Real-Time Dynamic Level of Detail Terrain Rendering with ROAM

Bryan Turner, 2000

Introduction

Like many people, I find photographs of rolling hills or perilous canyons both calming and awe-inspiring. It is unfortunate that as gamers, we are not able to revel in the natural beauty of the outdoors. Only a few current and upcoming games give us this feast for the eyes (*Tribes 1 & 2, Tread Marks, Outcast, Myth 1 & 2, and HALO are a few examples*). These games have taken 3D action gaming to the next level with the inclusion of incredibly detailed worlds upon which the story and action are played out.

In this article I will briefly examine the state of the art in hardware accelerated landscape engines and the algorithms which power them. One algorithm in particular will be presented, discussed, and finally implemented as a starting point for anyone looking to add landscapes to their next project. I'll assume an intermediate level of C++ knowledge and at least general knowledge of 3D rendering.

Introduction to Terrain Visualization

It seems you can't shake a stick in the world of terrain visualization without hitting a reference to *Level of Detail (LOD) Terrain Algorithms*. Level of Detail algorithms use a set of heuristics to determine which parts of a landscape need more detail to look correct. You'll no doubt find a slew of references to *Height Fields* and GPS datasets too. All of this is perpetrated by the military *SimNet* applications; however, it is now finding its way into more trivial pursuits.

One of the many technical challenges to terrain rendering is how to store the features inherent in a landscape. Height Fields are the de-facto standard solution. Simply put, they are two-dimensional arrays which hold the height of the terrain at that point. Think of a piece of graph paper where a surveyor has filled in each square using his altitude measuring tools. Height Fields are sometimes called Height Maps; I will use the two terms interchangeably.

Overview of LOD Terrain Algorithms

A good overview of LOD Terrain Algorithms can be represented by three papers: [1 Hoppe] [2 Lindstrom] [3 Duchaineau]. In [1], Hugues Hoppe presents an algorithm based on *Progressive Meshes*, a relatively new and spiffy technique for adding triangles to arbitrary meshes as you need more detail. The paper is an excellent read, but a bit complex and has high memory requirements for our needs.

The second paper [2] is more our style. Lindstrom et. Al. present a structure called a *Quad Tree* that is used to represent a patch of landscape. A Quad Tree recursively tessellates the landscape creating an approximation of the Height Field. Quad Trees are very simple and efficient, sharing many of the design principles of the next algorithm (such as recursion); However, the added bonuses of the next paper tilt the scale.

Finally, in [3] Duchaineau et. Al. present an algorithm (*Real-time Optimally Adapting Meshes*) based on a *Binary Triangle Tree* structure. Here, each patch is a simple isosceles right triangle. Splitting the triangle from its apex to the middle of its hypotenuse produces two new

isosceles right triangles. The splitting is recursive and can be repeated on the children until the desired level of detail is reached.

The *ROAM algorithm* caught my eye while researching due to its simplicity and extensibility. Unfortunately, the paper is extremely short and only minimal pseudocode is presented to hint at implementations. However, it can be implemented from the most basic level up to the most advanced optimizations in a nearly continuous spectrum. This is helpful since each step can be validated before continuing. Also, ROAM tessellates very rapidly and allows dynamic updates to the Height Map.

The engine presented here was patterned after the engine in *Tread Marks* (http://www.treadmarks.com). The lead programmer, Seumas McNally, was instrumental from its conception to completion. See the Acknowledgements at the end for more info.

Introduction to the ROAM Implementation

The code in the archive is written for Visual C++ 6.0 and uses OpenGL to perform the rendering. I am new to OpenGL, but I have used every available means to code this aspect of the project correctly. Comments and suggestions on the engine's design or implementation are welcome.

The project contains several files that are not covered in this explanation. These files consist of utility routines and general application overhead needed to run an OpenGL/Win32 application. Only *"ROAMSimple.cpp"* and associated header files are examined here.

ROAM Source Explanation

Let me introduce the algorithm with a bird's-eye view and then we can focus on how the individual pieces interact:

- *Height Map* files are loaded into memory and associated with an instance of a *Landscape* class. Multiple *Landscape* objects may be linked to generate terrains of infinite size.
- A new *Landscape* object parcels out sections of the loaded *Height Map* to new *Patch* class objects. The purpose for this step is two-fold:
    1. The tree-based structures used for the rest of the algorithm expand RAM usage exponentially with depth, so keeping the areas small limits their depths.
    2. Dynamic updates of the *Height Field* need a complete recalculation of the variance tree over the modified locations. Overly large *Patches* would be too slow to recompute in a real-time application.
- Each *Patch* object is then called to create a mesh approximation (tessellation). The Patches employ a structure called a *Binary Triangle Tree* which stores implicit coordinates for the triangles that will be displayed onscreen (instead of explicit, X, Y, Z coordinates). By storing the vertices in a logical manner, ROAM saves upwards of 36 bytes of RAM per triangle. Coordinates are calculated efficiently as part of the rendering step (below).
- After tessellation, the engine traverses the Binary Triangle Tree created in the previous step. Leaf nodes in the tree represent triangles which need to be output to the graphics pipeline. The triangle coordinates are calculated on the fly during the traversal.

Height Map File Format

I have chosen the simplest route, reading in raw data files containing 8-bit height samples in row major format. This happens to be the exact format my paint program outputs (by mere coincidence, of course). The Height Field is kept in memory at all times. I will discuss how to extend the algorithm to take on larger datasets in the Advanced Topics section.

Binary Triangle Trees

Instead of storing a huge array of triangle coordinates to represent the landscape mesh, the ROAM algorithm uses a structure called a *Binary Triangle Tree*. This structure can be viewed as the result of a surveyor cutting the landscape into triangular plots. The owners of these plots logically view each other in terms of neighbor-relationships (left/right neighbor, etc). Likewise, when an owner gives land as an inheritance, it is split equally between the two children.

To extend this analogy further, the original owner of a plot is the root node of a Binary Triangle Tree. Other original owners are root nodes of their own trees. The *Landscape* class acts like a local land-registry, keeping track of all the original owners and which plot they owned. The registry also keeps records of all inheritances from parents to children.

The more generations of children, the more heavily surveyed the land becomes. Any amount of detail can be produced simply by expanding the 'population' in areas which need better approximations. See Figure 1 for an example.

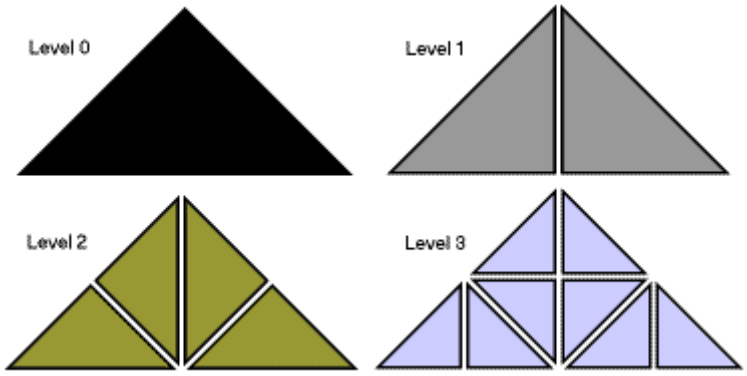

Figure 1 - Binary Triangle Tree structure levels 0-3.

Binary Triangle Trees are represented by the *TriTreeNode* structure and keep track of the five basic relationships needed for ROAM. Refer to Figure 2 for the standard view of these relationships.

```
struct TriTreeNode
{
	// Children
	TriTreeNode *LeftChild;
	TriTreeNode *RightChild;

	// Neighbors
	// Below us
	TriTreeNode *BaseNeighbor;
	// To our left and right
	TriTreeNode *LeftNeighbor;
	TriTreeNode *RightNeighbor;
};
```

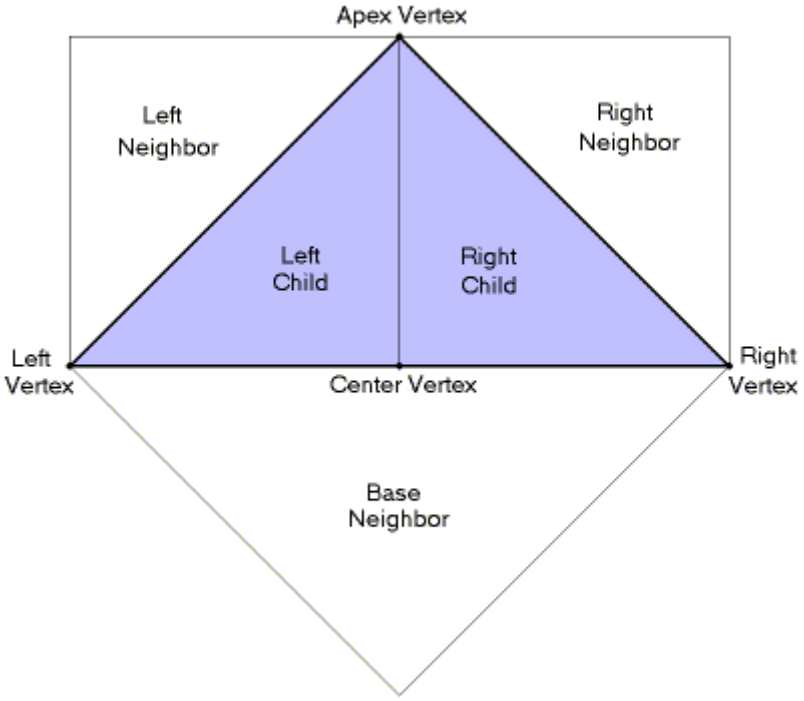

Figure 2 - Basic Binary Triangle with children and neighbors.

When creating a mesh approximation for the Height Field, we will recursively add children to the tree until the desired level of detail is reached. After this step is complete, the tree can be traversed again, this time rendering the leaf nodes as actual triangles onscreen. This two-pass system is the basic engine and required resetting for each frame. One nice feature of the recursive method is that we are not storing any per-vertex data, freeing up huge amounts of RAM for other goodies.

In fact, the TriTreeNode structures are created and destroyed so many times that the most efficient method of allocation is mandated. Also, there may be tens of thousands of these structures, so even one extra pointer would bloat the memory requirements tremendously. The TriTreeNode structures are allocated from a static pool, bypassing the overhead of dynamic memory allocation, which also gives us a rapid method for resetting the state.

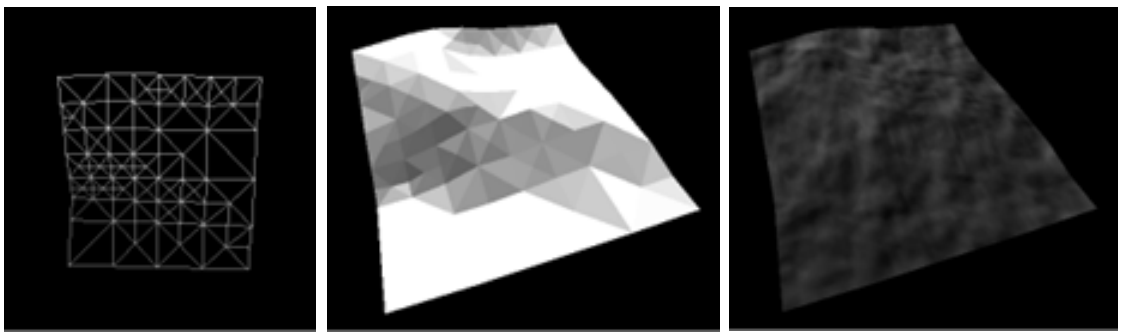

Figure 3 - Typical patch of terrain. From left to right, wireframe (overhead view), lit, and textured.

Explanation of Landscape Class

The Landscape class acts as the high-level encapsulation for the dirty details of landscape rendering. From the point of view of the application, the landscape should simply appear in the screen buffer after a few simple setup calls. Here's the important bits of the *Landscape* class definition:

```
class Landscape
{
public:
      // Initialize the whole process
      void Init(unsigned char *hMap);

      // Reset for a new frame
      void Reset();

      // Create mesh approximation
      void Tessellate();

      // Render the current mesh static
      void Render();

      // Allocate a new node for the mesh
      TriTreeNode *AllocateTri();

protected:
      // Index to the next free TriTreeNode
      static int m_NextTriNode;

      // Pool of nodes for tessellation
      static TriTreeNode m_TriPool[];

      // Array of patches to be rendered
      Patch m_aPatches[][];

      // Pointer to Height Field data
      unsigned char *m_HeightMap;
};
```

The Landscape class manages large square plots and can work together with other Landscape objects each with their own plots. This design comes into play later when you'll want to page-in larger terrain sets. During initialization, the Height Map is cut into more manageable pieces and given to new *Patch* objects. It is the Patch class and associated methods that we will spend the most time on.

Note the simplicity of the functions. The Landscape class is designed to be easily dropped into a rendering pipeline – especially given the gratuitous hardware z-buffering available these days. Several globals are used to further simplify this demo.

Explanation of Patch Class

The Patch class is the meat & potatoes of the engine. It is roughly broken into two halves, the stub half and the recursive half. Here's the data declaration and stub half of the Patch class:

```
class Patch
{
public:
      // Initialize the patch
      void Init(int height, int heightY, int worldX, int worldY, unsigned char *hMap);

      // Reset for next frame
      void Reset();
```

```
        // Create mesh
        void Tessellate();

        // Render mesh void
        void Render();

        // Update for Height Map changes
        ComputeVariance();

        ...

protected:
        // Adjusted pointer into Height Field
        unsigned char *m_HeightMap;

        // World coordinate offset for patch
        int m_worldX, m_worldY;

        // Left and right variance tree
        unsigned char m_VarianceLeft[];
        unsigned char m_VarianceRight[];

        // Pointer to current tree in use
        unsigned char *m_CurrentVariance;

        // Does variance tree need updating?
        unsigned char m_VarianceDirty;

        // Root node for left and right triangle trees
        TriTreeNode m_BaseLeft;
        TriTreeNode m_BaseRight;

        ...
};
```

In the flow of code, the stub functions explained below are called for each Patch object held by the parent Landscape. The Patch class method names are equivalent to the Landscape methods which call them. These methods are rather simplistic so there is no need for a detailed analysis:

- *Init()* requires the offsets into the Height Field array and World. These are used for scaling the patch over different sizes of terrain. The pointer to the Height Field is adjusted to point to the first byte of this patch's data and stored internally.
- *Reset()* erases any references to invalid TriTreeNodes, followed by relinking the two Binary Triangle Trees that make up each patch. It hasn't been mentioned until now, but each patch is actually made up of two discrete Binary Triangle Trees fitted together into a square (called a *'Diamond' in the ROAM paper*). Take a look at Figure 2 again if you're confused. Much more detail on this in the next section.
- *Tessellate()* is the first of our stub functions. It simply passes the proper parameters for the highest-level triangles (the two root nodes from each patch) on to the recursive version of the function. Same goes for *Render()* and *ComputeVariance()*.

ROAM Guts

So far, we've only discussed the support structure for the actual algorithm. Now it's time to get to the goods. It might be handy to have a copy of the ROAM paper at this point, but I'll explain it as we go. Refer back to Figure 2 with the triangle relationships, and stele yourself for the next phase.

First, we must define a metric for visible error in a mesh approximation. The method I use is a clone of the Tread Marks engine called *'Variance'*. We will need such a metric for deciding when a node should be split (to add detail), and how deeply to split it. The ROAM paper uses a metric based on nested world-space bounds. While this metric is more accurate, it is also vastly slower.

Variance is the difference in height of the interpolated hypotenuse midpoint for a binary triangle node and the actual Height Field sample at that point. Simply put, how far off is the current binary triangle node from the actual Height Field area it covers. This calculation is relatively quick and only required one memory hit for the Height Field lookup:

```
triVariance = abs(centerZ – ((leftZ + rightZ) / 2));
```

But wait, there's more! We can't just calculate the variance for the two root Binary Triangle Trees of each Patch because the error associated with this calculation is too high. It has to be calculated deeper into the tree, then averaged back up to get a better estimate. The depth of this calculation for the demo can be specified at compile time.

Normally, the variance calculation would be required for each frame, however it won't change unless the underlying Height Field changes. Therefore, we introduce a *Variance Tree* which works alongside the Binary Triangle Tree.

A Variance Tree is a full-height binary tree written into a sequential array. A few simple macros allow us to navigate the tree efficiently, and the data we fill it with is a single byte value of difference per node. Refer to Figure 4 if you've not encountered this structure before. Two variance trees are stored in the patch class, one each for the Left and Right Binary Triangles.

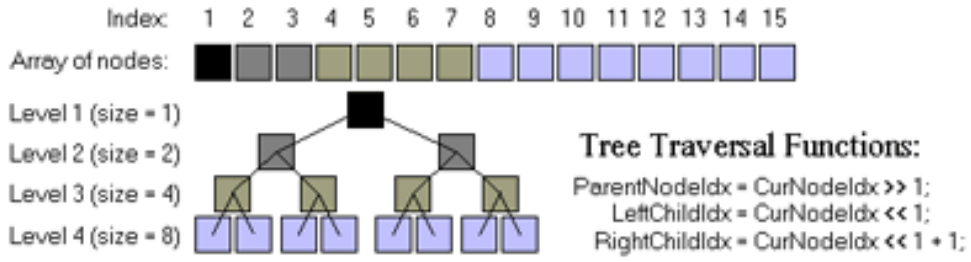

Figure 4 - Implicit binary tree structure.

Now we can get back to the job of creating an approximate mesh. Given our error metric (variance), we will decide to split the Binary Triangle node over a particular spot if its variance is too high. That is, if the terrain under the current triangle is bumpy, then we should split it to give a better approximation. Splitting entails creating two child nodes that exactly fill the parent triangle's area (see Figure for an example).

After moving down to the children, we repeat the process. The variance roughly drops in half each iteration. At some point we either find smooth enough terrain to approximate with a single triangle, or we run out of 'steps'… after all, we can only create meshes down to the resolution of the Height Field, no more.

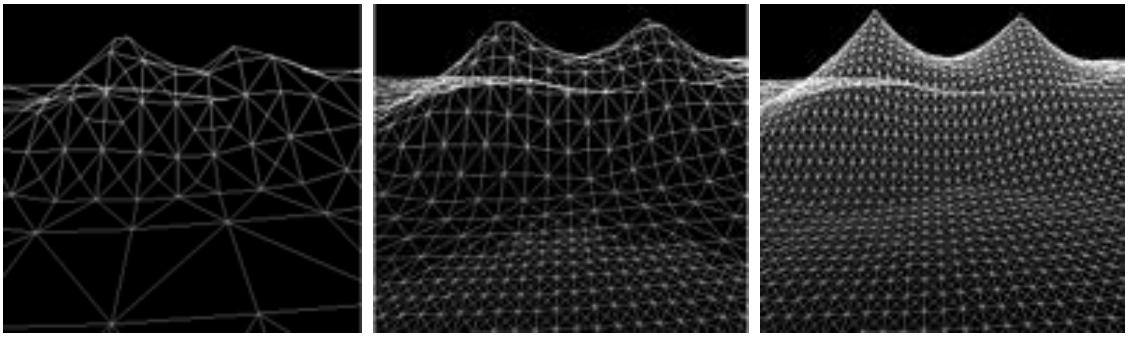

Figure 5 - Terrain displayed at low, optimal and high variance settings.

There's still one more complication. When splitting Binary Triangle Trees that are adjacent on the landscape, cracks will often appear in the mesh. These cracks are due to uneven splitting of the trees across patch boundaries. This problem is illustrated in Figure 6.

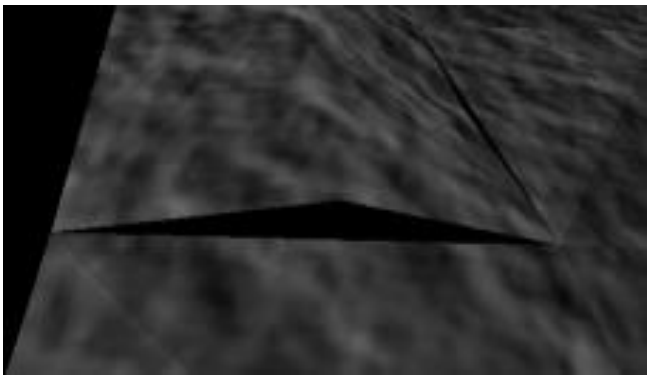

Figure 6 - Patch showing cracks in mesh.

To fix this, ROAM makes use of the neighbor pointers and an interesting fact of the mesh itself: neighbors of a particular node are either of the same level, one level finer (for right/left neighbors) or one level coarser (for base neighbors). We apply this during the creation of the mesh in order to keep neighboring trees in sync with us.

It comes down to a simple rule: only split if the current node and its base neighbor both point to each other (see Figure 7). This relationship is referred to as a Diamond. It is special because a split of one node in a Diamond can be mirrored by the other without causing cracks in the mesh.

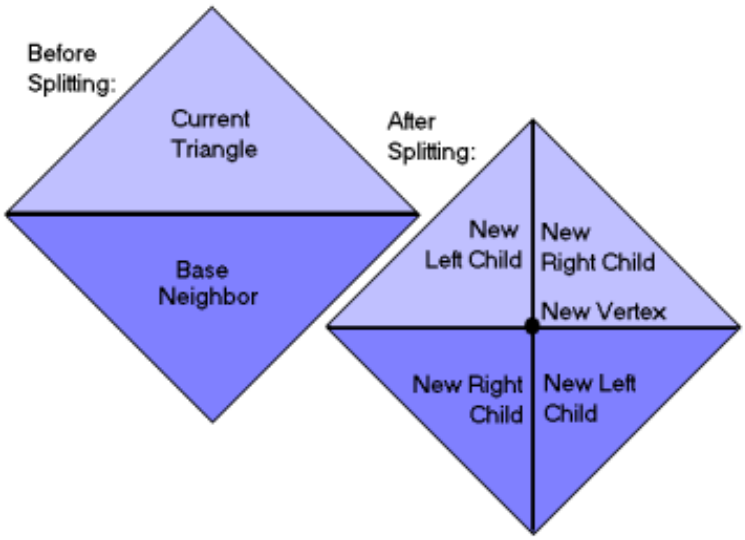

Figure 7 - Split operation on a diamond.

Three cases exist when attempting to split a node:

1. The Node is part of a Diamond: split the node and its base neighbor.
2. The Node is on the edge of the mesh: trivial, only split the node.
3. The node is not part of a diamond: force split the base neighbor.

A Forced Split is a recursive traversal of the mesh which ends when it finds a Diamond or an edge triangle. Here's how it works: when splitting a node, check to see that it is part of a Diamond first. If not, call a second split operation on the Base Neighbor to create a Diamond, then continue with the original split.

The second call to split will do the same check and recurs the process again if need be. Once a node is found that can be split legally, the recursion unwinds, splitting nodes along the way. Figure 8 illustrates this.

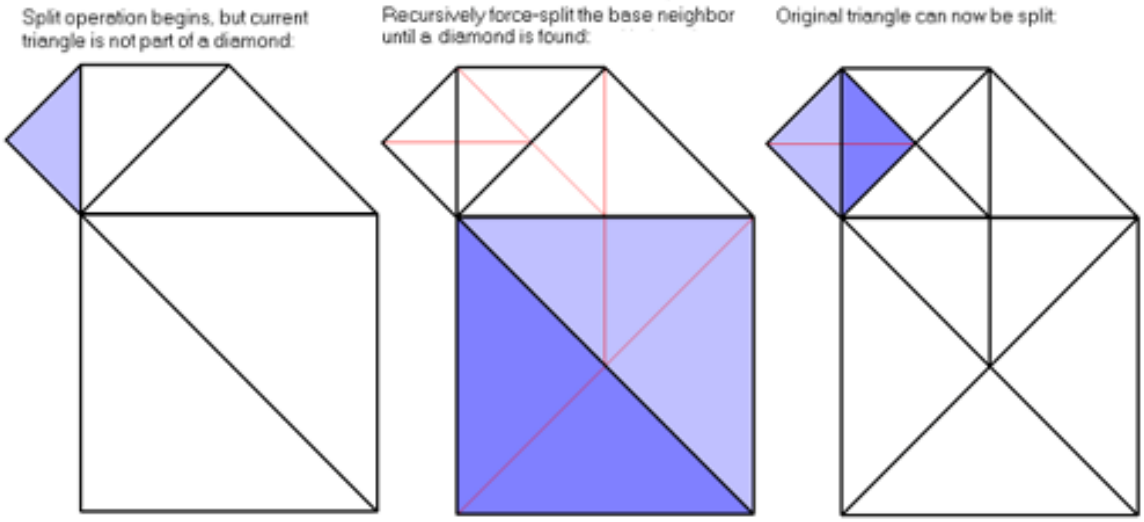

Figure 8 - Forced split operation.

So, let's review. Given a patch made up of two Binary Triangle Trees covering a particular area of the Height Field, we will perform the following operations:

1. Compute Variance Tree: fill out a full-height binary tree with variance data for each Binary Triangle Tree. 'Variance' is the metric we are using to determine if our approximation is good enough. It is the difference between the height sample at the middle of the hypotenuse versus the interpolated height from the two points which border the hypotenuse.
2. Tessellate the Landscape: using the variance tree we will split our Binary Triangle Trees by adding children if the variance of the top level is undesirably high.
3. Forced Splits: if the node we are attempting to split is not part of a Diamond, then call a Forced Split on the offending node. This will give us a Diamond to complete the original split operation.
4. Repeat: the tessellation step is repeated on the children until all the triangles in the Binary Triangle Tree are under the variance limit for the current frame – or until we run out of nodes in our allocation pool.

Return to Patch Guts

Now that we have all the details of the ROAM algorithm, let's finish up the Patch class implementation. All of the recursive functions (except Split) take coordinates for the triangles

they represent. These coordinates are calculated on the stack and passed down to the next level or given to OpenGL for rendering. Even at the deepest level of the Binary Triangle Tree, there are rarely more than thirteen triangles on the stack.

This is the basic algorithm for recursion that the following functions use:

```
// X and Y coords for Hypotenuse center
int centerX = (leftX + rightX) / 2;
int centerY = (leftY + rightY) / 2;

// Recurse Left and Right
Recurse(apexX, apexY, leftX, leftY, centerX, centerY);
Recurse(rightX, rigthY, apexX, apexY, centerX, centerY);
```

Recursive Patch Class Functions:

```
void Patch::Split(TriTreeNode *tri);

unsigned char Patch::RecurseComputeVariance(int leftX, int leftY, unsigned char leftZ,
int rightX, int rightY, unsigned char rightZ, int apexX, int apexY, unsigned char apexZ,
int node);

void Patch::RecurseTessellate(TriTreeNode *tri, int leftX, int leftY, int rightX, int
rightY, int apexX, int apexY, int node);

void Patch::RecurseRender(TriTreeNode *tri, int leftX, int leftY, int rightX, int rightY,
int apexX, int apexY);
```

Split() performs all the ROAM splitting, including the Forced Split operation. It checks for a legal Diamond, allocates child nodes, links them into the surrounding mesh, and calls additional Splits where needed.

RecurseComputeVariance() takes the full set of coordinates for the current triangle and a few extra bits of info to keep track of where we are. Variance for the triangle is calculated and combined with that of its children. I chose to pass in the height for each point as well as its X & Y coordinates in order to reduce the memory hits on the Height Field array.

RecurseTessellate() is where the Level of Detail operation is performed. After calculating distance to the camera, it adjusts the variance of the current node by a factor of the distance. This makes closer nodes have larger variances. The resulting mesh will use many triangles near the camera and fewer in the distance. Distance is calculated using a square root for simplicity (which is slow and should be replaced with a faster method).

RecurseRender() is remarkably simple, but take a look at the Triangle Fanning optimization under Advanced Topics for the next step up from here. Basically, if the current triangle is not a leaf node, recurs into the children. Otherwise, output a single triangle to OpenGL. Note that the OpenGL rendering is not optimized, but rather designed for maximum readability. That's all, folks! We've covered everything you'll need to understand the code. The rest is icing for those who want to take the next step. But first, I'll give some engine qualifiers, and a note on the variance calculations.

Roam Engine Qualifiers

Platform: Win98, AMD K6-2 450 MHz, 96 Mb RAM, NVIDIA GeForce 256 DDR video.

Resolution: 640x480, 32bit color.

Note on Variance: Variance is the single most important variable in this engine. It has been glossed over in this article to present the overall framework. Try modifying the calculation used for the variance tree, or the tree's depth. Specifically, set depth to an extremely small value like 3, then try a large one like 13 and note the difference in rendering quality.

The variance is also used during tessellation. Adjust the calculation for the current node to ignore distances entirely. Then try it based purely on distance, ignoring the value from the variance tree. It will be up to you to find calculations which work best for your application.

Desired # of TriTree Nodes	Textured FPS	Solid-Fill FPD
5000	57	62
10000	30	36
15000	20	25
20000	16	19

Other Engines in the Industry

In the process of writing this article, I contacted several developers in industry for comments and poked a few questions about their engines.

Starseige:Tribes (www.dynamix.com)

Starseige Tribes is an online-only game of fast-paced squad warfare. The game is played seamlessly between indoor and outdoor environments where terrain features are extremely strategic to the success of a mission. Long distance kills are commonplace, and enemies may hide behind hills to avoid detection. Mark Frohnmayer responded to questions about *Starseige Tribes* for this article.

- **On Height Maps:** Mark Frohnmayer: "The heights in *Tribes 1* are stored on a regular 8m square grid. *Tribes 2* grid size is selectable by mission."
- **On the Engine:** Mark Frohnmayer: "The *Tribes 1* terrain engine used a distance-based quad tree algorithm created by Tim Gift. For *Tribes 2* we came up with a new approach to screen-error based on edge traversal. Our approach makes up for the limitations of current published algorithms, including texturing with a bin-tree approach and seaming up edges between squares in quadtrees algorithms."

  They chose a Quad Tree algorithm, see [2] for reference on the basics of this system. I'm puzzled over the new screen error metric, however from the screenshots of *Tribes 2*, it works amazingly well! Perhaps in the coming months they will enlighten us with more details.

- **Other Comments:** Mark Frohnmayer: "Texturing has been the single biggest headache with landscapes in the game – allowing the mission editor to select a texture for every square and dynamically generate the combination textures for squares that are at a lower detail level, as well as automatically texturing the terrain based on vertex material. The most interesting problem was coming up with the algorithm to detail the terrain mesh that solved all of our requirements."

  I mentioned before the shear depth of the landscape visualization field, and Mark's comment hints at this. The included engine is only a drop in the bucket. Be sure to check out a good technical library or the many online sites devoted to this topic.

  Mark goes on to say that their engine is not frame-coherent, requiring a rebuild of the Quad Tree each frame. This is required for their view metric and clipping code.

Outcast (www.outcast-thegame.com)

Outcast is a deeply designed action/adventure game with exquisite role-play elements. The unique look and feel of *Outcast* plays beautifully into the alien landscapes and immersive storyline. Christophe Chaudy responded to questions about *Outcast* for this article.

- **On Height Maps:** Christophe Chaudy: "We are using tiled heightmaps. Each heightmap's instance contains its specific scale, offset and some color modifiers. With only 150 or 200 unique heightmaps we can build a huge world with more than 36000 tiles of 10x10 meters. It was difficult to deal with map continuity at tile edges but that's the price to pay – and our graphics people rock!"
- **On the Engine:** Christophe Chaudy: "We started with the software voxel engine. During the production of *Outcast*, seeing the explosion of 3D HW market, we looked at polygon terrain rendering algorithms but:
    - We were not able to recreate the complexity, diversity and density of voxel terrain with polygons.
    - We need to output A LOT of polygons to achieve a valid terrain quality (too many for 1998-99 3D HW).
    - In *Outcast*, we used a lot of post process algorithms which operate directly on the frame/Z buffer to render special effects like fog, water & shadows. These techniques could not be easily implemented on 1999 3D HW.
    - We didn't have time to rebuild a completely different render system.

  So, we stayed with the software rendering approach. There were a lot of drawbacks but finally, even if the terrain renderer is not perfect, it's something that looks different. And that was a very important criteria in *Outcast* production."

- **Other Comments:** Christophe Chaudy: "Voxel rendering is very constraining; very CPU intensive, strong modeling constraints, low resolution, poor quality on low-end computer, etc. But the landscape in *Outcast* is unique. Even today, I don't see a polygon engine that can reproduce such geometric complexity. But, under the market pressure, we are designing a totally new system for outdoor visualization. It will of course use today 3D hardware rendering. In the future, when the CPU is faster, the ray-casting algorithm on voxel heightmaps could come back."

  Outcast is by far the most unique looking game on the market. If you have not seen or experienced the world of *Outcast*, make sure to swing by the homepage. Also take a look at the presentation on the rendering technology of *Outcast* from last year's Game Developer's Conference (see the Bibliography for a link).

Advanced Topics

As promised, here are a few hints and tips for advanced optimizations and features. Each could be its own article, so I've attempted to distill the most important aspects into a few short paragraphs on each topic.

Triangle Fanning

Triangle Fanning is an optimization you can use when triangles all share a central vertex. It allows you to specify fewer vertexes for the same number of triangles, giving an improvement in overall speed. Triangle Fans in OpenGL flow clockwise, as do the points in each of the triangles. You will have to switch which side is the triangle face or OpenGL will cull out all your triangles!

In order to get the correct output of triangles, it helps to switch the order in which child nodes are visited at each level of rendering. Thus, if we visit the Left Child first at level 1, then visit the Right Child first for level 2, then back to the Left Child for level 3.

The order of the vertexes is important too. The first vertex specified must be the central point around which the other triangles 'fan' out. This is done by passing down a reference to one of the triangle's vertexes as the "best center vertex". At each level, this value is switched to point to a new best vertex. When a leaf is found, it is added to a small buffer of vertexes with the "best vertex" first and the others in clockwise order.

At the next leaf node, we need only compare the "best vertex" to the first vertex in the buffer. If they differ, output the fan to OpenGL and start over. However, if the two vertexes are equal, then the last vertex in the buffer equal to the next clockwise vertex in the triangle. Again, if they differ, output the fan to OpenGL and start over. Otherwise, append the last vertex of the triangle to the end of the vertex buffer.

Fan lengths cannot be more than 8 triangles using this method, however average lengths are more commonly 3-4 triangles per fan.

GeoMorphing

An unfortunate side effect of rendering with dynamic levels of detail is the sudden visual 'pop' that occurs when triangles are inserted or removed from the mesh. This distortion can be reduced to nearly unnoticeable amounts by Vertex Morphing, also called GeoMorphing. GeoMorphing is the gradual rise or fall of a vertex's height from the un-split position to its new split position over the course of several frames.

GeoMorphing is not difficult but has a lot of tricky aspects. Essentially, a value may be stored in the TriTreeNode during tessellation which contains the amount of 'morph' this triangle has. This morph value should be in the range 0.0-1.0. Then, during rendering, transmute from the interpolated height value to the actual Height Field value using the following function:

```
MorphedZ = (fMorph * actualZ) + ((1 – fMorph) * interpolated);
```

Frame Coherence

Frame Coherence is the most advanced optimization under ROAM. With frame coherence, the mesh which was created last frame can be used again. This feature also enabled dynamic frame timing, allowing you to continue to improve the mesh for the current frame right up to the frame's deadline.

In a fast-action game, this means you don't have to spend all the overhead to tessellate the landscape. Instead, deal with the most important fast-action components first, then tessellate the landscape for the rest of the frame time and render what you have at the end. So, if a player is in the middle of a firefight, the landscape will dynamically render at lower detail to save time.

It is beyond the space for this article to explain the implementation of Frame Coherence. However, a few tips for the traveler: add a 'Parent' pointer to TriTreeNode. Create a Merge() function which undoes one Split() operation. Use a priority queue or other priority structure which contains all leaf nodes in the entire mesh. During the tessellation, merge any nodes which are too detailed for this frame followed by splitting all the nodes which are too coarse for the frame (or until time runs out).

Supporting Larger Topologies

The included engine is structured to simplify the creatin of very large worlds. By loading separate height maps for each Landscape class and then rendering each Landscape, there is no limit to its size! There are other limits, however, like RAM and computational power.

The Landscape class was designed to hold a paged-in piece of the world, along other Landscape classes holding other blocks. Each Landscape must link its patches to those in the other Landscapes nearby. This is done in Patch::Reset(), instead of setting the Neighbor pointers for edge nodes to NULL, lookup the correct patch in the Landscape which borders that side.

Speculations on the Future

The future of landscape rendering is wide open. No doubt the polygon count will continue to rise as will the detail of environments and the distances to be viewed. Also, the current LOT algorithms are not designed to take advantage of the new graphics cards which offload triangle setup calculations. This reduces the algorithm's gains for certain applications.

Additionally, OpenGL display lists might be used to render an entire landscape, then sent to the graphics card in one fell swoop each frame. This is feasible for small terrains like this demo and the faster memory busses of the future. We may even see a re-emergence of software rendered voxel landscapes, given the availability of fast CPUs and the inherent advantage of voxel displays.

Acknowledgements

This article was inspired by many people and many projects I have seen. First and foremost are Seumas McNally and the *Tread Marks* engine, from which this project was modeled after. Visit http://www.longbowdigitalarts.com to join in the lively programming forum or learn more about Tread Marks.

I would also like to thank the many terrain visualization projects in the public domain, including ROAM.C by C. Cookson. Also, the many great programming articles in the Gamasutra Features archive, and the super-programmers of the Gamasutra Connections board (I didn't even have to ask questions, the answers were already there!).

And to the industry gurus who reviewed this article and made suggestions, thanks again for the input. It is my sincerest desire to see more outdoor games and epic journeys in the coming years. I hope this article may inspire new projects for the genre.

Bibliography and References

1. Hoppe, H. "Smooth View-Dependent Level-of-Detail Control and its Applications to Terrain Rendering" (http://www.research.microsoft.com/~hoppe).
2. Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L., Faust, N., Turner, G., "Real-Time Continuous Level of Detail Rendering of Height Fields" (http://www.cc.gatech.edu/gvu/people/peter.lindstrom/papers/siggraph96).
3. Duchaineau, M. Wolinski, M., Sigeti, D., Miller, M., Aldrich, C., and Mineev-Weinstein, M., "ROAMing Terrain: Real-time Optimally Adapting Meshes" (http://www.llnl.gov/graphics/ROAM).

4. Binary Triangle Tree Article (http://www.longbowdigitalarts.com/seumas/progbintri.html).
5. VTerrain.org Engine Comparison List (http://vterrain.org/LOD/runtime-reg.html).
6. Outcast Engine Technology Presentation for GDC (http://www.appeal.be/products/page1/Outcast_GDC/outcast_gdc_1.htm).
7. Longbow Digital Arts Programming Discussion Forum (http://www.longbowdigitalarts.com).
8. Gamasutra Features Archive (http://www.gamasutra.com).
9. GameDev Programming Archives (http://www.gamedev.net).

Epilogue

On Tuesday, March 21st 2000, Seumas McNally lost his battle with Hodgkins Lymphoma.

My sincerest condolences go out to his surviving family, Jim, Wendy and Philippe. I never had the chance to meet Seumas personally, nor thank him for the encouragement and free exchange of ideas he gave. His passing is a great loss to our community of developers. May his commitment, determination, and unfailing humanity live on as an example for us all. Goodbye, Seamus – and Thank You.

Bryan Turner is a freelance research programmer continually in search of cutting-edge technologies. If you would like to contact him, e-mail bryan.turner@pobox.com.