

Terrain Rendering at High Levels of Detail

Jonathan Blow

jon@number-none.com

Bolt Action Software

March 11, 2000

In the course of developing a terrain renderer that processes geometry and textures at high levels of detail, we have found the need to extend existing terrain algorithms and develop new techniques.

This paper is divided into two parts. The first part is a detailed disclosure of a subsystem we have developed for handling high levels of texture detail in a scalable way. The second part is a series of less-formal notes that describe the content of the lecture; these notes discuss work that is more recent, and they take a wider survey of the terrain rendering situation.

Part I:

Fast Reduction of Texture Memory Usage for Terrain Rendering

Abstract

When rendering terrains that utilize high resolution texture maps, texture memory management can quickly become the performance bottleneck. We present a system that computes a small working set of texture maps for each frame to be rendered. Utilizing frame coherence and a simple recurrence relation, the algorithm computes the texture resolutions necessary to render the scene, requiring only a few arithmetic operations per polygon.

Introduction

When rendering terrains by polygonal tessellation, we often use texture maps to enhance the appearance of the scene. In the case of a terrain renderer, we may use an aerial photograph of the terrain as a large texture map shared by all the polygons. We want this texture map to be displayed at a high resolution, so that the terrain appears realistic. However, for terrains of sufficient size and detail, the memory requirements of such a texture map become prohibitive.

The usual solution is to construct a quadtree that addresses small portions of the texture. This approach is explored in e.g. [Phototextured]. Indexing texture space with a quadtree is more of a general idea than an algorithm. Here we present a specific system that uses such a quadtree to manage texture resolution. Our algorithm behaves in a similar way: a quadtree organizes our working set of textures, with a texture's position in the tree determined by its resolution and position in global texture space.

A function maps each rendered polygon to a node in the quadtree; this tells us which portions of the texture quadtree are necessary for each frame. Only the textures assigned to these nodes are instantiated, thus saving on texture storage space. Furthermore, distinct portions of the quadtree can be instantiated at different resolutions; we only use as much resolution for a particular terrain

region as we need to make the output look good. This makes practical the rendering of terrain using high-detail textures.

There are many ways such a quadtree can be constructed, and many potential mappings from displayed polygons to quadtree regions. The main contribution of this paper is a fast algorithm to determine the necessary texture detail given the set of output polygons. Utilizing frame coherence, affinity between data structures, and a simple recurrence relation, and by exploiting the properties of IEEE-754 floating-point numbers, the algorithm reduces the computational overhead of texture optimization to a few arithmetic operations per polygon per frame.

Another approach, the “clipmap”, has been proposed in [Clipmap]. We dislike clipmaps because they require hardware support, are difficult to interface with software, and do not provide the scaling properties we desire. Full discussion of clipmaps is beyond the scope of this paper, but the reader is encouraged to consult the reference.

We proceed in the following order:

- * Summary of the method used to generate terrain polygons.
- * Brief presentation of the quadtree used to store texture map information.
- * Discussion of the equations used to compute texture resolution.
- * Presentation of the optimizations we employ.
- * Discussion of implementation details.
- * Conclusion and future work.

1. Generating Terrain Polygons

Here we discuss the algorithms used to create terrain polygons, the data structure used to organize those polygons, and how that structure affects the computation of texture resolutions.

1.1 Summary of the polygon generation algorithm

We generate terrain as a continuous mesh of polygons stored in a binary triangle tree. The binary triangle tree is discussed in the papers on ROAM [ROAM] and the Lindstrom height field algorithm [Lindstrom]. It is a desirable data structure because it facilitates crack-fixing and produces nicely-shaped polygons.

The terrain itself consists of a two-dimensional array of Bezier patches, on which are layered procedurally-generated displacement maps. Thus we can inspect the terrain to an arbitrarily fine level of detail, and there are always new features to see. The polygon generation algorithm uses hierarchical bounding volumes around the terrain to expand the binary triangle tree at its most appropriate nodes, in order to minimize error in the rendered image, while keeping the polygon count low.

Because we are able to achieve fine levels of geometric detail, we want to color those polygons using high-resolution texture maps; otherwise the image would look poorer than it should, texture map resolution having become the main barrier to high image quality.

The algorithm that generates terrain polygons is tangential to this paper, so we will not describe it further. For texture map generation, the most important fact is that the tessellated terrain is

represented by a binary triangle tree. We will heavily exploit the structure of this tree. We present a review of the tree's properties.

1.2 The properties of the binary triangle tree

Each node of a binary triangle tree represents a right isosceles triangle. The tree has two roots, two triangles that share a hypotenuse. As such, the triangles form a square the size of the entire terrain (which must be square). The height values at the four corners of the square are made identical to the height values of the terrain at its corners. This is shown in figure 1a.

Any node of the tree may be subdivided to create two children. This subdivision is enacted by bisecting the triangle with a line that passes through its apex and the midpoint of its hypotenuse. The midpoint of the hypotenuse is then given a new elevation value, derived from the source data. This process is shown in figure 1b.

We assign an integer level index to each node in the tree. The roots are defined to be level zero. The index of any other node is one greater than the index of its parent. Figure 1c shows the tree fully subdivided to various levels. Finally, figure 1d shows a tree that has been adaptively subdivided to a variety of depths. The tree for a rendered terrain will look like this, though it will be subdivided to a much finer detail level.

When generating texture maps, we exploit the fact that all nodes of the binary triangle tree are right isosceles triangles with sizes and orientations that are constrained by their level indices.

2.1 Organizing texture map data

In the application for which we developed this technique, the terrain is textured by a map containing 2^{36} pixels. This map cannot be contained in memory on consumer hardware, as it is larger than a 32-bit microprocessor can address. Furthermore, it is several orders of magnitude too large to be utilized by a hardware graphics accelerator.

Thus we can only afford to instantiate the portions of the texture that we need at a given time. We use a quadtree to divide the terrain texture into organized units. The root of the quadtree represents a square area that covers the entire terrain. Its children are four congruent squares that each represent a quarter of the terrain. Any node of this tree can be expanded to yield four new children. Each node in the tree represents a square texture.

Note that there is redundancy in this tree with respect to the area covered by each node. A node covers the same area of the map as its children; thus when we wish to texture a terrain region that fits within a node, we could theoretically use that node or any of its parents, because they all cover the region. There is a limit to how deeply we can descend the tree; generally, rendering primitives cannot concatenate multiple textures to tile across a polygon. Thus we cannot descend into child nodes that would cease to cover the polygon. (Unless we then choose to subdivide the polygon, a subject that we won't explore here.)

We do not want to choose textures that are too high in the quadtree, because this would waste texture memory. But we also do not want to choose textures that are too low in the quadtree; doing so will greatly increase the number of unique textures used by the system, which will negatively affect rendering performance (for example, by causing many more context switches on graphics hardware).

Generally, a rendering primitive will have some maximum and minimum texture sizes that can be efficiently used. For example, a hardware graphics implementation may limit the size of a texture to 2048x2048 pixels; using textures as small as 32x32 pixels would be inefficient due to context switches.

The task of our texture generation algorithm is to dynamically match these resolution constraints against the resolution needs of the scene to be rendered; the algorithm chooses a set of textures that cover the terrain at the necessary level of detail, but do not waste much memory, and do not waste many context switches.

There is an affinity in structure between the quadtree and the binary triangle tree; the binary triangle tree is like a quadtree that proceeds downward in half-steps rather than full steps, and that divides space triangularly rather than squarely. Since those triangles themselves are halves of squares, it is easy to map terrain polygons into the quadtree areas that cover them.

2.2 The spaces that concern us

When deriving our resolution computations, we will discuss the vertices of a triangle with respect to three different spaces. *World space* is the coordinate system that contains our source data, e.g. the positions of points in an elevation grid. *Texture space* is the two-dimensional space defined by a triangle's texture coordinates. We will speak of associating a single triangle with many different textures; each pairwise association creates a new set of texture coordinates and thus defines a new texture space. *Screen space* is the two-dimensional space representing triangles' coordinates within the viewport. It is computed from world space by a transform that involves perspective warping.

When a triangle is bisected to add a new vertex and increase detail, this bisection occurs in world space. The texture space coordinates of the original vertices are not affected, and the new vertex receives texture coordinates at the midpoint of the hypotenuse in texture space. Thus the distance between the base vertices in texture space does not change; but because the new vertex in world space is vertically displaced, the lengths of the edges in world space become longer with respect to their texture space analogues. That is, the texture on a triangle is stretched slightly with each detail enhancement or reduction, as shown in figure 2. Traditionally, terrain rendering algorithms ignore this effect. We choose to ignore it as well; the whole idea of these detail reduction algorithms is that they only make changes to the rendered triangles when the amount of introduced warping is very small when projected to the viewport.

3. Computing necessary texture resolution

There are a number of ways to compute the texture resolution necessary for a triangle. One such way is described by the OpenGL 1.2 specification [GLspec], which involves computing the lengths of texture gradient vectors over the entire area covered by the polygon. The specification states that this method is often impractical, and proceeds to state rules defining acceptable approximations.

We use an approximation that involves measuring the output triangle's area in screen pixels. We extrapolate this area measurement, based on the triangle's area in texture space, to compute the necessary resolution of the triangle's texture. This method is similar to the technique described in [Phototextured]. It ignores the effect of perspective distortion, but in terrains tessellated to a suitable level of detail, this effect is minor, and the approximation is quite good.

3.1 Defining our resolution metric

We assume that all textures are square. We define the function $Texture_Area(n) = 2^{2n}$, where the parameter n is a real number that we refer to as a *resolution level*. This function describes the number of pixels contained by a square texture that is 2^n pixels on its side. When the parameter n coincides with an integer, the result represents a square texture that is an integer power of two in width. Some graphics hardware requires power-of-two textures, so it makes sense to restrict our output to these values. Even when hardware does not constrain our texture widths, power-of-two textures provide a good progression in the tradeoff between storage space and image resolution. In cases where mipmapping [Mipmap] is used, these powers of 2 correspond to mipmap levels of the texture.

Suppose we know the number of pixels a polygon occupies when projected to the screen, represented by the real number k . Suppose also that, in texture space, the polygon's area is $1/m$, where an area of 1 would indicate that the polygon completely covers the texture. We can use the inverse function $Texture_Area^{-1}(r) = \frac{1}{2} \log_2(r)$ to find the necessary resolution level. We use $r = mk$ as our input value. That is, r is an extrapolated version of the screenspace area to account for the fact that the triangle covers only part of the texture.

$Texture_Area^{-1}$ will yield us a resolution level as a real number. Taking the ceiling of this result provides us with an integer resolution level that we can use to generate a texture.

This technique ignores perspective, so the results are not conservative; it can produce answers that are slightly too low. However, in systems with high polygon densities, the effect of perspective will be small. Because the algorithm uses a binary triangle tree, the polygons in consideration are all right isosceles; thus we will encounter no triangles that are long and thin, the kind that usually cause perspective-related problems. Also because the system will assign one texture to many polygons, it is likely that if a polygon produces an answer that is too low, some other polygon using the same texture will request a higher resolution level anyway. Finally, we can scale all our triangle area measurements by a constant before pushing them through $Texture_Area^{-1}$, thus giving ourselves a bit of room for error. (This correction would be incorporated into the scaling factor introduced at the end of this section).

We show an example in figure 3: we draw a triangle that covers half of its texture. Its screen-space area is 515.42. The necessary resolution of the texture is $\lceil \frac{1}{2} \log_2(2 \cdot 515.42) \rceil$, or 6. We generate a texture that is 64x64 pixels to use for that polygon.

So far we've assumed that we want our polygons to be textured at a ratio of 1 texture pixel per screen pixel. We may instead wish to provide a user-controlled texture quality parameter, or we may wish to bias our texture resolutions upward to provide a safety margin. If p represents the ratio of texture to screen pixels (measured along one dimension), then we can multiply the screen-space area measurement by p^2 . Thus our equations become:

$$\begin{aligned} Texture_Area(n) &= p^{-2} 2^{2n} \\ Texture_Area^{-1}(k) &= \frac{1}{2} \log_2(p^2 k) \end{aligned}$$

Since p^2 and p^{-2} are constants (at least over the duration of one frame or batch of polygons), the inclusion of these scaling factors is not expensive.

We find the area k of the triangle by taking half the 2D perp-dot product [Hill] of two of the triangle's edges. Let these edges be the vectors \mathbf{s}_1 and \mathbf{s}_2 ; the area is $k = \frac{1}{2}(s_{1x} s_{2y} - s_{2x} s_{1y})$. This

scaling factor $\frac{1}{2}$ can be pre-multiplied with the previously mentioned p^2 factor to save CPU cycles.

3.2 Dealing with the hierarchies of the binary triangle tree and the texture quadtree

Suppose first that we want to render all polygons using a single texture that covers the entire terrain, that is, using the root of the texture quadtree. It is evident that, if l is the level of a triangle in the binary triangle tree, we must multiply its texture-space area by 2^{l+1} to find the area of the full terrain square (each root triangle is $\frac{1}{2}$ the area of the terrain, and each child triangle is $\frac{1}{2}$ the area of its parent). Note that, because triangles' world space z-coordinates do not affect texture coordinates, we are essentially treating the binary triangle tree as an arrangement of 2D triangles.

Usually the resolution level required for the root texture will be prohibitively high. So we descend the quadtree, to the child into which the texture fits. We now multiply the triangle's texture-space area by 2^l to extrapolate, then recompute $Texture_Area^{-1}$. Because the result will be $\frac{1}{4}$ the previous calculation, the newly computed resolution level will be lower. We repeatedly descend the quadtree until the resolution level becomes feasibly low.

We must stop descending the quadtree if there is no child texture into which the current triangle fits. Because the quadtree and binary triangle tree are nicely aligned to each other, this will only occur in genuine circumstances, when no smaller texture would fit across the triangle. That is, no inefficiency arises from inconvenient positioning of the quadtree's partitions.

The alignment between the trees also allows us to quickly compute which quadtree node to descend into (assuming the children are large enough to hold the triangle). Any point interior to the triangle, not including its edges, can be quickly tested against the division lines of a quadtree node, and the entire triangle fits into the child that contains the test point. We cannot use the triangle's edges because one of them may coincide with an edge of the quadtree, yielding an inconclusive decision. In our implementation we use the triangle's barycenter as the decision point. (Figure 4).

Here is the pseudocode for finding a texture's resolution level:

```

Input values are  $v_0, v_1, v_2$ , and area  $k$ 

b := Barycenter( $v_0, v_1, v_2$ )
t := root of texture quadtree
l := level of triangle in binary triangle tree
done := false

while done = false
  q := level of quadtree node(t)
  s :=  $\frac{1}{2}\log_2(p^2k \cdot 2^{l+q+1})$ 
   $\alpha$  := ceil(s)
  if ShouldAccept( $\alpha$ ) then done = true
  else if (l - q = 0) then done = true
  else t := find child(t, b)

Ensure_Texture_Is_Generated(t,  $\alpha$ )

```

3.3 Stopping Criteria

In the above pseudocode we left the stopping criteria vague, relegating such a choice to the function `Should_Accept()`. One possible stopping criterion is to choose a resolution level β representing a maximum texture size, and accept α if $\alpha \leq \beta$. This will assign a triangle to the largest texture that can provide sufficient resolution. This tends to maximize the number of triangles assigned to each texture (looked at another way, it minimizes the overall number of textures) which is good when texture context switches are very expensive.

One may formulate more elaborate stopping criteria, such as accepting α for large textures only if those textures have already been generated at a detail level of at least α (this helps reduce the number of textures generated as the viewpoint moves). In our implementation we cause triangles to gravitate toward the top or bottom of the quadtree; they do not like to stay in the middle levels. Triangles near the top of the quadtree (representing the whole world) or near the bottom (representing individual terrain tiles) have been precomputed. Thus our bias in the stopping criterion causes the dynamic texture generator to engage less frequently, which results in smoother operation. Such heuristics are of course application dependent.

4.1 Exploitation of frame coherence

It would be computationally expensive to perform the pseudocode of section 3 for each polygon in every rendered frame. We can avoid this by exploiting resolution coherence between frames: a polygon rendered at a certain texture resolution during one frame will likely be near that resolution in the next frame.

Many binary triangle tree implementations maintain a persistent tree between frames, updated incrementally. We assume such a tree is available for storing persistent resolution information.

After a call to the pseudocode of section 3, we store on the triangle's node the integer result α , as well as the two real numbers $k_1 = \text{Texture_Area}(\alpha)$ and $k_0 = \text{Texture_Area}(\alpha-1)$. During subsequent frames, we measure the triangle's area k . If $k_0 \leq k \leq k_1$, our previous answer α is still valid (that is, it's still true that $\text{Texture_Area}^{-1}(k) = \alpha$), so we proceed without calling the resolution computation code. If instead k crosses one of those boundaries, we compute the new value of α . Note that $k_0 = \frac{1}{4} k_1$ and that k_1 is a simple function of α so if storage space is crucial we can economize.

For improved efficiency we can combine this boundary check with the rendering pipeline's backface-culling test. We backface-cull the triangle if its area $k \leq 0$, but since $k_0 \geq 0$ there is a dependency between the tests that we can exploit to reduce the number of comparisons in the general case. We here present pseudocode for two combinations of resolution testing and backface-culling, the first done in a straightforward manner, the second done in an optimized manner:

Straightforward:

```

if  $k < 0$  then cull polygon;
else if  $k < k_0$  or  $k > k_1$  then recompute texture resolution.

```

Optimized:

```

if  $k < k_0$ 
    if  $k < 0$  then cull polygon
    else recompute  $\alpha$ 
else if  $k > k_1$  then recompute  $\alpha$ 

```

The optimized formulation requires 2 comparisons per polygon, regardless of whether the polygon is culled. The straightforward formulation requires 1 or 3 comparisons, depending on whether a polygon is culled. If fewer than half the terrain's polygons that pass through this code are backface-culled, the straightforward implementation averages more than 2 comparisons per polygon. Because in terrain rendering the viewpoint is usually above the majority of polygons, and those polygons face upward, significantly fewer than half the polygons are backface-culled. Thus this optimization makes sense.

4.2 Exploitation of a simple recurrence relation

In the pseudocode of section 3 we perform some slow mathematical operations inside a loop. Most of these operations are redundant and can be lifted out.

We note that the quantity $s = p^2 k \cdot 2^{l-q+1}$ changes only insofar as q is incremented during each iteration. Thus the quantity changes by a factor of $1/4$ per iteration; thus $\frac{1}{2} \log_2(s)$ decreases by 1 with each iteration. Therefore, $\lceil \frac{1}{2} \log_2(s) \rceil$ also decreases by 1. The result is that we can compute α at the beginning of the code, then decrement it within the loop. The new pseudocode:

```

b := Barycenter(v0, v1, v2)
t := root of texture quadtree
l := level of triangle in binary triangle tree
q := 0
 $\alpha$  :=  $\lceil \frac{1}{2} \log_2(p^2 k \cdot 2^{l+1}) \rceil$  ;  $q$  is known to be 0 at this point
done := false

while done = false
    if Should_Accept( $\alpha$ ) then done = true
    else if ( $l - q = 0$ ) then done = true
    else
        t := find child(t, b)
         $\alpha$  :=  $\alpha - 1$ 
        q :=  $q + 1$ 

Ensure_Texture_Is_Generated(t,  $\alpha$ )

```

If our stopping criteria are simple enough we can continue exploiting this relation as the triangle is rendered over the course of many frames. When the triangle's area k falls below its lower bound $\text{Texture_Area}^{-1}(\alpha-1)$, we simply decrement α and potentially climb up the quadtree, starting at the triangle's current position, using a very fast routine. When the area rises above $\text{Texture_Area}(\alpha)$, we increment α and potentially descend the quadtree.

We can quickly update the boundary values using a scaling factor, rather than using Texture_Area . The pseudocode for an incremental update:

```

if  $k < 0$  then cull polygon;
else if ( $k < k_0$ )
     $\alpha := \alpha - 1$ 
     $t := \text{Fast\_Reduce\_Resolution}(t, \alpha)$ 
     $k_1 := k_0$ 
     $k_0 := \frac{1}{4}k_0$ 
else if ( $k > k_1$ )
     $\alpha := \alpha + 1$ 
     $t := \text{Fast\_Increase\_Resolution}(t, \alpha)$ 
     $k_0 := k_1$ 
     $k_1 := 4k_1$ 

```

The functions *Fast_Reduce_Resolution* and *Fast_Increase_Resolution* will navigate the quadtree in an implementation-dependant way, depending on the stopping criteria chosen. These functions can be made nearly trivial; thus the only time we evaluate slow mathematical expressions is when a new triangle is rendered for the first time. (Though because LOD operations are common, new triangles may appear with considerable frequency.)

4.3 Elimination of the slow math

The pseudocode of 4.2 still performs some expensive arithmetic. In cases when the viewpoint is moving quickly, and thus new polygons are rapidly introduced by LOD operations, these computations may consume more CPU cycles than we would like. We will now eliminate this problem.

Letting $a = p^2k$, our task is to compute $\alpha = \lceil \frac{1}{2} \log_2(a \cdot 2^{l+1}) \rceil$ as quickly as possible.

We first simplify the log expression by pulling out the exponent:

$$\alpha = \lceil \frac{1}{2}(l+1 + \log_2 a) \rceil$$

Now we introduce the variables m and x , where $a = m \cdot 2^x$, $m \in \mathbf{R}$, $x \in \mathbf{Z}$, $1 \leq m < 2$. The motivation for this is that m and x correspond to the mantissa and exponent of a floating-point number stored according to the IEEE-754 specification. Such m and x will always exist if $a > 0$. But if $a \leq 0$, the triangle will have been backface-culled so it could not reach this code. Introducing m and x gives us:

$$\alpha = \lceil \frac{1}{2}(l+1 + \log_2 m \cdot 2^x) \rceil$$

$$\alpha = \lceil \frac{1}{2}(l+x+1 + \log_2 m) \rceil$$

To simplify further, we break our analysis into two cases, depending on the sign of $(l+x+1)$. If $(l+x+1)$ is even,

$$\alpha = \frac{1}{2}(l+x+1) + \lceil \frac{1}{2} \log_2 m \rceil$$

Since $1 \leq m < 2$, $0 \leq \log_2 m < 1$. Thus $\lceil \frac{1}{2} \log_2 m \rceil = 1$ for all m except ($m=1$). We choose to treat the case ($m=1$) as ($m=1+\epsilon$). This introduces negligible inefficiency into the system (a very small

percentage of the output polygons are treated as though their areas were infinitesimally larger than they really are). Thus

$$\begin{aligned}\lceil \frac{1}{2} \log_2 m \rceil &= 1 \\ \alpha &= \frac{1}{2} (l+x+1) + 1 \\ \alpha &= \frac{1}{2} (l+x+3).\end{aligned}$$

Now, returning to our bifurcation of the problem, we consider the case where $(l+x+1)$ is odd. Again we start from

$$\alpha = \lceil \frac{1}{2} (l+x+1 + \log_2 m) \rceil$$

If $(l+x+1)$ is odd then $(l+x)$ is even, so:

$$\begin{aligned}\alpha &= \frac{1}{2} (l+x) + \lceil \frac{1}{2} + \frac{1}{2} \log_2 m \rceil \\ \alpha &= \frac{1}{2} (l+x) + 1 \\ \alpha &= \frac{1}{2} (l+x+2)\end{aligned}$$

Using a conditional branch we could choose which of these solutions for α to evaluate based on the sign of $(l+x+1)$. But we wish also to eliminate this conditional.

Let us define the operator \gg so that $(a \gg b) = \lfloor a \cdot 2^{-b} \rfloor$. This operator represents the bit-shifting instruction common to CPUs and provided by programming languages like C++.

If $(l+x+1)$ is even, $(l+x+3)$ is even, so:

$$(l+x+3) \gg 1 = \lfloor (l+x+3) \cdot 2^{-1} \rfloor = \frac{1}{2} (l+x+3) = \alpha$$

If $(l+x+1)$ is odd, $(l+x+2)$ is even, so:

$$(l+x+3) \gg 1 = \lfloor ((l+x+2)+1) \cdot 2^{-1} \rfloor = \frac{1}{2} (l+x+2) + \lfloor \frac{1}{2} \rfloor = \frac{1}{2} (l+x+2) = \alpha$$

Therefore in all cases, $(l+x+3) \gg 1 = \alpha$. The programmer way of looking at this is that when $(l+x+2)$ is even, we are free to add an extra 1 to make it $(l+x+3)$; the extra 1 will be shifted off the end of the word anyway.

The integer x can be extracted from a with a bit shifting and masking operation as described in [Hecker]. With this method we also need to add an offset value corresponding to the IEEE-754 exponent bias; since this offset is a constant it can be combined with the addition of the 3, so that we compute $(l+x_{raw} + (bias+3)) \gg 1$.

Thus we achieve the revised pseudocode:

```

b := Barycenter(v0, v1, v2)
t := root of texture quadtree
l := level of triangle in binary triangle tree

x := extract exponent from p2k
α := (l+x+3) >> 1
αdone = α - 1
    
```

```
done := false
while done = false
  if Should_Accept( $\alpha$ ) then done = true
  else if ( $\alpha = \alpha_{done}$ ) then done = true
  else
    t := find child(t, b)
     $\alpha := \alpha - 1$ 

Ensure_Texture_Is_Generated(t,  $\alpha$ )
```

Now that we have reduced the loop's iteration step to a decrement of α , it becomes apparent that with proper data structures and a simple enough stopping criterion, we could jump directly to the level in the quadtree indicated by the first computation of α . For example suppose the stopping criterion is the example given in 3.3, to accept the first α such that $\alpha \leq \beta$:

```
function Should_Accept( $\alpha$ )
  if ( $\alpha \leq \beta$ ) then return true
  else return false
```

We could speed up the quadtree traversal, changing it to a constant-time operation: starting at the root we simply jump downward by $(\alpha - \beta)$ levels, then figure out which child of that level the barycenter lies in. Whether this is a good idea is implementation dependent.

5. Implementation details

The area computations we perform per-triangle depend on the triangle being right isosceles. If a triangle were to be clipped early in the geometry pipeline, this system would generate incorrect results. Thus we assume the resolution measurement occurs before clipping.

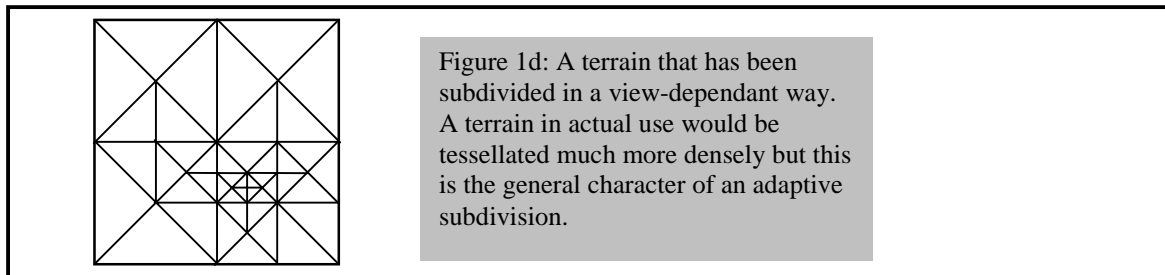
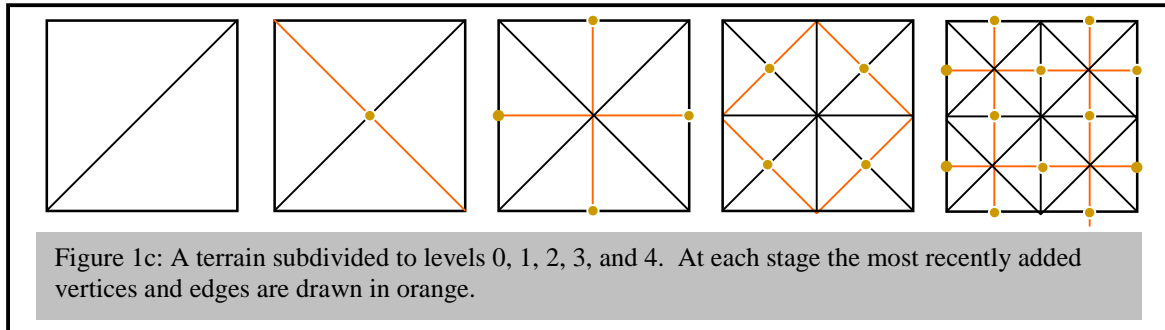
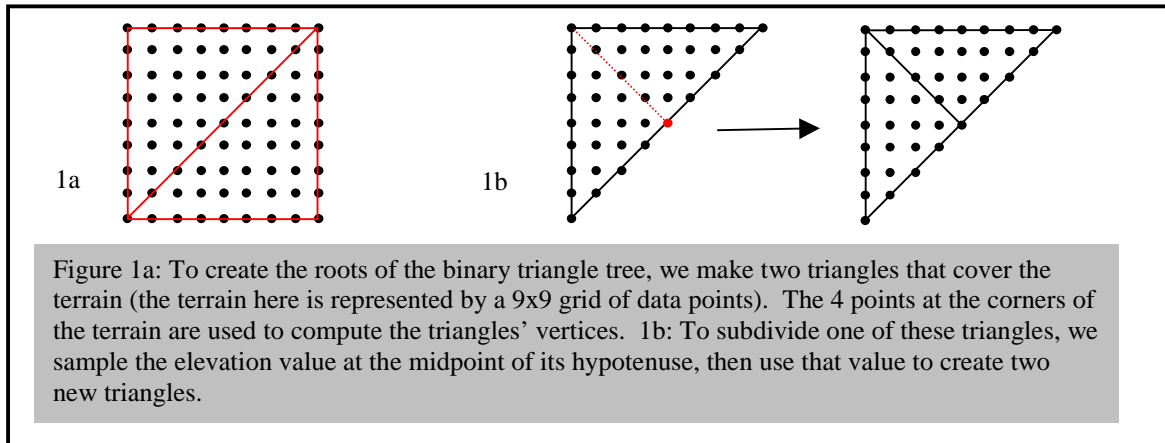
This creates one further issue, which is that triangles with vertices behind the viewpoint, which have not yet been clipped, would generate messy and incorrect results. Thus we perform steps in the following order: 1) Compute which planes of the frustum a polygon will be clipped against. 2) If the polygon might be clipped by the near clipping plane, set its α to an artificially maximum value; otherwise compute α according to our algorithm. 3) Clip the polygon. 4) Render the polygon.

The triangles close to the near clipping plane will also be the ones upon which the perspective effect is most severe. Thus, forcing these triangles' textures to maximum resolution has the nice side effect of eliminating most of the cases where the algorithm might produce an answer that is too low (due to its lack of perspective handling as discussed in 3.1).

Conclusions and Future Work

Using this technique we have successfully rendered terrains with large textures at interactive rates, without requiring a far clipping plane or fog wall to obscure distant terrain features. Specifically, with 8000 visible polygons and a texture map that is 2^{36} texels large, we achieve 40 frames per second on a 450 MHz AMD Athlon processor and an Nvidia GeForce 256 graphics card. The rest of our rendering pipeline is not peephole optimized so we feel this number could be increased considerably.

One limitation of our technique is that it must know the screen-space coordinates of terrain polygons in order to compute texture resolutions. Thus the terrain vertices must be transformed into screen-space by the software. However, hardware-accelerated vertex transformation is now coming into existence on consumer graphics hardware. Our algorithm cannot currently take advantage of such acceleration. Thus an interesting bit of future work would be an algorithm similar to the one presented here that does not require transformed coordinates.



References

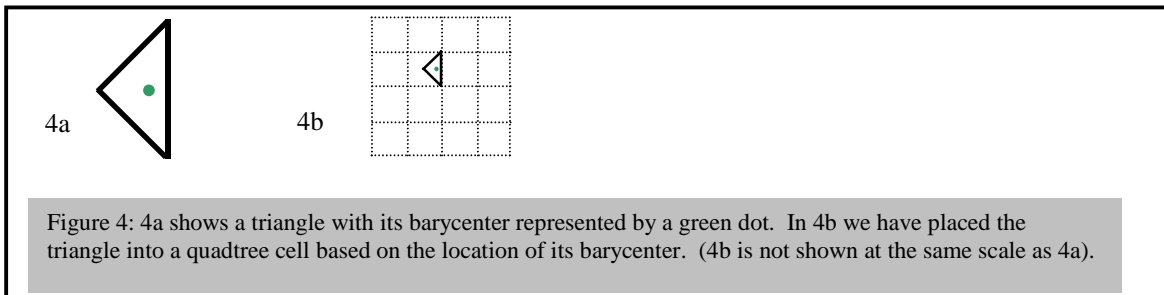
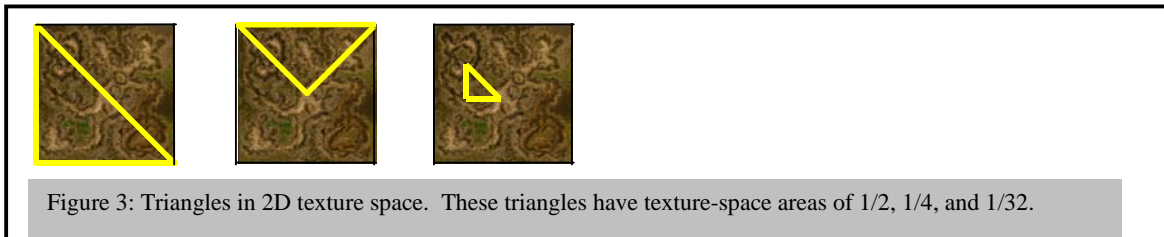
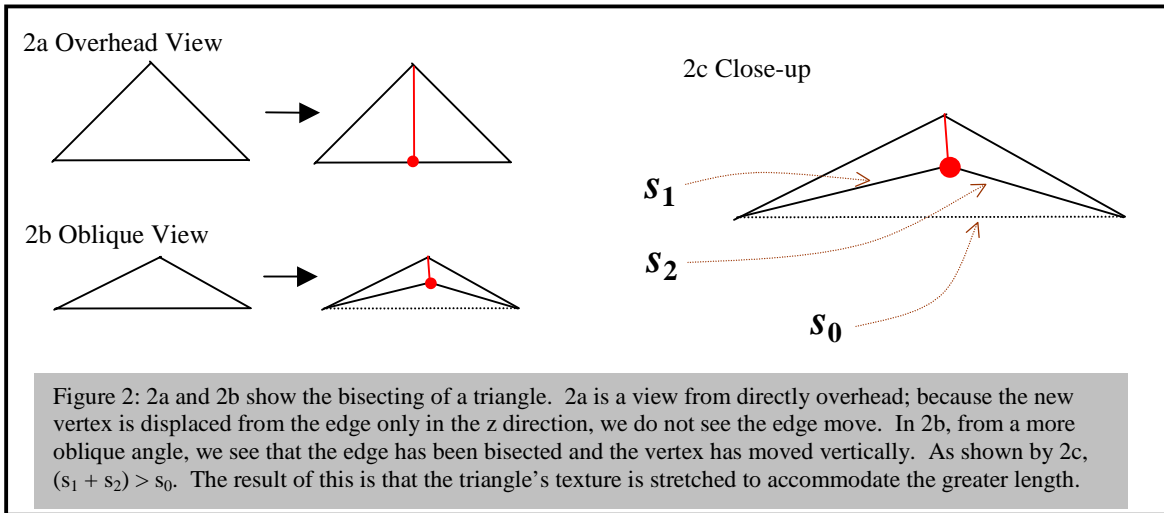
- [Clipmap] "The Clipmap: a Virtual Mipmap", Christopher C. Tanner, Christopher Midgal, and Michael T. Jones, *Proceedings of SIGGRAPH 1998*.
- [GLspec] "OpenGL 1.2 specification", Mark Segal and Kurt Akeley, available at www.opengl.org/Documentation/specs.html. Section 3.8.5.
- [Hecker] "Let's get to the (floating) point", Chris Hecker, *Game Developer* magazine, February / March 1996.

[Lindstrom] “Real-Time, Continuous Level of Detail Rendering of Height Fields,” Peter Lindstrom, David Koller et al, *Proceedings of SIGGRAPH 1996*.

[Phototextured] “Level-of-Detail Management for Real-Time Rendering of Phototextured Terrain”, Peter Lindstrom, David Koller et al, Georgia Tech Technical report GIT-GVU-95-06. January 1995.

[ROAM] “ROAMing terrain: Real-time optimally adapting meshes”, Mark Duchaineau, Murray Wolinsky et al, Proceedings of the ACM Symposium on Volume Visualization 1997.

[Hill] “The Pleasures of ‘Perp Dot’ products”, F.S. Hill, Jr., *Graphics Gems IV*, Paul S. Heckbert, ed. AP Professional, 1994.



Part II:

Lecture Notes for Jonathan Blow's half of "Two Advanced Terrain Rendering Systems"

Previous systems and current design goals

The current terrain system we're working on is our third generation system. Our first generation was created back in 1996 and used Lindstrom's algorithm to manage levels of detail. Our second generation, created in 1998, used a modified version of Lindstrom's algorithm that used a lot more persistent state. However, for our third generation engine, we wanted to support levels of detail that are much higher than the previous systems, so we needed to come up with something new.

Our design goals were: the ability to move smoothly around a terrain that is 2^{15} samples on each side (a total of 2^{31} polygons), and that was textured with a very detailed texture (2^{36} texels), with an unlimited view distance. That is, there would be no fog to obscure the user's view, and the user can view the terrain from any position at any angle. Since we do not want to use large amounts of disk space to store the texture, we assume that it will be cleverly synthesized from artist-controlled components and not stored on the disk as a huge bitmap (though we could do that if we chose). We also wanted our terrains to look more organic than in previous systems, so we decided to create them using Bezier patches with overlaid displacement maps.

Lindstrom versus ROAM: which should we use?

For our initial implementation of the new engine we chose the ROAM algorithm. ROAM has the nice property of being a top-down algorithm, rather than a bottom-up one like Lindstrom. In order to compute LOD for a terrain, ROAM only requires that you can compute bounding volumes around your terrain data; ROAM does not need to know what is inside those volumes. Thus there doesn't have to be a "bottom" to your data, and you don't need to have much instantiated source geometry at all; when it comes time to split a ROAM wedge you can use curve subdivision and fractal embellishment to compute new data points. In contrast, because Lindstrom works bottom-up from a set of known data points, you need to have those points instantiated, then detail reduce them, so that you might eventually expand your terrain tree back to that level of detail.

There is a CPU cost associated with detail reduction / enhancement (deciding when you need more or less detail in a certain area), and there is also a memory cost (storing whatever data you need for the algorithm to do its stuff). The CPU and memory cost of the Lindstrom algorithm are on the order of the total amount of terrain data that you have (because Lindstrom is bottom-up). The CPU and memory cost of ROAM are on the order of the amount of terrain data that is currently instantiated to the viewer, which is generally much less than the entire terrain (that's what LOD is for!). Lindstrom's algorithm employs block-based terrain subdivision to cut down on CPU and memory costs, but upon initial inspection of the two algorithms it would seem that ROAM is much more efficient.

We began our third-generation terrain system by implementing ROAM and found that it has its own set of performance problems. The authors of the ROAM paper advocate computing a priority value for each instantiated terrain wedge; the higher the priority, the sooner that wedge would be subdivided. They then place the wedges into priority queues, with priorities updated based on the motion of the viewpoint.

We found this system to be very inefficient at high detail levels. In a well-tesselated high-detail terrain, a large percentage of the terrain polygons will be close to the viewpoint and, correspondingly, small. Thus their size and distance from the viewpoint are not very much larger than the distance one might expect the viewpoint to move in a short period of time (e.g. 200 milliseconds). Because the wedges are weighted only by a single priority, the algorithm is unable to differentiate between wedges that are becoming closer to the viewpoint and wedges that are receding. In other words, the priority queue assumes that all wedges are rushing toward the viewpoint at an equal speed, and so it spends a lot of time re-evaluating wedges prematurely. In a high-detail landscape the number of wedges re-visited in this way will be a staggeringly large percentage of the current instantiated detail.

We spent a month trying to create optimized priority queue implementations, which helped, but in the end we decided that the method just wasn't workable. We needed to come up with our own algorithm that was like ROAM, in that it proceeded top-down, but that used different criteria for splitting and merging wedges. Before too long we came up with one, and were happy to see that it was much faster than ROAM for high detail levels.

An improved algorithm

By looking at the behavior of ROAM's priority queues it became clear that our system needed to be more mindful of the spatial relationships between the terrain wedges. We noted that both Lindstrom and ROAM use effectively similar functions for prioritizing their splits and merges. They both employ functions that look at the coordinates of some point in view space (x, y, z) , and some height value 'h', and produce a scalar 'p' that determines the priority of that point with respect to splitting or merging. That is they use functions $p = f(x, y, z, h)$. When we consider that 'h' is constant for any particular vertex or wedge, and that these algorithms are mainly trying to deal with the motion of the viewpoint with respect to terrain vertices, then we see that $p = f(x, y, z)$, where 'f' is now some function that is customized for each vertex or wedge to incorporate 'h', and (x, y, z) is the position of the viewpoint.

In other words, both Lindstrom and ROAM take information that is three-dimensional $(x, y, \text{ and } z)$, and compress that down into a one-dimensional result (p) which they use to organize the terrain data. 'f' is a transformation from a 3-dimensional space into a 1-dimensional space; thus it introduces a whole lot of aliasing. It is this aliasing that makes the algorithms computationally inefficient. Lindstrom fights back against the aliasing by subdividing the terrain into blocks, so that only points within each particular block are aliased to each other. But this only mitigates the problem; it does not solve it.

The real solution is to actually use the 3 dimensions of source data to perform LOD computations, rather than the aliased 1-dimensional projection. This is what our algorithm does. Rather than sorting vertices by the priority value 'p', it maintains a data structure of the isosurfaces consisting of all (x, y, z) for which $f(x, y, z) = p$. These isosurfaces are three-dimensional shapes. For computational simplicity, as well as to provide some desirable behavioral properties to the rest of the game engine, we choose our functions 'f' so that the isosurfaces are spheres.

Thus, when there are a series of terrain wedges in space that we might wish to split or merge, we represent each wedge by a sphere in 3D space. When the viewpoint crosses into that sphere, the wedge should be split. When the viewpoint leaves a sphere, it is time for that wedge to be merged.

This algorithm would be slow if we simply maintained a linear array of spheres that are tested against the viewpoint each frame. Instead, we create a hierarchy of spheres. If sphere B is contained inside sphere A, then we know that the viewpoint cannot cross into B unless it first crosses into A. We place A and B into a sphere tree where B is a child of A. Wedge B will never be looked at again until A is split. Note that this is a clear advantage over ROAM, wherein wedge B would always be re-evaluated eventually.

We say that there is an *emergent hierarchy* of spheres: wedges B and A do not need to be closely related in the binary triangle tree. All that is necessary is that one sphere happens to contain the other when we compute our isosurfaces. Exploiting this hierarchy trims down the width of the tree drastically. But it is still pretty wide; there tend to be hundreds of spheres at the top levels of the tree (or thousands, depending on instantiated detail level). To further improve speed, we perform clustering on any level of the tree that becomes too populated: we choose a group of isosurfaces that are close together and create a new, "fake" isosurface that does not correspond to a wedge in the terrain tree; it just serves to group those surfaces together and improve runtime efficiency.

We find this system to be incredibly efficient, much more so than we could have hoped. The amount of CPU time spent determining splits and merges for high-detail terrain, with a quickly moving viewpoint, is negligible for all data sets we have tried so far. On these same data sets, our ROAM implementation causes the frame rate to stutter as it re-evaluates buckets full of vertices.

Using this algorithm for texture detail determination

Part I of this paper, “Fast Reduction of Texture Memory Usage for Terrain Rendering”, describes a system we had devised for computing the required amount of texture detail to display a scene. In the conclusion of that discussion we state that the system does not cooperate well with hardware-accelerated vertex transformation because it requires projected coordinates of the terrain geometry.

For a while we had been thinking of better systems for solving this problem. Our terrain isosurface system provided a great solution. For each wedge we add isosurfaces that represent the distances at which the wedge requires a certain mipmap level of its texture. When the viewpoint crosses into one of these surfaces, we increase that mipmap level and add a new isosurface inside it, that represents the next higher mipmap level. When the viewpoint crosses out, we do the opposite. At any time each wedge has two instantiated isosurfaces, one for increasing its mipmap level and one for decreasing it.

We find that this system runs just as fast as our polygonal detail isosurface system, and that it removes the requirement that we transform the terrain geometry every frame.

High-quality shading

Now that we had the geometry and texture problems for our terrain system solved, we began thinking about lighting. For shading the terrain polygons we used illumination maps. These illumination maps are computed during a preprocess, so that expensive global illumination computations can be performed. The end result is that we have self-shadowing terrain (with respect to global light sources like the sun and moon).

We compute 64 of these illumination maps, evenly distributed throughout the length of a day. We animate the shading on the terrain by cycling between them, interpolating to provide smoothness. Thus you can see the shadow of a peak crawl across the terrain as the sun sets.

These illumination maps would normally require a lot of space. We compress them heavily (using wavelet compression). Because the 64 textures are an animated sequence, they exhibit continuity through time as well as through the ‘s’ and ‘t’ dimensions of the texture. Thus we can compress the whole set as one three-dimensional texture, and achieve very high compression ratios. Also, because the illumination maps are generally expected to be blurry, we can overcompress them without worrying so much about the usual issues of wavelet overcompression (blurring of sharp edges, ringing).

Volumetric light

We decided that it was pretty weird for people to be walking around on a terrain like this, with moving shadows, and for the people themselves to not be shadowed. So we wanted to come up with a volumetric lighting solution.

We saw that, with respect to a distant overhead light source like the sun, then the shadow volume of a height-mapped terrain is itself a height map (to a good approximation; because of the regular spacing of the samples, the shadow volumes cannot be made to intersect the ground at arbitrary positions). We compute the shadow volumes as a preprocess, then approximate them with height maps. We then use our existing terrain LOD system to manage these height maps in the game. We generate 64 of these terrains and morph between them, as with the illumination maps.

When rendering an entity, we test it against this (invisible) shadow terrain. If it is entirely below the shadow terrain, it is drawn shaded. If it is entirely above, it is drawn fully lit. If the entity crosses the surface of the shadow terrain, we clip the entity into two pieces, rendering one piece in light and the other in shade.

Current and future work

There are some distinct similarities between wavelet compression, or the classical Laplacian image pyramid, and the ROAM-like system of hierarchies of bounding volumes. Each level of a binary triangle tree can be viewed as a low-frequency filtered version of the level below it. This is not quite true because the computations usually used to create these bounding wedges do not correspond directly to the mathematics of filtering. But by changing our view of what these wedges are supposed to be, and by modifying the way in which our engine handles the wedges, we can solidify the correspondence.

Thus any algorithm that uses a top-down binary triangle tree (like ROAM, or our new algorithm), can be modified philosophically and algorithmically so that it begins at a low-frequency version of the terrain and, when wedges are to be split, reaches into the higher frequencies. This is isomorphic to an incremental wavelet decompression, or the selective expansion of a Laplacian pyramid. These sparse image representations are often saved into a packed structure using zerotrees. There is a close correspondence between the structure of a zerotree and the structure of a binary triangle tree, which makes it similarly easy to navigate a zerotree incrementally.

We expect that future versions of our terrain algorithm will operate directly on compressed data, with no reduction in their running speed.