of the height data for
this patch.
            if (height_data) {
                  delete height_data;
            }
            height_data=new T[(size+1) * (size+1)];
            if (colormap_data) {
                  delete colormap_data;
            }
            colormap_data=new T[(size+1) * (size+1)];
      }

      Triangle = new TriangleInfo[(size+1) * (size+1)*2];
      TriangleIndex = 0;

      m_VarianceLeft = new float[1<<info->variance_depth];
        m_VarianceRight = new float[1<<info->variance_depth];

        // Initialize flags
        m_VarianceDirty = 1;
```

```
	// Allocate Texture
	if (pTexture!=NULL) {
		delete pTexture;
	}
	pTexture = new unsigned char[size * size *4]; // 4 bytes for RGBA
	if (pTexture == NULL) {
		cout << "ERROR: Unable to allocate RAM" << endl;
	}
}

template <class T>
float Patch<T>::getData(int x, int y)
{
	return height_data[(y * width) + x];
}

// ---------------------------------------------------------------------
-
// Reset the patch.
//
template <class T>
void Patch<T>::Reset()
{
	// Reset the important relationships
	m_BaseLeft.LeftChild = m_BaseLeft.RightChild =
m_BaseRight.LeftChild = m_BaseLeft.LeftChild = NULL;

	// Attach the two m_Base triangles together
	m_BaseLeft.BaseNeighbor = &m_BaseRight;
	m_BaseRight.BaseNeighbor = &m_BaseLeft;

	// Clear the other relationships.
	m_BaseLeft.RightNeighbor = m_BaseLeft.LeftNeighbor =
m_BaseRight.RightNeighbor = m_BaseRight.LeftNeighbor = NULL;
	TriangleIndex = 0;
}

// ---------------------------------------------------------------------
-
// Compute the variance tree for each of the Binary Triangles in this
patch.
//
template <class T>
void Patch<T>:: ComputeVariance()
{
	// Compute variance on each of the base triangles...

	m_CurrentVariance = m_VarianceLeft;
	RecursComputeVariance(  0,          width-1,
height_data[(width) * (width-1) ],
				width-1,      0,
height_data[width-1],
```

```
                                0,          0,          height_data[0],
                                1);

        m_CurrentVariance = m_VarianceRight;
        RecursComputeVariance(  width-1,      0,          height_data[
width-1],
                                0,          width-1,      height_data[
(width) * (width-1)  ],
                                width-1,      width-1,
height_data[(width * width)-1],
                                1);

        // Clear the dirty flag for this patch
        m_VarianceDirty = 0;
}

// ------------------------------------------------------------------------
-
// ------------------------------------------------------------------------
-
// Set patch's visibility flag.
//
template <class T>
void Patch<T>::SetVisibility( GLdouble *proj_matrix, GLdouble
*model_matrix, GLint *viewport )
{
        // Get patch's center point
      if ((IsSeen(m_WorldX, m_WorldY, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX+width, m_WorldY, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX, m_WorldY+width, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX+width, m_WorldY+width, proj_matrix,
model_matrix, viewport) == 1)) {
            stuff->m_isVisible = 1;
      } else {
            stuff->m_isVisible = 0;
      }
}

// ------------------------------------------------------------------------
-
// Create an approximate mesh.
//
template <class T>
void Patch<T>::Tessellate()
{
        // Split each of the base triangles
        m_CurrentVariance = m_VarianceLeft;
        RecursTessellate(&m_BaseLeft,
                         m_WorldX,        m_WorldY+width-1,
                         m_WorldX+width-1,  m_WorldY,
```

```
                        m_WorldX,        m_WorldY,
                        1 );

        m_CurrentVariance = m_VarianceRight;
        RecursTessellate(&m_BaseRight,
                        m_WorldX+width-1,m_WorldY,
                        m_WorldX,      m_WorldY+width-1,
                        m_WorldX+width-1,m_WorldY+width-1,
                        1 );
}

// ------------------------------------------------------------------------
-
// Render the mesh.
//
template <class T>
void Patch<T>::BuildRender()
{
}

template <class T>
void Patch<T>::Render()
{
      long index;
      TriangleInfo *t;
        glPushMatrix();
      if ((flg_NewTexture==1) || (m_WorldY != stuff->y_pos) ||
(m_WorldX != stuff->x_pos)) {
            BindNewTexture();
            stuff->y_pos = m_WorldY;
            stuff->x_pos = m_WorldX;
      }
      glColor4f(1,1,1,1);
      glBindTexture(GL_TEXTURE_2D, 0);
        // Translate the patch to the proper world coordinates
        glTranslatef( (GLfloat)m_WorldX, 0, (GLfloat)m_WorldY );
        RecursRender (  &m_BaseLeft,
                0,     width-1,
                width-1, 0,
                0,     0);

        RecursRender(   &m_BaseRight,
                width-1, 0,
                0,     width-1,
                width-1, width-1);


        // Restore the matrix
        glPopMatrix();
}

template <class T>
void Patch<T>::RenderTexPanel()
```

```
{
        if ((m_WorldY != stuff->y_pos) || (m_WorldX != stuff->x_pos)) {
                BindNewTexture();
                stuff->y_pos = m_WorldY;
                stuff->x_pos = m_WorldX;
        }
      glBindTexture(GL_TEXTURE_2D, stuff->tex_id);
      glBegin(GL_QUADS);
      glTexCoord2f(0,1);
      glVertex2f(0,0);

      glTexCoord2f(1,1);
      glVertex2f(1,0);

      glTexCoord2f(1,0);
      glVertex2f(1,1);

      glTexCoord2f(0,0);


      glVertex2f(0,1);
      glEnd();
}
```

2. Landscape.cpp

```
#include <iostream>
#include <GL/glu.h>
#include <fstream>
#include <string>
#include <math.h>

using namespace std;

#include "Landscape.h"

VCRoamSurfaceFloat::VCRoamSurfaceFloat(void)
{
      myplot = NULL;
}

VCRoamSurfaceFloat::~VCRoamSurfaceFloat(void)
{
      if (myplot)
            delete (myplot);
}

void VCRoamSurfaceFloat::Draw(void)
{
        glPushMatrix();
        myplot->Render();
```

```
        glPopMatrix();
      glBindTexture(GL_TEXTURE_2D, 0);
}

void VCRoamSurfaceFloat::PerFrameCalculations(void)
{
      vjMatrix dummy1;
      vjVec3 location;

        dummy1 = user->getPosition();
        location=dummy1.getTrans();
        myplot->info.gViewPosition[0]=location[0];
        myplot->info.gViewPosition[1]=location[1];
        myplot->info.gViewPosition[2]=location[2];

        //myplot->info.gClipAngle = -gAnimateAngle;

        myplot->Reset();
        myplot->Tessellate();
}

void VCRoamSurfaceFloat::PostRender(void)
{
      myplot->PostRender();
}

int VCRoamSurfaceFloat::Status(void)
{
      return 0; // This component never deletes itself
}

void VCRoamSurfaceFloat::InitGL(void)
{
      myplot->InitGL();

}

void VCRoamSurfaceFloat::Init(DataManager *DM, UserData *userdata,
string filename, string header)
{
   CfgHeader = header;
   CfgFileName = filename;
   user = userdata;
   myplot = new ROAMLandScape<float>(DM , CfgFileName, header);
   myplot->info.gViewPosition[0] = ReadIntFromFile(CfgFileName,
"SYSTEM", "startx", 0);
   myplot->info.gViewPosition[1] = 0;
   myplot->info.gViewPosition[2] = ReadIntFromFile(CfgFileName,
"SYSTEM", "starty", 0);
   myplot->info.cmap.LoadColormap(ReadStringFromFile(CfgFileName,
header, "colormap", ""));
   myplot->info.cmap.SetLand(0, 1, 0, 1);
```

```
   myplot->info.user = user;
   myplot->Init();

}


template <class T>
ROAMLandScape<T>::ROAMLandScape(DataManager *DM, string filename,
string Header)
{
       LoadConfigFile(filename, Header);

      info.m_TriPool = new TriTreeNode[info.max_nodes];

      cout << "ROAM:constructor --> Building new Tile Manager" << endl;
      TileManager = DM->GetDataBase(ReadStringFromFile(filename,
Header, "data", ""));
      TileManager->setLayer(ReadIntFromFile(filename, Header,
"data_layer", 1));
      info.layer = TileManager->getLayer();

      info.land = this;
      if(ReadStringFromFile(filename, Header, "color", "") == "shared")
{
            cout << "--> Using Shared Colormap/Heightmap data" << endl;
            color_data = TileManager;
      } else {
            cout << "--> Not Using Shared Colormap/Heightmap data" <<
endl;
            color_data = DM->GetDataBase(ReadStringFromFile(filename,
Header, "color", ""));
            color_data->setLayer(ReadIntFromFile(filename, Header,
"color_layer", 1));
      }


      for(int index=0;index<VAR_MAX;index++)
            info.TriVarianceList[index] = 1.0 / (float)(index);
      m_Patches = NULL;
}

template <class T>
void ROAMLandScape<T>::LoadConfigFile(string filename, string Header)
{
      info._patchsize = ReadIntFromFile(filename, Header, "patchsize",
0);
      info._viewsquare = ReadIntFromFile(filename, Header,
"viewsquare", 0);
      info.variance_depth = ReadIntFromFile(filename, Header,
"variance_depth", 0);
      info.max_nodes = ReadIntFromFile(filename, Header, "max_nodes",
0);
```

```
	info.gDesiredTris = ReadIntFromFile(filename, Header,
"gDesiredTris", 0);
	info.map_size = ReadIntFromFile(filename, Header, "map_size", 0);
	info.gDrawMode = ReadIntFromFile(filename, Header, "render", 0);
	info.gFrameVariance = ReadIntFromFile(filename, Header,
"FrameVariance", 0);
	info.AutoLOD = ReadIntFromFile(filename, Header, "AutoLOD", 0);
	info.MinVariance = ReadFloatFromFile(filename, Header,
"MinVariance", 0.0);
}

template <class T>
void ROAMLandScape<T>::InitGL(void)
{
	// This used to create the land texture
	// Obviously now removed
}


// ------------------------------------------------------------------
------------------------------
//	LANDSCAPE CLASS
// ------------------------------------------------------------------
------------------------------

// ------------------------------------------------------------------
-
// Allocate a TriTreeNode from the pool.
//
template <class T>
TriTreeNode *ROAMLandScape<T>::AllocateTri()
{
	TriTreeNode *pTri;

	// IF we've run out of TriTreeNodes, just return NULL (this is
handled gracefully)
	if ( info.m_NextTriNode >= info.max_nodes )
		return NULL;

	pTri = &(info.m_TriPool[info.m_NextTriNode++]);
	pTri->LeftChild = pTri->RightChild = NULL;

	return pTri;
}

// ------------------------------------------------------------------
-
// Initialize all patches
//
template<class T>
void ROAMLandScape<T>::Init(void)
{
	Patch<T> *patch;
```

```
      Patch_Info<T> *ppatch;
        int X, Y;
      int diff;
      int sx_offset, sy_offset;
      int halfdist;
      TileManager->setLayer(info.layer);

      // Make sure we load the data centered around the user, so go
      // half of the viewable distance to the left,right,front, and
back of the user
      halfdist = ((info._viewsquare-1) * (info._patchsize)) >>1 ;

      diff = ((int)info.gViewPosition[0] % (info._patchsize));
      sx_offset = (info.gViewPosition[0]-diff) - halfdist;

      diff = ((int)info.gViewPosition[2] % (info._patchsize));
      sy_offset = (info.gViewPosition[2]-diff) - halfdist;

      // Now this is fucked up, I can't explain this
      //  But it doesn't work without it
//    if (sx_offset < 0) sx_offset--;
//    if (sy_offset < 0) sy_offset--;

        // Initialize all terrain patches
        // Store the Height Field array
      if (m_Patches == NULL) {
            // The patches haven't been allocated yet.. This means we
need
            // to allocate memory & load all the patches.
            m_Patches = new
Patch_Info<T>[info._viewsquare*info._viewsquare];
            PatchPool = new
Patch<T>[info._viewsquare*info._viewsquare];
            patch = &(PatchPool[0]);
            ppatch = &(m_Patches[0]);
            for ( Y=0; Y < info._viewsquare; Y++)
                    for ( X=0; X < info._viewsquare; X++ )
                    {
                  ppatch->the_patch = patch;
                  ppatch->valid = TRUE;
                  ppatch->updated = TRUE;
                        patch->Init(sx_offset + X*(info._patchsize),
                              sy_offset+ Y*(info._patchsize),
                              info._patchsize,
                              &info);
                  // Load data now.. If no data is found, then this
patch is
                  // Invalid (outside data-able area)
                  if(TileManager->loadArea(sx_offset +
X*(info._patchsize),
                                    sy_offset + Y*(info._patchsize),
                                          info._patchsize+1,
                                    info._patchsize+1,
```

```
                                        patch->height_data) == -1 )
                        ppatch->valid = FALSE;
                  else  {
                        color_data->loadArea(sx_offset +
X*(info._patchsize),
                                             sy_offset +
Y*(info._patchsize),
                                             info._patchsize+1,
                                             info._patchsize+1,
                                             patch->colormap_data);
                              patch->ComputeVariance();
                  }
                  patch++;
                  ppatch++;
                    }
      } else {
            // The patches have been previously allocated, so they
contain data.
            // Find the valid data to preserve, and load the rest.
            // Algorithm:
            // First search all the data tiles.  If one is "Valid",
swap it to it's correct
            //    location and mark it valid.
            // Then pass through all "invalid" tiles and load them.
            int left, right, top, bottom;
            int patch_x, patch_y;
            int p2x, p2y;
            Patch<T> *patch_temp;
            Patch_Info<T> *dpatch; // Destination Patch

            left = sx_offset;
            top = sy_offset;
            right = left + (info._viewsquare-1)*(info._patchsize);
            bottom = top + (info._viewsquare-1)*(info._patchsize);

            // Start by marking them all invalid for later checks
            ppatch = &(m_Patches[0]);
            for ( X=0; X < info._viewsquare*info._viewsquare; X++,
ppatch++)  {
                  ppatch->updated = FALSE;
//                ppatch->valid = FALSE;
            }

            // Now find valid's/invalid's
            cout << "Swapping" << flush;
            ppatch = &(m_Patches[0]);
            for ( X=0; X < info._viewsquare*info._viewsquare; X++,
ppatch++)
                // First see if this patch is valid.
                if (ppatch->valid == TRUE) { // don't bother checking
if we know it's an empty/bad patch
                  (ppatch->the_patch)->getLocation(&patch_x, &patch_y);
```

```
                  // if this patch starts within the viewable area then
it must be valid.
                  // Because all patches are required to load on
(viewsquare-1 *n) boundaries.
                  if ((patch_x>=left) && (patch_x <= right) &&
(patch_y>=top) && (patch_y<=bottom)) {
                        // Valid tile
                        // Find where this tile *should* be.
                        dpatch = &m_Patches[ int((patch_y - sy_offset)
/ (info._patchsize)) * info._viewsquare +
                                          int((patch_x - sx_offset) /
(info._patchsize))];
                        if (dpatch != ppatch) { // Make sure it belongs
somewhere other than where it is.
                              // see (if due to a glitch I can't seem
to fix any other way) the
                              // source & destination patches are not
the same, but both store the
                              // same data, then mark this one invalid
& kill it.
                              dpatch->the_patch->getLocation(&p2x,
&p2y);
                              if ((p2x==patch_x) && (p2y==patch_y)) {
                                    ppatch->valid = FALSE;
                                    X =
info._viewsquare*info._viewsquare;
                              } else {
                                    // Now swap the two patch
pointers..
                                    patch_temp = dpatch->the_patch;
                                    dpatch->the_patch = ppatch-
>the_patch; // This patch is valid now
                                    ppatch->the_patch = patch_temp;
                                    // The new patch is valid, we know
it.
                                    // The one we swapped it with may
or may not be, so just copy the status
                                    ppatch->valid = dpatch->valid;
                                    dpatch->valid = TRUE;
                                    // Now reset it.. We may have
swapped so that a tile will
                                    // be skipped, so restart at the
beginning.
                                    // Back up 1 because they will be
incremented
                                    // NOTE: Look at improving this
brute-force method.
                                    //    Possible recursion? That may
be worse..
                                    X = -1;
                                    ppatch = &(m_Patches[0]);
                                    ppatch--;
                                    cout << "." << flush ;
```

```
                              }
                        } // end of if dpatch != ppatch
                  } else {
                        // Invalid tile
                        ppatch->valid = FALSE;
                  } // end of if within left/top/bottom/right
                } // end of if valid

            // Now we know which ones are valid/invalid..
            // So reload the invalid ones.
                ppatch = &(m_Patches[0]);
                for ( Y=0; Y < info._viewsquare; Y++)
                    for ( X=0; X < info._viewsquare; X++, ppatch++ )
                  if(ppatch->valid == FALSE) {
                        // This patch isn't valid.. so reload.
                        ppatch->updated = TRUE;
                              (ppatch->the_patch)->Init(sx_offset +
X*(info._patchsize),
                                          sy_offset+ Y*(info._patchsize),
                                          info._patchsize,
                                          &info);
                              if(TileManager->loadArea(sx_offset +
X*(info._patchsize),
                                                sy_offset +
Y*(info._patchsize),
                                                info._patchsize+1,
                                                info._patchsize+1,
                                                (ppatch->the_patch)-
>height_data) == -1) {
                              ppatch->valid = FALSE;
                        } else  {
                              color_data->loadArea(sx_offset +
X*(info._patchsize),
                                                sy_offset +
Y*(info._patchsize),
                                                info._patchsize+1,
                                                info._patchsize+1,
                                                (ppatch->the_patch)-
>colormap_data);
                                    (ppatch->the_patch)-
>ComputeVariance();
                              ppatch->valid = TRUE;
                        }
                  }

      }
      info.height_dataMin = TileManager->getMin();
      info.height_dataMax = TileManager->getMax();
      if (info.height_dataMax < info.height_dataMin) {
            T tmp = info.height_dataMin;
            info.height_dataMin = info.height_dataMax;
            info.height_dataMax = tmp;
      }
```

```
      info.color_dataMin = TileManager->getMin();
      info.color_dataMax = TileManager->getMax();
      if (info.color_dataMax < info.color_dataMin) {
            T tmp = info.color_dataMin;
            info.color_dataMin = info.color_dataMax;
            info.color_dataMax = tmp;
      }

      info.cmap.SetMaxMin(info.color_dataMax, info.color_dataMin);
      ppatch = &(m_Patches[0]);
        for ( Y=0; Y < info._viewsquare*info._viewsquare; Y++){
            if((ppatch->valid == TRUE) && (ppatch->updated == TRUE))
                  (ppatch->the_patch)->GenerateTexture();
            ppatch++;
      }
      info.height_dataSpread = info.height_dataMax -
info.height_dataMin;
      info.height_dataSpread = info.height_dataMax -
info.height_dataMin;
}

template <class T>
void ROAMLandScape<T>::Reset(void)
{
        //
        // Perform simple visibility culling on entire patches.
        //   - Define a triangle set back from the camera by one patch
size, following
        //     the angle of the frustum.
        //   - A patch is visible if any of it's 4 corners are within
the viewable area
      //          -- uses gluUnProject to figure out
        //   - This visibility test is only accurate if the camera
cannot look up or down significantly.
        //
        int X, Y, size;
      int startx, starty;
        Patch_Info<T> *patch;
      size = info._viewsquare;
      Patch_Info<T> *pLeft, *pRight, *pTop, *pBottom;

        // Set the next free triangle pointer back to the beginning
        SetNextTriNode(0);

      // See if we are outside the center patch
      patch = &m_Patches[(size * size) >> 1]; // set Patch to center
patch
                                    // NOTE: Only works when size is an
odd number
      if (patch->valid == TRUE) {
          (patch->the_patch)->getLocation(&startx, &starty);
          if ((info.gViewPosition[0] < startx) ||
(info.gViewPosition[0] > startx+info._patchsize+1) ||
```

```
            (info.gViewPosition[2] < starty) || (info.gViewPosition[2]
> starty+info._patchsize+1)) {
            // We are outside the center patch, so reload data to put
us inside the center patch
            vjDEBUG(vjDBG_ALL, 1) << "User at: " <<
info.gViewPosition[0] << "," <<
                  info.gViewPosition[2] << endl << vjDEBUG_FLUSH;
            vjDEBUG(vjDBG_ALL, 1) << "Center tile at: " << startx <<
"," << starty << endl << vjDEBUG_FLUSH;
            vjDEBUG(vjDBG_ALL, 1) << "Not Centered. Reloading\n" <<
endl << vjDEBUG_FLUSH;
            // Insert code to swap stuff out HERE
            Init();
          }
      }
        // Reset rendered triangle count.
        info.gNumTrisRendered = 0;
      patch=&(m_Patches[0]);
        // Go through the patches performing resets, compute variances,
and linking.
        for ( Y=0; Y < info._viewsquare; Y++ )
                for ( X=0; X < info._viewsquare; X++)
                {
                        // Reset the patch
                if (patch->valid == TRUE) { // If patch is invalid,
then don't bother rebuilding it
                                    // Prevents "edge of the world" bug
                                    // Edge of world is now darkness..
Just like the europeans thought :)
                        (patch->the_patch)->Reset();

                        // Check to see if this patch has been deformed
since last frame.
                        // If so, recompute the varience tree for it.
                        if ( (patch->the_patch)->isDirty() )
                                (patch->the_patch)->ComputeVariance();
                  pTop = &(m_Patches[(Y-1)*size + X]);
                  pBottom = &(m_Patches[(Y+1)*size +X]);
                  pLeft = &(m_Patches[(Y*size) + X-1]);
                  pRight = &(m_Patches[(Y*size) + X+1]);

                        // Link all the patches together.
                        if ((X > 0) && (pLeft->valid == TRUE) )
                                (patch->the_patch)->GetBaseLeft()-
>LeftNeighbor = pLeft->the_patch->GetBaseRight();
                        else
                                (patch->the_patch)->GetBaseLeft()-
>LeftNeighbor = NULL;              // Link to bordering Landscape
here..

                        if (( X < (size-1)) && (pRight->valid == TRUE)
)
```

```
                                (patch->the_patch)->GetBaseRight()-
>LeftNeighbor = pRight->the_patch->GetBaseLeft();
                        else
                                (patch->the_patch)->GetBaseRight()-
>LeftNeighbor = NULL;             // Link to bordering Landscape here..

                        if (( Y > 0) && (pTop->valid == TRUE) )
                                (patch->the_patch)->GetBaseLeft()-
>RightNeighbor = pTop->the_patch->GetBaseRight();
                        else
                                (patch->the_patch)->GetBaseLeft()-
>RightNeighbor = NULL;             // Link to bordering Landscape
here..

                        if (( Y < (size-1)) && (pBottom->valid == TRUE)
)
                                (patch->the_patch)->GetBaseRight()-
>RightNeighbor = pBottom->the_patch->GetBaseLeft();
                        else
                                (patch->the_patch)->GetBaseRight()-
>RightNeighbor = NULL;    // Link to bordering Landscape here..
                }
                patch++;
                }
}


// ------------------------------------------------------------------------
-
// Create an approximate mesh of the landscape.
//
template<class T>
void ROAMLandScape<T>::Tessellate()
{
        // Perform Tessellation
      int x, count;
        Patch_Info<T> *patch = &(m_Patches[0]);
      count = info._viewsquare * info._viewsquare;

        for(x=0;x<count; x++, patch++){
            if (patch->valid == TRUE)
                  (patch->the_patch)->Tessellate( );
      }
        for(patch = &(m_Patches[0]),x=0;x<count; x++, patch++){
            if (patch->valid == TRUE)
                  (patch->the_patch)->BuildRender();
        }
}

template <class T>
float ROAMLandScape<T>::GetAltitude(float x, float y)
{
        float corners[2][2], topApprox, botApprox;
```

```
        float ix, iy;
      ix = ftrunc(x);
      iy = ftrunc(y);
        corners[0][0] = getData(ix, iy);
        corners[0][1] = getData(ix+1, iy);
        corners[1][0] = getData(ix, iy+1);
        corners[1][1] = getData(ix+1, iy+1);

        topApprox = corners[0][0] +
                (x - ix)*(corners[0][1] - corners[0][0]);
        botApprox = corners[1][0] +
                (x - ix)*(corners[1][1] - corners[1][0]);
        return(topApprox +
                (y - iy)*(botApprox - topApprox));

}

// ---------------------------------------------------------------------
-
// Render each patch of the landscape & adjust the frame variance.
//
template <class T>
void ROAMLandScape<T>::Render()
{
        int x,y ;
        Patch_Info<T> *patch = &(m_Patches[0]);
      GLdouble projection[16], model[16];
      GLint viewport[4];

      glDisable(GL_LIGHTING);
      glGetDoublev(GL_PROJECTION_MATRIX,projection);
      glGetDoublev(GL_MODELVIEW_MATRIX, model);
      glGetIntegerv(GL_VIEWPORT,viewport);
      // Set visibility on patches (View Frustrum Culling)

for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare; x++,
patch++)
                if (patch->valid == TRUE)
                  (patch->the_patch)->SetVisibility(projection, model,
viewport);
      // Force nearby patches to be visible (reduce blicker);
      //  (for the unknowing.. blicker = flicker + blink )
      //  Blicker (c) Randall Hand 2001
      for(x=-1;x<=1; x++) {
            for(y=-1; y<=1; y++) {

      MakeVisible(info.gViewPosition[0]+(x*(info._patchsize)),
                        info.gViewPosition[2]+(y*(info._patchsize)));
            }
      }
      // Render Visible Patches
      if ((info.user)->Render_Wireframe) {
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

```
            glEnable(GL_CULL_FACE);
      } else {
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      }

for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare; x++,
patch++){
            if ((patch->valid == TRUE) && ((patch->the_patch)-
>isVisible()))   {
                  (patch->the_patch)->Render();
            }
        }

      glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      glDisable(GL_CULL_FACE);
      DrawMap();
}

template <class T>
void ROAMLandScape<T>::DrawMap(void)
{
      vjVec3 angles;
      int x;
        Patch_Info<T> *patch = &(m_Patches[0]);

        glDisable(GL_DEPTH_TEST);
      for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare
; x++, patch++){
            if(patch->valid == TRUE) {
                  glPushMatrix();
                  glLoadIdentity();
                  glTranslatef(0, 5, 0);
                  glScalef(0.5,0.5,0.5);
                  glTranslatef((x % info._viewsquare), 10-
int(x/info._viewsquare),-15);
                  (patch->the_patch)->RenderTexPanel();
                  glPopMatrix();
            }
      }

      glPushMatrix();
      glLoadIdentity();
      glTranslatef(0,5,0);
      glScalef(0.5,0.5,0.5);
      glTranslatef((float)info._viewsquare/2.0, 11.0 -
((float)info._viewsquare/2.0), -15);
      angles = (info.user)->GetUserOrientation();
      if (angles[0] == -180) {
            glRotatef(-(180-angles[1]), 0,0,1);
      } else {
            glRotatef(-angles[1],0,0,1);
      }
      glBindTexture(GL_TEXTURE_2D, 0);
```

```
      glColor4f(1,1,1,1);
      glBegin(GL_TRIANGLES);
         glVertex3f(0,0,0);
         glColor4f(1,1,1,0);
         glVertex3f(-2, 3, 0);
         glVertex3f(2, 3, 0);
      glEnd();
      glPopMatrix();


        glEnable(GL_DEPTH_TEST);

}



template <class T>
void ROAMLandScape<T>::PostRender(void)
{
        // Check to see if we got close to the desired number of
triangles.
        // Adjust the frame variance to a better value.
      int x;
      Patch<T> *patch;
      if (info.AutoLOD == 0)
            return;
        if ( GetNextTriNode() != info.gDesiredTris )
                info.gFrameVariance += ((float)GetNextTriNode() -
(float)info.gDesiredTris) / (float)info.gDesiredTris;

        // Bounds checking.
        if ( info.gFrameVariance < 0 )
                info.gFrameVariance = 0;


for(x=0,patch=&(PatchPool[0]);x<info._viewsquare*info._viewsquare; x++,
patch++){
              patch->ClearFlags();
      }
}

template <class T>
void ROAMLandScape<T>::SetDrawModeContext()
{
        switch (info.gDrawMode)
        {
        case DRAW_USE_TEXTURE:
                glDisable(GL_LIGHTING);
                glEnable(GL_TEXTURE_2D);
        //        glPolygonMode(GL_FRONT, GL_FILL);
                break;

        case DRAW_USE_LIGHTING:
```

```
                glEnable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_FILL);
                break;

        case DRAW_USE_FILL_ONLY:
                glDisable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_FILL);
                break;

        default:
        case DRAW_USE_WIREFRAME:
                glDisable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_LINE);
                break;
        }
}

template <class T>
float ROAMLandScape<T>::getData(int x, int y)
{
      int size;
      int startx, starty;
      int px, py;
        Patch_Info<T> *patch = &(m_Patches[0]);
      size = info._patchsize;

      (patch->the_patch)->getLocation(&startx, &starty);
      px = (x - startx) / size;
      py = (y - starty) / size;

      patch = &(m_Patches[(py * info._viewsquare) + px]);
      (patch->the_patch)->getLocation(&startx, &starty);

      return (patch->the_patch)->getData(x-startx, y-starty);
}

template <class T>
void ROAMLandScape<T>::MakeVisible(int x, int y)
{
      int size;
      int startx, starty;
      int px, py;
        Patch_Info<T> *patch = &(m_Patches[0]);
      size = info._patchsize;

      (patch->the_patch)->getLocation(&startx, &starty);
      px = (x - startx) / size;
      py = (y - starty) / size;

      patch = &(m_Patches[(py * info._viewsquare) + px]);
```

```
        (patch->the_patch)->ForceVisibility();
}
```

APPENDIX C

SOURCE LISTINGS FOR ALGORITHM C

Only modified files are listed here.

1. Patch.cpp

```
#include <iostream>
#include <GL/glu.h>
#include <fstream>
#include <string>
#include <math.h>
#include "Utility.h"
using namespace std;

#include "Patch.h"

// Added 1-9-2k1 REH
template <class T>
Patch<T>::Patch(void)
{
      // Simple Constructor to initialize a few vars
      flg_NewTexture = 0;
      pTexture = NULL;
      height_data = NULL;
      colormap_data = NULL;
}

template <class T>
void Patch<T>::ClearFlags(void)
{
      // This function is called after the Draw routines to reset any
flags
      // Namely, the "There is a new texture" flag (flg_NewTexture)
      // Called per-frame, not per-context
      flg_NewTexture = 0;
}

template <class T>
void Patch<T>::GenerateTexture(void)
{
      // Generate the texture for this tile,
      // Memory is allocated in Init.
      // This routine is called from the Patch's Init, whenever new
data is loaded
      // The texture size will match the patch's size
      //   (The patch size is already ^2's, so this should work nicely.
      //    Let the video system handle color smoothing and such)
      // NOTE: At a later date, add in extra pixels (use 2x2 or 4x4
squares)
      //       and support software based lighting calculations.)
      T *d;
      unsigned char *tex;
      RGBA color;
```

```
      d = &colormap_data[width+1]; // skip the top row & left column
      tex = pTexture;
      for(int x=0; x<(width-1)*(width-1); x++) {
            info->cmap.GetColorRGBA((double)(*d), &color);
            *(tex+0) = color[0];
            *(tex+1) = color[1];
            *(tex+2) = color[2];
            *(tex+3) = color[3];
            tex+=4;
            d++;
            if ((d-colormap_data) % width == 0) d++; // continue to
skip the left column of data
      }
      flg_NewTexture = 1;
}

template <class T>
void Patch<T>::BindNewTexture(void)
{
      // Bind this texture, deleting/unbinding old texture if necessary
      // This routine is called once for each GL context
      // Delete any textures if there are any
      if (glIsTexture(stuff->tex_id) == GL_TRUE) {
            glDeleteTextures(1, &(stuff->tex_id));
      }

      // Now bind the new texture
        glEnable(GL_TEXTURE_2D);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

                    // Generate OpenGL texture IDs.
        glGenTextures(1, &(stuff->tex_id));

        glBindTexture(GL_TEXTURE_2D, stuff->tex_id);

        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, // RGBA textures.
             width-1, width-1, 0, GL_RGBA,
             GL_UNSIGNED_BYTE,
             pTexture);

}
// End of 1-9-2k1 changes

template <class T>
```

```
void Patch<T>::Split(TriTreeNode *tri)
{
      //We are already split, so there is no need to split again
      if (tri->LeftChild)
            return;
      //If this triangle is not in a proper diamond,
      //    force split the base neighbor.
      // This means, if this triangle's base is not shared as the base
of
      // another triangle (This triangle leans on the side of another,
      // not on the bottom), then if we split this one, we must split
all
      // the way back to a base.
      if (tri->BaseNeighbor && (tri->BaseNeighbor->BaseNeighbor !=
tri))
            Split(tri->BaseNeighbor);

      // Setup the two childrenfor the split
      tri->LeftChild = info->land->AllocateTri();
      tri->RightChild = info->land->AllocateTri();

      // If there was no available nodes for the children, then
      // exit gracefully.
      if (!tri->LeftChild)
            return;
      if (!tri->RightChild)
            return;
      // Point the new child to the current base & neighbors
      tri->LeftChild->BaseNeighbor  = tri->LeftNeighbor;
        tri->LeftChild->LeftNeighbor  = tri->RightChild;

        tri->RightChild->BaseNeighbor  = tri->RightNeighbor;
        tri->RightChild->RightNeighbor = tri->LeftChild;

      // Point the left neighbor directly to the newly created children
      if (tri->LeftNeighbor != NULL) {
            // One of the left's neighbor's neighbor-pointers should
            // point to me.. Find out which it is, and point it at
            // the new child
            if (tri->LeftNeighbor->BaseNeighbor == tri)
                        tri->LeftNeighbor->BaseNeighbor = tri-
>LeftChild;
                else if (tri->LeftNeighbor->LeftNeighbor == tri)
                        tri->LeftNeighbor->LeftNeighbor = tri-
>LeftChild;
                else if (tri->LeftNeighbor->RightNeighbor == tri)
                        tri->LeftNeighbor->RightNeighbor = tri-
>LeftChild;
                else
                        ;// Illegal Left Neighbor!
        }
        // Link our Right Neighbor to the new children
        if (tri->RightNeighbor != NULL)
```

```
        {
                if (tri->RightNeighbor->BaseNeighbor == tri)
                        tri->RightNeighbor->BaseNeighbor = tri-
>RightChild;
                else if (tri->RightNeighbor->RightNeighbor == tri)
                        tri->RightNeighbor->RightNeighbor = tri-
>RightChild;
                else if (tri->RightNeighbor->LeftNeighbor == tri)
                        tri->RightNeighbor->LeftNeighbor = tri-
>RightChild;
                else
                        ;// Illegal Right Neighbor!
        }

        // Link our Base Neighbor to the new children
        if (tri->BaseNeighbor != NULL)
        {
                if ( tri->BaseNeighbor->LeftChild )
                {
                        tri->BaseNeighbor->LeftChild->RightNeighbor =
tri->RightChild;
                        tri->BaseNeighbor->RightChild->LeftNeighbor =
tri->LeftChild;
                        tri->LeftChild->RightNeighbor = tri-
>BaseNeighbor->RightChild;
                        tri->RightChild->LeftNeighbor = tri-
>BaseNeighbor->LeftChild;
                }
                else
                        Split( tri->BaseNeighbor);  // Base Neighbor
(in a diamond with us) was not split yet, so do that now.
        }
        else
        {
                // An edge triangle, trivial case.
                tri->LeftChild->RightNeighbor = NULL;
                tri->RightChild->LeftNeighbor = NULL;
        }
}

// ---------------------------------------------------------------------
-
// Tessellate a Patch.
// Will continue to split until the variance metric is met.
//
template <class T>
void Patch<T>::RecursTessellate( TriTreeNode *tri,
                                                                int leftX,
int leftY,
                                                                int rightX,
int rightY,
                                                                int apexX,
int apexY,
```

```
                                                                int node )
{
        float TriVariance, d_num;
        int centerX = (leftX + rightX)>>1; // Compute X coordinate of
center of Hypotenuse
        int centerY = (leftY + rightY)>>1; // Compute Y coord...

        if ( node < (1<<info->variance_depth) )
        {
                // Extremely slow distance metric (sqrt is used).
                // Replace this with a faster one!
                float distance = 1.0f + ( SQR((float)centerX - info-
>gViewPosition[0]) +
                                                SQR((float)centerY -
info->gViewPosition[2]) );
            if (distance>=VAR_MAX)
                  d_num = 1.0/distance; //info-
>TriVarianceList[(int)VAR_MAX-1];
            else
                  d_num = info->TriVarianceList[(int)distance];

                // Egads!  A division too?  What's this world coming
to!
                // This should also be replaced with a faster
operation.
                TriVariance = ((float)m_CurrentVariance[node] * info-
>map_size * 2) * d_num;;
                   //* d_num; // Take both distance and variance into
consideration
        }

        if ( (node >= (1<<info->variance_depth)) || (TriVariance >
info->gFrameVariance))
        {
                Split(tri); // Split this triangle.

                if (tri->LeftChild &&  // If this triangle was split,
try to split it's children as well.
                           ((abs(leftX - rightX) >= 3) || (abs(leftY -
rightY) >= 3)))   {
                        RecursTessellate( tri->LeftChild,   apexX,
apexY, leftX, leftY, centerX, centerY,    node<<1  );
                        RecursTessellate( tri->RightChild, rightX,
rightY, apexX, apexY, centerX, centerY, 1+(node<<1) );
                }
        }
}

// ---------------------------------------------------------------------
-
// Render the tree.  Simple no-fan method.
//
template <class T>
```

```
void Patch<T>::RecursRender( TriTreeNode *tri, int leftX, int leftY,
int rightX, int rightY, int apexX, int apexY )
{
      TriangleInfo *t;
        if ( tri->LeftChild )                                    // All
non-leaf nodes have both children, so just check for one
        {
                int centerX = (leftX + rightX)>>1;      // Compute X
coordinate of center of Hypotenuse
                int centerY = (leftY + rightY)>>1;      // Compute Y
coord...

                RecursRender( tri->LeftChild,  apexX,   apexY, leftX,
leftY, centerX, centerY );
                RecursRender( tri->RightChild, rightX, rightY, apexX,
apexY, centerX, centerY );
        }
        else
// A leaf node!  Output a triangle to be rendered.
        {
                // Actual number of rendered triangles...
                info->gNumTrisRendered++;
            t=&Triangle[TriangleIndex++];
                GLfloat leftZ  = height_data[((leftY) *width)+leftX ];
                GLfloat rightZ = height_data[((rightY)*width)+rightX];
                GLfloat apexZ  = height_data[((apexY) *width)+apexX ];

                // Perform polygon coloring based on a height sample
            t->vertex[0][0]=leftX;
            t->vertex[0][1]=leftZ;
            t->vertex[0][2]=leftY;
            t->texture[0][0] = (GLfloat)(leftX)/(width);// +
(1.0/width);
            t->texture[0][1] = (GLfloat)(leftY)/(width); // +
(1.0/width);


                t->vertex[1][0]=rightX;
                t->vertex[1][1]=rightZ;
                t->vertex[1][2]=rightY;
            t->texture[1][0] = (GLfloat)(rightX)/(width); // +
(1.0/width);
            t->texture[1][1] = (GLfloat)(rightY)/(width); // +
(1.0/width);


                t->vertex[2][0]=apexX;
                t->vertex[2][1]=apexZ;
                t->vertex[2][2]=apexY;
            t->texture[2][0] = (GLfloat)(apexX)/(width); // +
(1.0/width);
            t->texture[2][1] = (GLfloat)(apexY)/(width); // +
(1.0/width);
```

```
        }
}

// ------------------------------------------------------------------
-
// Computes Variance over the entire tree.  Does not examine node
relationships.
// NOTE: left, right & APex are in LOCAL coordinates (locations within
the patch)
template <class T>
float Patch<T>::RecursComputeVariance( int leftX,  int leftY,  T leftZ,
                                       int rightX, int rightY, T rightZ,
                                       int apexX,  int apexY,  T apexZ,
                                       int node)
{
        //        /|\
        //      /  |  \
        //    /    |    \
        //  /      |      \
        //  ~~~~~~~*~~~~~~~  <-- Compute the X and Y coordinates of '*'
        //
        int centerX = (leftX + rightX) >>1;             // Compute X
coordinate of center of Hypotenuse
        int centerY = (leftY + rightY) >>1;             // Compute Y
coord...
        float myVariance;
      float tempvar;

        // Get the height value at the middle of the Hypotenuse
        T centerZ  = height_data[((centerY) * width) + centerX];

        // Variance of this triangle is the actual height at it's
hypotenuse midpoint minus the interpolated height.
        // Use values passed on the stack instead of re-accessing the
Height Field.
        myVariance = abs(centerZ - ((int)(leftZ + rightZ)>>1));
        // Since we're after speed and not perfect representations,
        //    only calculate variance down to an 8x8 block
        if ( (abs(leftX - rightX) >= 8) ||
                  (abs(leftY - rightY) >= 8) )
        {
                // Final Variance for this node is the max of it's own
variance and that of it's children.
            tempvar = RecursComputeVariance( apexX,   apexY,  apexZ,
                                             leftX, leftY, leftZ,
                                             centerX, centerY, centerZ,
                                             node<<1 );
                myVariance = MAX( myVariance,tempvar);
      //Left Child Node

            tempvar = RecursComputeVariance( rightX, rightY, rightZ,
                                             apexX, apexY, apexZ,
                                             centerX, centerY, centerZ,
```

```
                                                1+(node<<1));
                myVariance = MAX( myVariance,tempvar);
      //Right Child Node
        }
        // Store the final variance for this node.  Note Variance is
never zero.
        if (node < (1<<info->variance_depth)) {
                m_CurrentVariance[node] = info->MinVariance +
myVariance;
      }
        return myVariance;
}


// ---------------------------------------------------------------------
-----------------------------
//      PATCH CLASS
// ---------------------------------------------------------------------
-----------------------------

// ---------------------------------------------------------------------
-
// Initialize a patch.
//
template <class T>
void Patch<T>::Init(int worldX, int worldY, int size, MetaData<T>
*ls_info )
{
        // Clear all the relationships
        m_BaseLeft.RightNeighbor = m_BaseLeft.LeftNeighbor =
m_BaseRight.RightNeighbor = m_BaseRight.LeftNeighbor =
            m_BaseLeft.LeftChild = m_BaseLeft.RightChild =
m_BaseRight.LeftChild = m_BaseLeft.LeftChild = NULL;

        // Attach the two m_Base triangles together
        m_BaseLeft.BaseNeighbor = &m_BaseRight;
        m_BaseRight.BaseNeighbor = &m_BaseLeft;

      info = ls_info;
        // Store Patch offsets for the world and heightmap.
        m_WorldX = worldX;
        m_WorldY = worldY;
      if (width != size) {
              width = size+1;
              // Store pointer to first byte of the height data for
this patch.
            if (height_data) {
                  delete height_data;
            }
            height_data=new T[(size+1) * (size+1)];
            if (colormap_data) {
                  delete colormap_data;
            }
```

```
            colormap_data=new T[(size+1) * (size+1)];
      }

      Triangle = new TriangleInfo[(size+1) * (size+1)*2];
      TriangleIndex = 0;

      m_VarianceLeft = new float[1<<info->variance_depth];
        m_VarianceRight = new float[1<<info->variance_depth];

        // Initialize flags
        m_VarianceDirty = 1;

      // Allocate Texture
      if (pTexture!=NULL) {
            delete pTexture;
      }
      pTexture = new unsigned char[size * size *4]; // 4 bytes for RGBA
      if (pTexture == NULL) {
            cout << "ERROR: Unable to allocate RAM" << endl;
      }
}

template <class T>
float Patch<T>::getData(int x, int y)
{
      return height_data[(y * width) + x];
}

// ---------------------------------------------------------------------
-
// Reset the patch.
//
template <class T>
void Patch<T>::Reset()
{
        // Reset the important relationships
        m_BaseLeft.LeftChild = m_BaseLeft.RightChild =
m_BaseRight.LeftChild = m_BaseLeft.LeftChild = NULL;

        // Attach the two m_Base triangles together
        m_BaseLeft.BaseNeighbor = &m_BaseRight;
        m_BaseRight.BaseNeighbor = &m_BaseLeft;

        // Clear the other relationships.
        m_BaseLeft.RightNeighbor = m_BaseLeft.LeftNeighbor =
m_BaseRight.RightNeighbor = m_BaseRight.LeftNeighbor = NULL;
      TriangleIndex = 0;
}

// ---------------------------------------------------------------------
-
// Compute the variance tree for each of the Binary Triangles in this
patch.
```

```
//
template <class T>
void Patch<T>:: ComputeVariance()
{
        // Compute variance on each of the base triangles...

        m_CurrentVariance = m_VarianceLeft;
        RecursComputeVariance(  0,          width-1,
height_data[(width) * (width-1) ],
                                width-1,      0,
height_data[width-1],
                                0,          0,          height_data[0],
                                1);

        m_CurrentVariance = m_VarianceRight;
        RecursComputeVariance(  width-1,      0,          height_data[
width-1],
                                0,          width-1,      height_data[
(width) * (width-1)  ],
                                width-1,      width-1,
height_data[(width * width)-1],
                                1);

        // Clear the dirty flag for this patch
        m_VarianceDirty = 0;
}

// ------------------------------------------------------------------------
-
// ------------------------------------------------------------------------
-
// Set patch's visibility flag.
//
template <class T>
void Patch<T>::SetVisibility( GLdouble *proj_matrix, GLdouble
*model_matrix, GLint *viewport )
{
        // Get patch's center point
      if ((IsSeen(m_WorldX, m_WorldY, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX+width, m_WorldY, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX, m_WorldY+width, proj_matrix, model_matrix,
viewport)==1) ||
          (IsSeen(m_WorldX+width, m_WorldY+width, proj_matrix,
model_matrix, viewport) == 1)) {
            stuff->m_isVisible = 1;
      } else {
            stuff->m_isVisible = 0;
      }
}
```

```
// ------------------------------------------------------------------------
-
// Create an approximate mesh.
//
template <class T>
void Patch<T>::Tessellate()
{
        // Split each of the base triangles
        m_CurrentVariance = m_VarianceLeft;
        RecursTessellate(&m_BaseLeft,
                         m_WorldX,        m_WorldY+width-1,
                         m_WorldX+width-1,  m_WorldY,
                         m_WorldX,        m_WorldY,
                         1 );

        m_CurrentVariance = m_VarianceRight;
        RecursTessellate(&m_BaseRight,
                         m_WorldX+width-1,m_WorldY,
                         m_WorldX,      m_WorldY+width-1,
                         m_WorldX+width-1,m_WorldY+width-1,
                         1 );
}

// ------------------------------------------------------------------------
-
// Render the mesh.
//
template <class T>
void Patch<T>::BuildRender()
{
        RecursRender (  &m_BaseLeft,
                0,     width-1,
                width-1, 0,
                0,     0);

        RecursRender(   &m_BaseRight,
                width-1, 0,
                0,     width-1,
                width-1, width-1);
}

template <class T>
void Patch<T>::Render()
{
      long index;
      TriangleInfo *t;
        glPushMatrix();
      if ((flg_NewTexture==1) || (m_WorldY != stuff->y_pos) ||
(m_WorldX != stuff->x_pos)) {
            BindNewTexture();
            stuff->y_pos = m_WorldY;
            stuff->x_pos = m_WorldX;
      }
```

```
      glColor4f(1,1,1,1);
      glBindTexture(GL_TEXTURE_2D, stuff->tex_id);
        // Translate the patch to the proper world coordinates
        glTranslatef( (GLfloat)m_WorldX, 0, (GLfloat)m_WorldY );
      // Bind to colormapped texture
        glBegin(GL_TRIANGLES);
      t = &Triangle[0];
      for(index=0; index<TriangleIndex; index++) {
//          glNormal3fv(t->normal);
            glTexCoord2fv(t->texture[0]);
            glVertex3fv(t->vertex[0]);

                glTexCoord2fv(t->texture[1]);
                glVertex3fv(t->vertex[1]);

                glTexCoord2fv(t->texture[2]);
                glVertex3fv(t->vertex[2]);
            t++;
      }
        glEnd();

        // Restore the matrix
        glPopMatrix();
}

template <class T>
void Patch<T>::RenderTexPanel()
{
        if ((m_WorldY != stuff->y_pos) || (m_WorldX != stuff->x_pos)) {
                BindNewTexture();
                stuff->y_pos = m_WorldY;
                stuff->x_pos = m_WorldX;
        }
      glBindTexture(GL_TEXTURE_2D, stuff->tex_id);
      glBegin(GL_QUADS);
      glTexCoord2f(0,1);
      glVertex2f(0,0);

      glTexCoord2f(1,1);
      glVertex2f(1,0);

      glTexCoord2f(1,0);
      glVertex2f(1,1);

      glTexCoord2f(0,0);


      glVertex2f(0,1);
      glEnd();
}
```

2. Landscape.cpp

```
#include <iostream>
#include <GL/glu.h>
#include <fstream>
#include <string>
#include <math.h>

using namespace std;

#include "Landscape.h"

VCRoamSurfaceFloat::VCRoamSurfaceFloat(void)
{
      myplot = NULL;
}

VCRoamSurfaceFloat::~VCRoamSurfaceFloat(void)
{
      if (myplot)
            delete (myplot);
}

void VCRoamSurfaceFloat::Draw(void)
{
        glPushMatrix();
        myplot->Render();

        glPopMatrix();
      glBindTexture(GL_TEXTURE_2D, 0);
}

void VCRoamSurfaceFloat::PerFrameCalculations(void)
{
      vjMatrix dummy1;
      vjVec3 location;

        dummy1 = user->getPosition();
        location=dummy1.getTrans();
        myplot->info.gViewPosition[0]=location[0];
        myplot->info.gViewPosition[1]=location[1];
        myplot->info.gViewPosition[2]=location[2];

        //myplot->info.gClipAngle = -gAnimateAngle;

        myplot->Reset();
        myplot->Tessellate();
}

void VCRoamSurfaceFloat::PostRender(void)
{
```

```
      myplot->PostRender();
}

int VCRoamSurfaceFloat::Status(void)
{
      return 0; // This component never deletes itself
}

void VCRoamSurfaceFloat::InitGL(void)
{
      myplot->InitGL();

}

void VCRoamSurfaceFloat::Init(DataManager *DM, UserData *userdata,
string filename, string header)
{
   CfgHeader = header;
   CfgFileName = filename;
   user = userdata;
   myplot = new ROAMLandScape<float>(DM , CfgFileName, header);
   myplot->info.gViewPosition[0] = ReadIntFromFile(CfgFileName,
"SYSTEM", "startx", 0);
   myplot->info.gViewPosition[1] = 0;
   myplot->info.gViewPosition[2] = ReadIntFromFile(CfgFileName,
"SYSTEM", "starty", 0);
   myplot->info.cmap.LoadColormap(ReadStringFromFile(CfgFileName,
header, "colormap", ""));
   myplot->info.cmap.SetLand(0, 1, 0, 1);
   myplot->info.user = user;
   myplot->Init();

}


template <class T>
ROAMLandScape<T>::ROAMLandScape(DataManager *DM, string filename,
string Header)
{
       LoadConfigFile(filename, Header);

      info.m_TriPool = new TriTreeNode[info.max_nodes];

      cout << "ROAM:constructor --> Building new Tile Manager" << endl;
      TileManager = DM->GetDataBase(ReadStringFromFile(filename,
Header, "data", ""));
      TileManager->setLayer(ReadIntFromFile(filename, Header,
"data_layer", 1));
      info.layer = TileManager->getLayer();

      info.land = this;
      if(ReadStringFromFile(filename, Header, "color", "") == "shared")
{
```

```
            cout << "--> Using Shared Colormap/Heightmap data" << endl;
            color_data = TileManager;
      } else {
            cout << "--> Not Using Shared Colormap/Heightmap data" <<
endl;
            color_data = DM->GetDataBase(ReadStringFromFile(filename,
Header, "color", ""));
            color_data->setLayer(ReadIntFromFile(filename, Header,
"color_layer", 1));
      }


      for(int index=0;index<VAR_MAX;index++)
            info.TriVarianceList[index] = 1.0 / (float)(index);
      m_Patches = NULL;
}

template <class T>
void ROAMLandScape<T>::LoadConfigFile(string filename, string Header)
{
      info._patchsize = ReadIntFromFile(filename, Header, "patchsize",
0);
      info._viewsquare = ReadIntFromFile(filename, Header,
"viewsquare", 0);
      info.variance_depth = ReadIntFromFile(filename, Header,
"variance_depth", 0);
      info.max_nodes = ReadIntFromFile(filename, Header, "max_nodes",
0);
      info.gDesiredTris = ReadIntFromFile(filename, Header,
"gDesiredTris", 0);
      info.map_size = ReadIntFromFile(filename, Header, "map_size", 0);
      info.gDrawMode = ReadIntFromFile(filename, Header, "render", 0);
      info.gFrameVariance = ReadIntFromFile(filename, Header,
"FrameVariance", 0);
      info.AutoLOD = ReadIntFromFile(filename, Header, "AutoLOD", 0);
      info.MinVariance = ReadFloatFromFile(filename, Header,
"MinVariance", 0.0);
}

template <class T>
void ROAMLandScape<T>::InitGL(void)
{
      // This used to create the land texture
      // Obviously now removed
}


// ------------------------------------------------------------------
-----------------------------
//      LANDSCAPE CLASS
// ------------------------------------------------------------------
-----------------------------
```

```
// ------------------------------------------------------------------
-
// Allocate a TriTreeNode from the pool.
//
template <class T>
TriTreeNode *ROAMLandScape<T>::AllocateTri()
{
        TriTreeNode *pTri;

        // IF we've run out of TriTreeNodes, just return NULL (this is
handled gracefully)
        if ( info.m_NextTriNode >= info.max_nodes )
                return NULL;

        pTri = &(info.m_TriPool[info.m_NextTriNode++]);
        pTri->LeftChild = pTri->RightChild = NULL;

        return pTri;
}

// ------------------------------------------------------------------
-
// Initialize all patches
//
template<class T>
void ROAMLandScape<T>::Init(void)
{
        Patch<T> *patch;
      Patch_Info<T> *ppatch;
        int X, Y;
      int diff;
      int sx_offset, sy_offset;
      int halfdist;
      TileManager->setLayer(info.layer);

      // Make sure we load the data centered around the user, so go
      // half of the viewable distance to the left,right,front, and
back of the user
      halfdist = ((info._viewsquare-1) * (info._patchsize)) >>1 ;

      diff = ((int)info.gViewPosition[0] % (info._patchsize));
      sx_offset = (info.gViewPosition[0]-diff) - halfdist;

      diff = ((int)info.gViewPosition[2] % (info._patchsize));
      sy_offset = (info.gViewPosition[2]-diff) - halfdist;

      // Now this is fucked up, I can't explain this
      //  But it doesn't work without it
//    if (sx_offset < 0) sx_offset--;
//    if (sy_offset < 0) sy_offset--;

        // Initialize all terrain patches
        // Store the Height Field array
```

```
      if (m_Patches == NULL) {
            // The patches haven't been allocated yet.. This means we
need
            // to allocate memory & load all the patches.
            m_Patches = new
Patch_Info<T>[info._viewsquare*info._viewsquare];
            PatchPool = new
Patch<T>[info._viewsquare*info._viewsquare];
            patch = &(PatchPool[0]);
            ppatch = &(m_Patches[0]);
            for ( Y=0; Y < info._viewsquare; Y++)
                    for ( X=0; X < info._viewsquare; X++ )
                    {
                  ppatch->the_patch = patch;
                  ppatch->valid = TRUE;
                  ppatch->updated = TRUE;
                        patch->Init(sx_offset + X*(info._patchsize),
                              sy_offset+ Y*(info._patchsize),
                              info._patchsize,
                              &info);
                  // Load data now.. If no data is found, then this
patch is
                  // Invalid (outside data-able area)
                  if(TileManager->loadArea(sx_offset +
X*(info._patchsize),
                                    sy_offset + Y*(info._patchsize),
                                          info._patchsize+1,
                                    info._patchsize+1,
                                          patch->height_data) == -1 )
                        ppatch->valid = FALSE;
                  else  {
                        color_data->loadArea(sx_offset +
X*(info._patchsize),
                                                sy_offset +
Y*(info._patchsize),
                                                info._patchsize+1,
                                                info._patchsize+1,
                                                patch->colormap_data);
                              patch->ComputeVariance();
                  }
                  patch++;
                  ppatch++;
                    }
      } else {
            // The patches have been previously allocated, so they
contain data.
            // Find the valid data to preserve, and load the rest.
            // Algorithm:
            // First search all the data tiles.  If one is "Valid",
swap it to it's correct
            //    location and mark it valid.
            // Then pass through all "invalid" tiles and load them.
            int left, right, top, bottom;
```

```
            int patch_x, patch_y;
            int p2x, p2y;
            Patch<T> *patch_temp;
            Patch_Info<T> *dpatch; // Destination Patch

            left = sx_offset;
            top = sy_offset;
            right = left + (info._viewsquare-1)*(info._patchsize);
            bottom = top + (info._viewsquare-1)*(info._patchsize);

            // Start by marking them all invalid for later checks
            ppatch = &(m_Patches[0]);
            for ( X=0; X < info._viewsquare*info._viewsquare; X++,
ppatch++)  {
                  ppatch->updated = FALSE;
//                ppatch->valid = FALSE;
            }

            // Now find valid's/invalid's
            cout << "Swapping" << flush;
            ppatch = &(m_Patches[0]);
            for ( X=0; X < info._viewsquare*info._viewsquare; X++,
ppatch++)
                // First see if this patch is valid.
                if (ppatch->valid == TRUE) { // don't bother checking
if we know it's an empty/bad patch
                  (ppatch->the_patch)->getLocation(&patch_x, &patch_y);
                  // if this patch starts within the viewable area then
it must be valid.
                  // Because all patches are required to load on
(viewsquare-1 *n) boundaries.
                  if ((patch_x>=left) && (patch_x <= right) &&
(patch_y>=top) && (patch_y<=bottom)) {
                        // Valid tile
                        // Find where this tile *should* be.
                        dpatch = &m_Patches[ int((patch_y - sy_offset)
/ (info._patchsize)) * info._viewsquare +
                                          int((patch_x - sx_offset) /
(info._patchsize))];
                        if (dpatch != ppatch) { // Make sure it belongs
somewhere other than where it is.
                              // see (if due to a glitch I can't seem
to fix any other way) the
                              // source & destination patches are not
the same, but both store the
                              // same data, then mark this one invalid
& kill it.
                              dpatch->the_patch->getLocation(&p2x,
&p2y);
                              if ((p2x==patch_x) && (p2y==patch_y)) {
                                    ppatch->valid = FALSE;
                                    X =
info._viewsquare*info._viewsquare;
```

```
                              } else {
                                    // Now swap the two patch
pointers..
                                    patch_temp = dpatch->the_patch;
                                    dpatch->the_patch = ppatch-
>the_patch; // This patch is valid now
                                    ppatch->the_patch = patch_temp;
                                    // The new patch is valid, we know
it.
                                    // The one we swapped it with may
or may not be, so just copy the status
                                    ppatch->valid = dpatch->valid;
                                    dpatch->valid = TRUE;
                                    // Now reset it.. We may have
swapped so that a tile will
                                    // be skipped, so restart at the
beginning.
                                    // Back up 1 because they will be
incremented
                                    // NOTE: Look at improving this
brute-force method.
                                    //    Possible recursion? That may
be worse..
                                    X = -1;
                                    ppatch = &(m_Patches[0]);
                                    ppatch--;
                                    cout << "." << flush ;
                              }
                        } // end of if dpatch != ppatch
                  } else {
                        // Invalid tile
                        ppatch->valid = FALSE;
                  } // end of if within left/top/bottom/right
                } // end of if valid

            // Now we know which ones are valid/invalid..
            // So reload the invalid ones.
                ppatch = &(m_Patches[0]);
                for ( Y=0; Y < info._viewsquare; Y++)
                    for ( X=0; X < info._viewsquare; X++, ppatch++ )
                  if(ppatch->valid == FALSE) {
                        // This patch isn't valid.. so reload.
                        ppatch->updated = TRUE;
                              (ppatch->the_patch)->Init(sx_offset +
X*(info._patchsize),
                                          sy_offset+ Y*(info._patchsize),
                                          info._patchsize,
                                          &info);
                              if(TileManager->loadArea(sx_offset +
X*(info._patchsize),
                                                sy_offset +
Y*(info._patchsize),
                                                info._patchsize+1,
```

```
                                                      info._patchsize+1,
                                                      (ppatch->the_patch)-
>height_data) == -1) {
                                ppatch->valid = FALSE;
                          } else  {
                                color_data->loadArea(sx_offset +
X*(info._patchsize),
                                                      sy_offset +
Y*(info._patchsize),
                                                      info._patchsize+1,
                                                      info._patchsize+1,
                                                      (ppatch->the_patch)-
>colormap_data);
                                      (ppatch->the_patch)-
>ComputeVariance();
                                ppatch->valid = TRUE;
                          }
                    }

      }
      info.height_dataMin = TileManager->getMin();
      info.height_dataMax = TileManager->getMax();
      if (info.height_dataMax < info.height_dataMin) {
            T tmp = info.height_dataMin;
            info.height_dataMin = info.height_dataMax;
            info.height_dataMax = tmp;
      }
      info.color_dataMin = TileManager->getMin();
      info.color_dataMax = TileManager->getMax();
      if (info.color_dataMax < info.color_dataMin) {
            T tmp = info.color_dataMin;
            info.color_dataMin = info.color_dataMax;
            info.color_dataMax = tmp;
      }

      info.cmap.SetMaxMin(info.color_dataMax, info.color_dataMin);
      ppatch = &(m_Patches[0]);
        for ( Y=0; Y < info._viewsquare*info._viewsquare; Y++){
            if((ppatch->valid == TRUE) && (ppatch->updated == TRUE))
                  (ppatch->the_patch)->GenerateTexture();
            ppatch++;
      }
      info.height_dataSpread = info.height_dataMax -
info.height_dataMin;
      info.height_dataSpread = info.height_dataMax -
info.height_dataMin;
}

template <class T>
void ROAMLandScape<T>::Reset(void)
{
        //
        // Perform simple visibility culling on entire patches.
```

```
        //   - Define a triangle set back from the camera by one patch
size, following
        //     the angle of the frustum.
        //   - A patch is visible if any of it's 4 corners are within
the viewable area
      //          -- uses gluUnProject to figure out
        //   - This visibility test is only accurate if the camera
cannot look up or down significantly.
        //
        int X, Y, size;
      int startx, starty;
        Patch_Info<T> *patch;
      size = info._viewsquare;
      Patch_Info<T> *pLeft, *pRight, *pTop, *pBottom;

        // Set the next free triangle pointer back to the beginning
        SetNextTriNode(0);

      // See if we are outside the center patch
      patch = &m_Patches[(size * size) >> 1]; // set Patch to center
patch
                                        // NOTE: Only works when size is an
odd number
      if (patch->valid == TRUE) {
          (patch->the_patch)->getLocation(&startx, &starty);
          if ((info.gViewPosition[0] < startx) ||
(info.gViewPosition[0] > startx+info._patchsize+1) ||
            (info.gViewPosition[2] < starty) || (info.gViewPosition[2]
> starty+info._patchsize+1)) {
            // We are outside the center patch, so reload data to put
us inside the center patch
            vjDEBUG(vjDBG_ALL, 1) << "User at: " <<
info.gViewPosition[0] << "," <<
                  info.gViewPosition[2] << endl << vjDEBUG_FLUSH;
            vjDEBUG(vjDBG_ALL, 1) << "Center tile at: " << startx <<
"," << starty << endl << vjDEBUG_FLUSH;
            vjDEBUG(vjDBG_ALL, 1) << "Not Centered. Reloading\n" <<
endl << vjDEBUG_FLUSH;
            // Insert code to swap stuff out HERE
            Init();
          }
      }
        // Reset rendered triangle count.
        info.gNumTrisRendered = 0;
      patch=&(m_Patches[0]);
        // Go through the patches performing resets, compute variances,
and linking.
        for ( Y=0; Y < info._viewsquare; Y++ )
                for ( X=0; X < info._viewsquare; X++)
                {
                        // Reset the patch
                if (patch->valid == TRUE) { // If patch is invalid,
then don't bother rebuilding it
```

```
                                    // Prevents "edge of the world" bug
                                    // Edge of world is now darkness..
Just like the europeans thought :)
                        (patch->the_patch)->Reset();

                        // Check to see if this patch has been deformed
since last frame.
                        // If so, recompute the varience tree for it.
                        if ( (patch->the_patch)->isDirty() )
                                (patch->the_patch)->ComputeVariance();
                  pTop = &(m_Patches[(Y-1)*size + X]);
                  pBottom = &(m_Patches[(Y+1)*size +X]);
                  pLeft = &(m_Patches[(Y*size) + X-1]);
                  pRight = &(m_Patches[(Y*size) + X+1]);

                        // Link all the patches together.
                        if ((X > 0) && (pLeft->valid == TRUE) )
                                (patch->the_patch)->GetBaseLeft()-
>LeftNeighbor = pLeft->the_patch->GetBaseRight();
                        else
                                (patch->the_patch)->GetBaseLeft()-
>LeftNeighbor = NULL;              // Link to bordering Landscape
here..

                        if (( X < (size-1)) && (pRight->valid == TRUE)
)
                                (patch->the_patch)->GetBaseRight()-
>LeftNeighbor = pRight->the_patch->GetBaseLeft();
                        else
                                (patch->the_patch)->GetBaseRight()-
>LeftNeighbor = NULL;              // Link to bordering Landscape here..

                        if (( Y > 0) && (pTop->valid == TRUE) )
                                (patch->the_patch)->GetBaseLeft()-
>RightNeighbor = pTop->the_patch->GetBaseRight();
                        else
                                (patch->the_patch)->GetBaseLeft()-
>RightNeighbor = NULL;              // Link to bordering Landscape
here..

                        if (( Y < (size-1)) && (pBottom->valid == TRUE)
)
                                (patch->the_patch)->GetBaseRight()-
>RightNeighbor = pBottom->the_patch->GetBaseLeft();
                        else
                                (patch->the_patch)->GetBaseRight()-
>RightNeighbor = NULL;    // Link to bordering Landscape here..
                }
                patch++;
                }
}
```

```
// ------------------------------------------------------------------
-
// Create an approximate mesh of the landscape.
//
template<class T>
void ROAMLandScape<T>::Tessellate()
{
        // Perform Tessellation
      int x, count;
        Patch_Info<T> *patch = &(m_Patches[0]);
      count = info._viewsquare * info._viewsquare;

        for(x=0;x<count; x++, patch++){
            if (patch->valid == TRUE)
                  (patch->the_patch)->Tessellate( );
      }
        for(patch = &(m_Patches[0]),x=0;x<count; x++, patch++){
            if (patch->valid == TRUE)
                  (patch->the_patch)->BuildRender();
        }
}

template <class T>
float ROAMLandScape<T>::GetAltitude(float x, float y)
{
        float corners[2][2], topApprox, botApprox;
        float ix, iy;
      ix = ftrunc(x);
      iy = ftrunc(y);
        corners[0][0] = getData(ix, iy);
        corners[0][1] = getData(ix+1, iy);
        corners[1][0] = getData(ix, iy+1);
        corners[1][1] = getData(ix+1, iy+1);

        topApprox = corners[0][0] +
                (x - ix)*(corners[0][1] - corners[0][0]);
        botApprox = corners[1][0] +
                (x - ix)*(corners[1][1] - corners[1][0]);
        return(topApprox +
                (y - iy)*(botApprox - topApprox));

}

// ------------------------------------------------------------------
-
// Render each patch of the landscape & adjust the frame variance.
//
template <class T>
void ROAMLandScape<T>::Render()
{
        int x,y ;
        Patch_Info<T> *patch = &(m_Patches[0]);
      GLdouble projection[16], model[16];
```

```
      GLint viewport[4];

      glDisable(GL_LIGHTING);
      glGetDoublev(GL_PROJECTION_MATRIX,projection);
      glGetDoublev(GL_MODELVIEW_MATRIX, model);
      glGetIntegerv(GL_VIEWPORT,viewport);
      // Set visibility on patches (View Frustrum Culling)

for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare; x++,
patch++)
               if (patch->valid == TRUE)
                 (patch->the_patch)->SetVisibility(projection, model,
viewport);
      // Force nearby patches to be visible (reduce blicker);
      //  (for the unknowing.. blicker = flicker + blink )
      //  Blicker (c) Randall Hand 2001
      for(x=-1;x<=1; x++) {
            for(y=-1; y<=1; y++) {

      MakeVisible(info.gViewPosition[0]+(x*(info._patchsize)),
                        info.gViewPosition[2]+(y*(info._patchsize)));
            }
      }
      // Render Visible Patches
      if ((info.user)->Render_Wireframe) {
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            glEnable(GL_CULL_FACE);
      } else {
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      }

for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare; x++,
patch++){
            if ((patch->valid == TRUE) && ((patch->the_patch)-
>isVisible()))   {
                 (patch->the_patch)->Render();
            }
        }

      glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      glDisable(GL_CULL_FACE);
      DrawMap();
}

template <class T>
void ROAMLandScape<T>::DrawMap(void)
{
      vjVec3 angles;
      int x;
        Patch_Info<T> *patch = &(m_Patches[0]);

        glDisable(GL_DEPTH_TEST);
```

```
      for(x=0,patch=&(m_Patches[0]);x<info._viewsquare*info._viewsquare
; x++, patch++){
            if(patch->valid == TRUE) {
                  glPushMatrix();
                  glLoadIdentity();
                  glTranslatef(0, 5, 0);
                  glScalef(0.5,0.5,0.5);
                  glTranslatef((x % info._viewsquare), 10-
int(x/info._viewsquare),-15);
                  (patch->the_patch)->RenderTexPanel();
                  glPopMatrix();
            }
      }

      glPushMatrix();
      glLoadIdentity();
      glTranslatef(0,5,0);
      glScalef(0.5,0.5,0.5);
      glTranslatef((float)info._viewsquare/2.0, 11.0 -
((float)info._viewsquare/2.0), -15);
      angles = (info.user)->GetUserOrientation();
      if (angles[0] == -180) {
            glRotatef(-(180-angles[1]), 0,0,1);
      } else {
            glRotatef(-angles[1],0,0,1);
      }
      glBindTexture(GL_TEXTURE_2D, 0);
      glColor4f(1,1,1,1);
      glBegin(GL_TRIANGLES);
         glVertex3f(0,0,0);
         glColor4f(1,1,1,0);
         glVertex3f(-2, 3, 0);
         glVertex3f(2, 3, 0);
      glEnd();
      glPopMatrix();


        glEnable(GL_DEPTH_TEST);

}



template <class T>
void ROAMLandScape<T>::PostRender(void)
{
        // Check to see if we got close to the desired number of
triangles.
        // Adjust the frame variance to a better value.
      int x;
      Patch<T> *patch;
      if (info.AutoLOD == 0)
            return;
```

```
        if ( GetNextTriNode() != info.gDesiredTris )
                info.gFrameVariance += ((float)GetNextTriNode() -
(float)info.gDesiredTris) / (float)info.gDesiredTris;

        // Bounds checking.
        if ( info.gFrameVariance < 0 )
                info.gFrameVariance = 0;


for(x=0,patch=&(PatchPool[0]);x<info._viewsquare*info._viewsquare; x++,
patch++){
              patch->ClearFlags();
      }
}

template <class T>
void ROAMLandScape<T>::SetDrawModeContext()
{
        switch (info.gDrawMode)
        {
        case DRAW_USE_TEXTURE:
                glDisable(GL_LIGHTING);
                glEnable(GL_TEXTURE_2D);
        //        glPolygonMode(GL_FRONT, GL_FILL);
                break;

        case DRAW_USE_LIGHTING:
                glEnable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_FILL);
                break;

        case DRAW_USE_FILL_ONLY:
                glDisable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_FILL);
                break;

        default:
        case DRAW_USE_WIREFRAME:
                glDisable(GL_LIGHTING);
                glDisable(GL_TEXTURE_2D);
                glPolygonMode(GL_FRONT, GL_LINE);
                break;
        }
}

template <class T>
float ROAMLandScape<T>::getData(int x, int y)
{
      int size;
      int startx, starty;
      int px, py;
```

```
        Patch_Info<T> *patch = &(m_Patches[0]);
      size = info._patchsize;

      (patch->the_patch)->getLocation(&startx, &starty);
      px = (x - startx) / size;
      py = (y - starty) / size;

      patch = &(m_Patches[(py * info._viewsquare) + px]);
      (patch->the_patch)->getLocation(&startx, &starty);

      return (patch->the_patch)->getData(x-startx, y-starty);
}

template <class T>
void ROAMLandScape<T>::MakeVisible(int x, int y)
{
      int size;
      int startx, starty;
      int px, py;
        Patch_Info<T> *patch = &(m_Patches[0]);
      size = info._patchsize;

      (patch->the_patch)->getLocation(&startx, &starty);
      px = (x - startx) / size;
      py = (y - starty) / size;

      patch = &(m_Patches[(py * info._viewsquare) + px]);
      (patch->the_patch)->ForceVisibility();
}
```