Dynamic Level of Detail Representation of Interactive 3D Worlds

Computer Science 95.495B Honours Project

By: ***Shahzad Malik*** *(219762)*
Supervisor: ***Dr. Wilf R. LaLonde***
Carleton University
Ottawa, Ontario, Canada
April 10, 2000

Abstract

Due to the wide spectrum of consumer-level 3D hardware, scalability is a major concern when representing interactive 3D worlds. In order to account for major performance differences in 3D hardware, dynamic level of detail techniques can be employed by the geometry such that the 3D world can be played back at interactive frame rates regardless of the underlying hardware. This report presents a simple and elegant approach to achieving dynamic detail levels on regularly spaced grids of elevation data, with a focus on visualizing 3D terrain. Additionally, the report describes a generalization to the terrain grids, whereby multiple arbitrarily oriented "patches" of elevation information can be seamlessly attached together to create a fully scalable representation of more than just terrain, such as tunnels and caves.

Acknowledgements

I would like to sincerely thank Dr. Wilf LaLonde for his valuable suggestions and guidance with this project. I would also like to thank the faculty, staff, and students in the Computer Science department at Carleton University for all their help during my undergraduate studies. Finally, I'd like to thank all my family and friends, especially the ones who have nothing to do with computers, for their understanding and support for these past 4 years.

Table of Contents

1 List of Tables 5

2 List of Figures 5

3 Introduction 6
- 3.1 Conventions Used 6

4 Representing Outdoor Terrain Geometry 7
- 4.1 Traditional Terrain Methods 7
- 4.2 Traditional Level of Detail 8
- 4.3 Binary Triangle Trees 9
- 4.4 Top-down LOD Algorithm 11
  - 4.4.1 Forced Splitting 12
  - 4.4.2 Terrain Tessellation 15
- 4.5 Texturing 17
  - 4.5.1 Unique Texturing 17
  - 4.5.2 Detail Texturing 18
  - 4.5.3 Texture Blocks and Terrain Blocks 19
- 4.6 Quadtree Storage of Blocks 20
- 4.7 LOD Error Metrics 21
  - 4.7.1 Bounding Spheres for Triangle Prioritization 21
  - 4.7.2 Variance Trees 22
- 4.8 Achieving Target Triangle Counts 25
- 4.9 Handling Dynamic Objects 26
- 4.10 Handling Popping via Vertex Morphing 27
- 4.11 Generating Triangle Strips 29

5 Representing Arbitrary World Geometry 30
- 5.1 Binary Triangle Tree Patches 30
  - 5.1.1 Comparison to Bezier Patches 31
- 5.2 Future Considerations 31
  - 5.2.1 Merging Traditional Indoor Representations 31
  - 5.2.2 Handling Details using Continuous Meshes 32
  - 5.2.3 Increasing Detail at the Texture Level 33
  - 5.2.4 Temporally Coherent Algorithm 33

6 Implementation Results 34
- 6.1 Top-down Algorithm Analysis 34
- 6.2 Summary 35

7 Conclusion 37

8 References 38

9 Appendices 39
- 9.1 Appendix A – Displacement Maps 39

1 List of Tables

Table 1	Basic Attribute Memory Usage	**34**
Table 2	Memory Usage Comparisons	**34**
Table 3	Performance Comparisons	**35**

2 List of Figures

Figure 1	8 x 8 height map	**7**
Figure 2	3D wire-frame terrain	**7**
Figure 3	Sphere tessellation	**8**
Figure 4	6 levels of a Binary Triangle Tree	**9**
Figure 5	T-junction explanation	**10**
Figure 6	Proper terrain grid resolution	**10**
Figure 7	Base triangulation	**11**
Figure 8	Triangle neighbourhood	**12**
Figure 9	Recursive forced splitting	**14**
Figure 10	Unique texturing (Grand Canyon)	**17**
Figure 11	Detail texturing (white noise)	**18**
Figure 12	Texture Blending	**18**
Figure 13	Base triangulation of 64x64 blocks	**19**
Figure 14	Bounding spheres	**21**
Figure 15	Dynamic objects on tessellating terrain	**26**
Figure 16	Vertex popping	**27**
Figure 17	Left-right Binary Triangle Tree traversal	**29**
Figure 18	Triangle strip of four vertices	**29**
Figure 19	Attaching two patches	**30**
Figure 20	Merging traditional 3D structures with patches	**32**
Figure 21	Sample displacement and colour map	**39**
Figure 22	Rendered Quarry scene	**35**
Figure 23	Rendered Grand Canyon scene with wireframe	**36**

3 Introduction

On current consumer-level graphics hardware, rendering large, detailed 3D worlds consisting of complex arbitrary geometry (such as terrain and indoor architecture) typically utilizes all available triangle bandwidth, such that frame rates drop below acceptable levels for real-time playback. One common workaround typically involves using extremely short view distances and fogging techniques to smoothly cull distant triangles from the rendering engine. This works well, since the number of rendered triangles is drastically cut down, but at the expense of reducing the output quality of the virtual world. Another common workaround involves limiting the geometry of the 3D world. Many 3D games on the market today are either indoors only or outdoor only, since there are well-established algorithms to represent these types of worlds on their own. In the case of indoor worlds, Binary Space Partition (BSP) or portal techniques are ideal, since the assumption with indoor worlds is that, typically, the viewer cannot “see” forever. On the other hand, these techniques don’t work well for outdoor worlds, since terrain renderers typically allow you to see to the horizon. In these cases, dedicated level of detail terrain systems can be used, with good results.

This project attempts to allow any arbitrary 3D geometry to be part of a dynamic level of detail 3D engine. Geometric detail will dynamically increase and decrease based on the camera position, providing details in curved or contoured sections while removing redundant triangles in relatively flat or planar regions. While realistic and seamless representations of indoor and outdoor worlds still haven’t been accomplished extremely well, this project makes a novel attempt by generalizing a continuous level of detail terrain system to arbitrarily oriented “patches” or surfaces. The level of detail algorithms will allow for a very scaleable representation of the 3D world, by making use of adjustable error metrics such as strict triangle counts and pixel error thresholds.

3.1 Conventions Used

- Pseudo code is represented using a C-like syntax, mixed with high-level statements in English
- The world coordinate system, when viewed from screen-space, has X positive to the right, Y positive in the up direction, and Z positive into the screen.

4 Representing Outdoor Terrain Geometry

Three dimensional representations of terrain have dozens of educational and commercial uses in industry today. Such things as virtual tourism, travel planning, land planning, weather/environmental analysis, and entertainment all benefit from accurate and realistic depictions of virtual terrain data. As a result, this section will focus on representing terrain at real-time frame rates, by employing a dynamic level of detail algorithm which makes use of the consistency and regularity of sampled height data.

4.1 Traditional Terrain Methods

Terrain information typically consists of a grid of evenly sampled elevation information. Figure 1 shows a height map of 8 x 8 regularly spaced points. The grid can be thought of as a top view of some elevation data, such that each point represents a height value. Additionally, between each 2x2 block of adjacent points there are 2 triangles. The spacing between points is fixed, and can be chosen rather arbitrarily if the elevation data is generated algorithmically. However, in the case of satellite terrain data, sampling is usually defined in the 50m range, such that the 8x8 grid would represent an area of 400m x 400m.

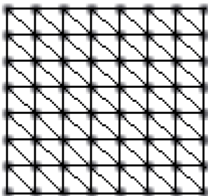


Figure 1

To view the terrain data, the 2 triangles in each 2x2 block can then be rendered in 3D. The vertices for each triangle can be defined by:

v = (x * grid_spacing, h(x, z), z * grid_spacing)

where ***x*** is the horizontal position of a grid point in the 2 dimensional height grid, ***z*** is the vertical position of a grid point in the 2 dimensional height grid, ***grid_spacing*** is the spacing between grid points, and ***h(x,z)*** is the height value at the given x,z coordinates in the height map. Figure 2 shows such a 3 dimensional depiction of a 2 dimensional height map, with a wireframe overlay showing the triangle boundaries.

As can be seen, this simple representation of height data results in realistic 3D visualizations of terrain. However, this methodology doesn't scale very well. Consider a height map consisting of 2500 x 2500 grid points, with a 1m sampling. This would result in a terrain of 2.5km x 2.5km, consisting of 2500 x 2500 = 6,250,000 height values, and 2500 x 2500 x 2 = 12,500,000 triangles.

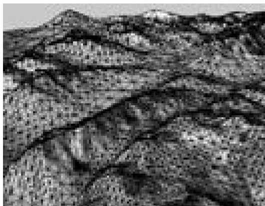


Figure 2

Now consider that a typical outdoor scene with a 2km to 3km visibility range is not uncommon. Rendering the entire 2500 x 2500 grid data in 3D would require almost all 12,500,000 triangles to be rendered. Further, to realistically display an interactive walkthrough of such a scene at 30 frames per second (fps) would require a triangle throughput of 12,500,000 x 30 = 375 million triangles per second!

Considering that even that fastest consumer-level 3D graphics card on the market today has a triangle throughput of only 15 million triangles per second, this method of rendering detailed terrain data is clearly not feasible.

4.2 Traditional Level of Detail (LOD)

The terrain in Figure 2 shows an interesting property of 3D graphics. Consider the traditional 3D to 2D perspective transformation [FOLE1990]:

$$X' = X / Z$$
$$Y' = Y / Z$$

where X, Y, and Z are the coordinates of some point in 3D camera space, and X' and Y' are the associated 2D integer screen-space coordinates. Clearly, as Z increases, the distances between points become less and less meaningful. For example, consider two points with a spacing of 10 meters. Using the above transformation, at a distance of 1m the two points will be 10 pixels apart. However, at a distance of 10 meters, the two points are now 1 pixel apart. As a result, the terrain nearing the horizon in Figure 2 shows many small triangles which are nearing the size of 2 or 3 pixels each, clearly causing the rendering system to waste valuable processing time on irrelevant geometry.

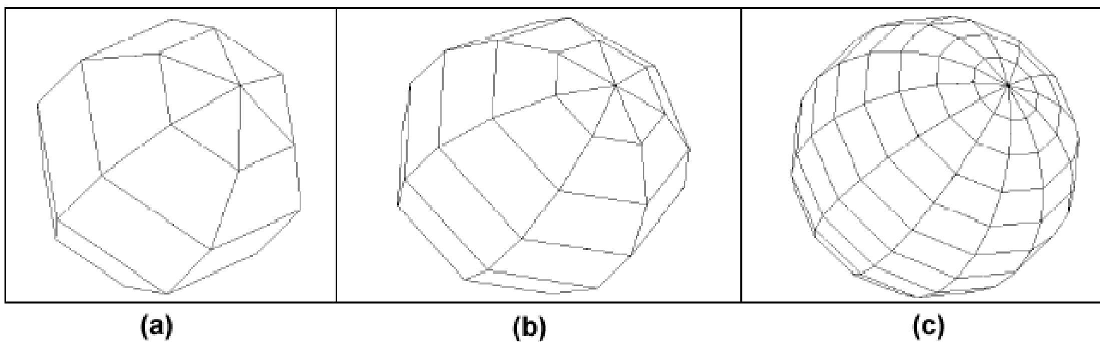


Figure 3 - sphere tessellation

Figure 3 shows three different representations of identical size spheres. Figure 3a consists of 30 polygons, Figure 3b consists of 49 polygons, and Figure 3c consists of 168 polygons. This is the traditional method of performing level of detail on small objects. By creating multiple versions of the same 3D mesh, with each version consisting of less and less polygons, a 3D rendering engine can swap in lower detail meshes as the object recedes into the horizon. The correctness of this method follows directly from the 3D to 2D projection property, where detail becomes less meaningful as objects get further away from the camera [HOPP1996].

4.3 Binary Triangle Trees

Figure 3 shows the traditional way to perform LOD on small objects, but how does this method scale to large scale geometry such as terrain? Creating multiple versions of a terrain is unfeasible since most terrain visualization occurs with the camera at some position within the terrain grid itself; swapping in lower resolution meshes simply wouldn't make sense. Additionally, terrain geometry can vary in more complex ways than simply based on Z distance from the camera, such as localized peaks and valleys. What is needed is some sort of continuous level of detail algorithm which adds and removes detail based on viewpoint and geometric complexity, in real-time. Enter Binary Triangle Trees.

Binary Triangle Trees are an extremely elegant way to represent regularly spaced terrain data with real-time, view-dependent level of detail. They combine the simplicity of a typical Binary Tree (each node having only two descendants) with the 2-dimensional area covering properties of a Quad Tree [MCNA1999a].

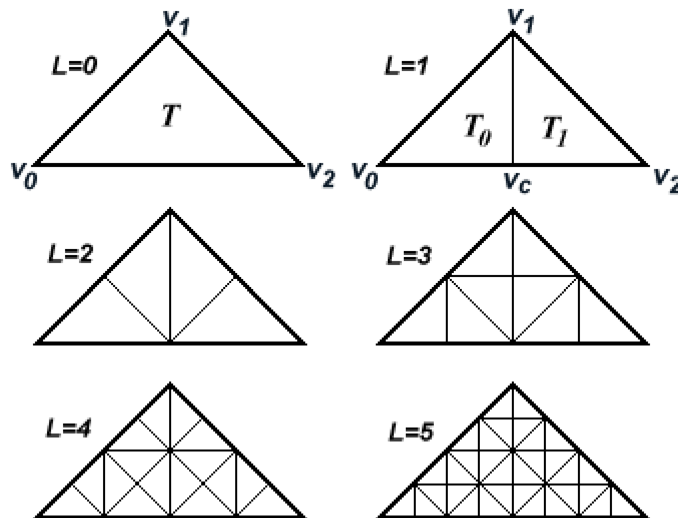


Figure 4 - Levels 0-5 of a binary triangle [DUCH1997]

Figure 4 shows the first six levels of a Binary Triangle Tree. The root triangle $\boldsymbol{T}$ = (v_0, v_1, v_2) is a right isosceles triangle, and is considered the coarsest level, L=0. The root triangle can then be split into two smaller triangles $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$, to form L=1. Splitting is performed on a triangle by creating an edge from v_1 down to v_c, where v_c is the midpoint of the hypotenuse defined by the edge (v_0, v_2). This gives us a left child $\boldsymbol{T_0}$ = (v_0, v_1, v_c) and a right child $\boldsymbol{T_1}$ = (v_c, v_1, v_2). This recursive subdivision can theoretically go on forever, with Figure 5 showing recursion down to level L=5.

So how do Binary Triangle Trees help with tessellating terrain? Consider the following key properties of Binary Triangle Trees so far:

- The right isosceles triangle property holds throughout the recursive splitting, giving the advantage that all areas consist of a 90 degree angle connecting two equal length sides.
- The splitting process seems to guarantee that geometry will never develop “cracks” or T-junctions (see Figure 5).

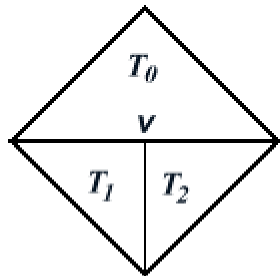


Figure 5 - T-junction

Due to limited floating-point precision, the geometry on the left will result in visual "cracks" at vertex v, since v is part of both triangle T1 and T2, but not triangle T0.

The only major catch with Binary Triangle Trees is the restriction they place on terrain dimensions. Ideally, terrain areas should consist of square samplings of points, in order to guarantee the right-isosceles triangle property. Additionally, since splitting of triangles involves choosing a point at the centre of the hypotenuse, there must be a way to assure that a point actually exists at that centre location *within* the height map. Figure 6a shows an 8x8 height map, with the base triangulation. Now consider splitting either one of the base triangles. Clearly the centre of the hypotenuse in each triangle is NOT a proper point within the height grid, and as such would require the creation of a new point. However, there is a simple way to ensure that all split points (down to the finest detail level) choose a centre point which is already IN the height map: simply restrict the dimensions of terrain grids to (2^n+1) x (2^n+1) points. Figure 6b shows a (2^3+1) x (2^3+1) = 9x9 terrain grid, which guarantees that all splits down to the finest detail level contain a valid centre hypotenuse point. **(Note: by wrapping the 2^n+1th point back to point 1 in the terrain grid, it is possible to have exact power of 2 terrain dimensions, which are desirable for storage and indexing purposes.)**

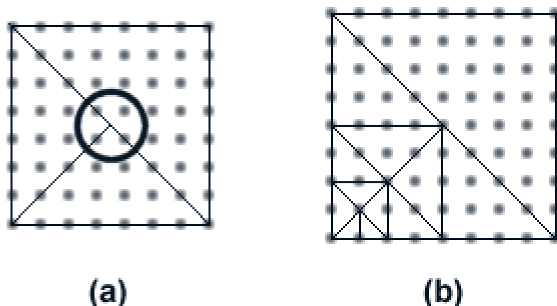


Figure 6 - (a) 8x8 grid doesn't allow proper grid splits. (b) 9x9 grid allows proper splitting down to finest resolution.

4.4 Top-down LOD algorithm
Using these properties, it is now possible to use Binary Triangle Trees for tessellating our regularly spaced height map data to non-regular triangle densities, so that close to the viewer we tessellate to a high density, and far from the viewer we use a very low triangle density.

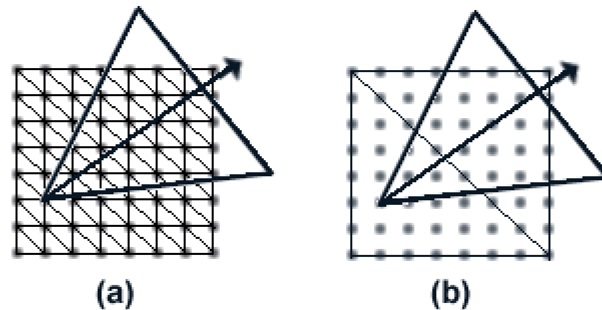


Figure 7 - (a) original height map, with view volume, (b) base triangulation, with view volume

Figure 7a shows a block of terrain at its highest-resolution mesh, with the current camera's view volume facing in the top-right direction. The top-down algorithm will start off by creating a *base triangulation* for the terrain block, as can be seen in Figure 7b, such that there are two triangles, with vertices defined using the 4 corner extents of the grid area. The idea is to use this base triangulation, and progressively add the desired amount of detail, in a top-down manner.

Before the tessellation algorithm can be described in further detail, its important to touch on how the triangle splitting will work.

4.4.1 Forced Splitting

All binary triangle structures will, at a minimum, look similar to the following:

```
struct BTT_Triangle {
   BTT_Triangle *m_pLeftChild;
   BTT_Triangle *m_pRightChild;
   BTT_Triangle *m_pLeftNeighbour;
   BTT_Triangle *m_pRightNeighbour;
   BTT_Triangle *m_pBottomNeighbour;
}
```

Each triangle has links to its left and right children (if they exist) as well as links to its left neighbour, right neighbour, and bottom neighbour (if they exist). The left, right, and bottom neighbours are defined based on viewing the triangle with the hypotenuse facing down, as depicted for triangle **T** in Figure 8a. **T** has left neighbour $\mathbf{T}_L$, right neighbour $\mathbf{T}_R$, and bottom neighbour $\mathbf{T}_B$.

Now in order to split a triangle, its important to maintain a proper Binary Triangle Tree, which will avoid the T-junctions which were discussed earlier. [DUCH1997] mentions that a key fact about Binary Triangle Tree triangulations is that neighbours for a triangle T are either from the same level L as T, or from the next *finer* level L+1 for left and right neighbours, or from the next *coarser* level L-1 for bottom neighbours. As a result, to split triangle T, we first divide it into its left and right children by adding a new vertex (from the height map) along the bottom edge (as in Figure 4). We then check whether T has a bottom neighbour which in turn has T as its bottom neighbour and if so, we recursively force the bottom neighbour to split. We need not worry about the left and right neighbours due to their elegant *finer* level property. Figure 8b shows such a forced split on T.

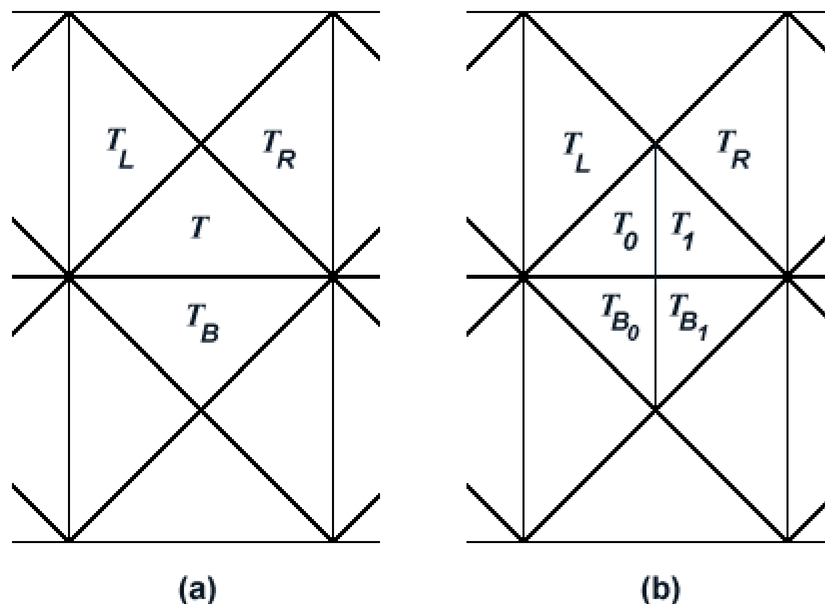


Figure 8 - [DUCH1997] **(a) neighbourhood for triangle T (b) neighbourhood for triangle T after splitting it**

Some pseudo code for our recursive splitting can be stated as follows:

```
void splitTriangle(BTT_Triangle *pTriangle)
{
   if(pTriangle->m_pBottomNeighbour != NULL) {
     if(pTriangle->m_pBottomNeighbour->m_pBottomNeighbour != pTriangle) {
       splitTriangle(pTriangle->m_pBottomNeighbour);
     }

     splitTriangleActual(pTriangle);
     splitTriangleActual(pTriangle->m_pBottomNeighbour);

     pTriangle->m_pLeftChild->m_pRightNeighbour
         = pTriangle->m_pBottomNeighbour->m_pRightChild;
     pTriangle->m_pRightChild->m_pLeftNeighbour
         = pTriangle->m_pBottomNeighbour->m_pLeftChild;
     pTriangle->m_pBottomNeighbour->m_pLeftChild->m_pRightNeighbour
         = pTriangle->m_pRightChild;
     pTriangle->m_pBottomNeighbour->m_pRightChild->m_pLeftNeighbour
         = pTriangle->m_pLeftChild;
   }
   else {
     splitTriangleActual(pTriangle);

     pTriangle->m_pLeftChild->m_pRightNeighbour = NULL;
     pTriangle->m_pRightChild->m_pLeftNeighbour = NULL;
   }
 }

void splitTriangleActual(BTT_Triangle *pTriangle)
{
   BTT_Triangle *pLeftChild, *pRightChild;

   pLeftChild = allocateTriangle();
   pRightChild = allocateTriangle();

   pTriangle->m_pLeftChild = pLeftChild;
   pTriangle->m_pRightChild = pRightChild;
   pTriangle->m_pLeftChild->m_pLeftNeighbour = pTriangle->m_pRightChild;
   pTriangle->m_pRightChild->m_pRightNeighbour = pTriangle->m_pLeftChild;
   pTriangle->m_pLeftChild->m_pBottomNeighbour
      = pTriangle->m_pLeftNeighbour;

   if(pTriangle->m_pLeftNeighbour != NULL) {
     if(pTriangle->m_pLeftNeighbour->m_pBottomNeighbour == pTriangle) {
       pTriangle->m_pLeftNeighbour->m_pBottomNeighbour
            = pTriangle->m_pLeftChild;
     }
     else {
       if(pTriangle->m_pLeftNeighbour->m_pLeftNeighbour == pTriangle) {
         pTriangle->m_pLeftNeighbour->m_pLeftNeighbour
              = pTriangle->m_pLeftChild;
       }
       else {
         pTriangle->m_pLeftNeighbour->m_pRightNeighbour
              = pTriangle->m_pLeftChild;
       }
     }
   }

   pTriangle->m_pRightChild->m_pBottomNeighbour
```

```
        = pTriangle->m_pRightNeighbour;

    if(pTriangle->m_pRightNeighbour != NULL) {
      if(pTriangle->m_pRightNeighbour->m_pBottomNeighbour == pTriangle) {
        pTriangle->m_pRightNeighbour->m_pBottomNeighbour
             = pTriangle->m_pRightChild;
      }
      else {
        if(pTriangle->m_pRightNeighbour->m_pRightNeighbour == pTriangle) {
          pTriangle->m_pRightNeighbour->m_pRightNeighbour
                = pTriangle->m_pRightChild;
        }
        else {
          pTriangle->m_pRightNeighbour->m_pLeftNeighbour
                = pTriangle->m_pRightChild;
        }
      }
    }

    pTriangle->m_pLeftChild->m_pLeftChild = NULL;
    pTriangle->m_pLeftChild->m_pRightChild = NULL;
    pTriangle->m_pRightChild->m_pLeftChild = NULL;
    pTriangle->m_pRightChild->m_pRightChild = NULL;
}
```

Figure 9 shows a more detailed forced split of a triangle, depicting the recursive nature of the split operation.

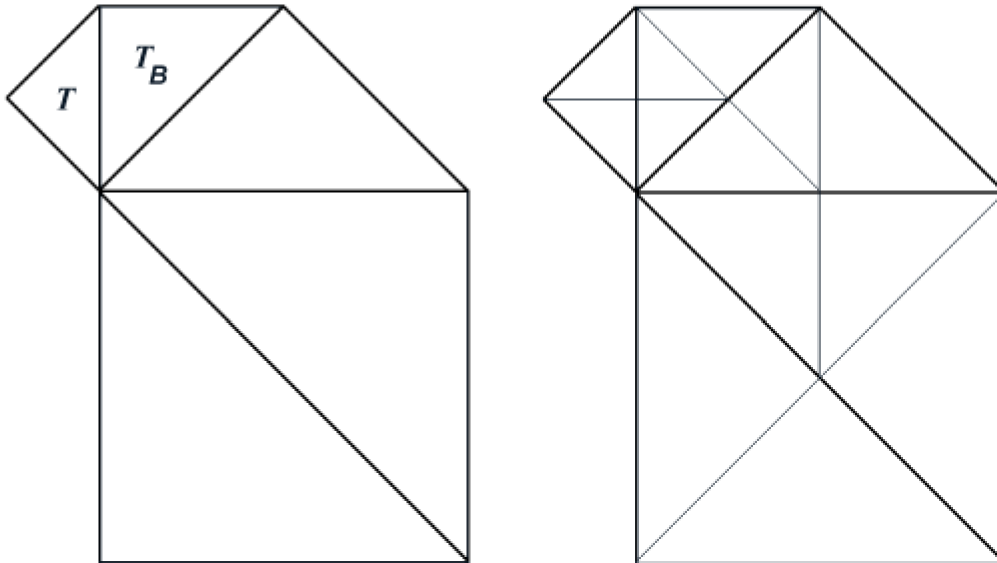


Figure 9 - forced splitting of triangle T [DUCH1997]

4.4.2 Terrain Tessellation

Now that the splitting operation has been defined, the top-down tessellation algorithm proceeds as follows:

```
Tessellate(pTriangle)
{
   // If this triangle has already been split, don't split it again
   if(pTriangle->m_pLeftChild || pTriangle->m_pRightChild) {
      Tessellate(pTriangle->m_pLeftChild);
      Tessellate(pTriangle->m_pRightChild);
      return;
   }

   // Calculate the variance for this triangle
   priority = CalculatePriority(pTriangle);

   // See if this triangle should be split
   if(priority >= 1.0) {
      SplitTriangle(pTriangle)
      Tessellate(pTriangle->m_pLeftChild);
      Tessellate(pTriangle->m_pRightChild);
   }
}
```

In order to tessellate a terrain, the above code would be called for each of the two triangles in the base triangulation. The algorithm starts by making sure that the triangle is not already split into its children. This case can occur by the forced splitting operations discussed previously.

The next step is to calculate the triangle's priority value. For the time being, assume that each triangle T is given a priority value $p(T)$ ª [0, 1], where 0 means the triangle should not be split anymore, and 1 means that the triangle requires further splitting. The error metrics will be discussed in more detail in Section 4.7.

The code concludes by checking whether the triangle needs to be split further, and if so, performs the split and then recursively tessellates the split triangle's children.

After the algorithm is finished tessellation, we will have two Binary Triangle Trees, one for each of the base triangles, with the base triangles as the respective roots. Thus, in order to render the terrain, we could simply traverse each of the trees and render the leaf nodes.

As can be seen, the algorithm is extremely simple, with most of the complexity quietly resting in the `CalculatePriority` function call. This will be saved for Section 4.7.

[DUCH1997] uses a priority queue to drive the split operations for tessellation, instead of the recursive nature of the above algorithm. After forcing a triangle to split, the priority queue tessellation algorithm then proceeds to remove the split triangles from the priority queue, and then inserts the new split children. This results in an extremely elegant tessellation system which, at each step, always splits the triangle

which requires tessellation the most. Additionally, due to the step-wise nature of the splitting process, the algorithm can achieve strict frame rates by simply jumping out of the tessellation loop if a certain amount of time has passed, with the tessellation being as optimal as possible for any given frame rate constraints.

However, the downfall with the split queue algorithm is the extra priority queue maintenance that is required. Even though most priority queue operations are O(lgn), they still present an extra overhead which is not needed in our top-down algorithm [CORM1990].

4.5 Texturing

Texture mapping is the process of mapping or projecting an image, either digitized or synthesized, onto a geometric surface. The image is called a *texture map*, and its individual elements are called *texels* [FOLE1990]. In the case of terrain, geometry is defined by vertices and triangles, but the surface details *within* each triangle are represented using texture mapping, in order to give the terrain a photo-realistic look.

Before continuing with a discussion of the metrics used in the top-down tessellation algorithm, its important to address the issue of terrain texturing as it plays an important role in determining storage schemes of certain LOD error metrics.

4.5.1 Unique Texturing

The concept behind unique texturing involves projecting a single, large texture map over the entire terrain area, such that there are no tiling or repeating effects visible, and the mapping is planar in the X and Z directions.

The main advantages to this method are:

- dynamic tessellation doesn't adversely affect texture coordinates
- tiling patterns can be completely eliminated
- lighting can be "baked-in" to the texture, since the texture doesn't repeat
- dynamic changes to the terrain (eg. explosion scorching) can be applied directly to the texture

The disadvantages to this method are:

- the texture can become quite large for large terrain areas
- certain consumer-level 3D hardware have maximum texture sizes
- its difficult to have high texture detail if one texture is stretched over a large terrain

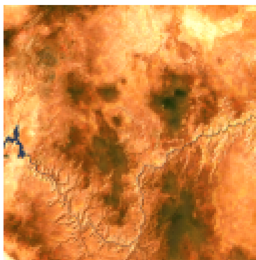


Figure 10 [HOPP1997] **Unique colour texture of the Grand Canyon**

Despite these disadvantages, unique texturing is the ideal way to texture a terrain since it takes advantage of the regular spacing and planar geometry of height maps. Figure 10 shows a unique colour texture for the Grand Canyon, with 20m sampling. While a flyby of this Grand Canyon terrain from a high elevation would give spectacular results, a surface walk of a terrain with 20m colour spacing would not be very realistic, since each square area of 20m would contain one large colour patch. As will be seen in the next section, certain visual tricks can be applied to address this lack of close-up texture detail.

4.5.2 Detail Texturing

Detail Texturing is the process of subtly blending an extra high resolution texture, which is often small and infinitely tiling, over top of a larger texture to add apparent detail, either using a second rendering pass or multi-texturing hardware (see Figure 11).

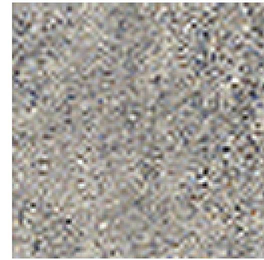


Figure 11
Small (64x64) detail texture

Consider a 1024x1024 terrain, with 1m spacing, and a unique 1024x1024 colour texture stretched over the height map as discussed in the previous section. This results in each 1m block of terrain to refer to only 1 texel in the colour texture, such that each pixel in the texture represents 1m. Clearly, the terrain will not be very detailed when observed close-up. Now consider *blending* the small 64x64 texture from Figure 11, such that it tiles over the entire terrain at its original resolution (ie. no stretching). This will give the entire landscape a sense of extra roughness, thereby reducing the large, bland colour patches which can become apparent when simply using a stretched colour texture.

Conceptually, this is almost the inverse of how id Software's *Quake* series of games perform their lighting. In *Quake*, the colour textures are high-resolution and tiling (eg. brick wall textures), while the lightmaps are extremely low resolution and unique for each wall. On the other hand for terrain, the colour texture is extremely low resolution and unique, and the detail texture map is high-resolution and tiling. Figure 12 shows some texture blending in action.

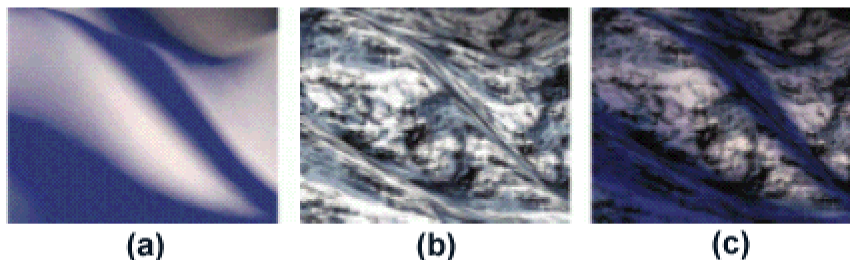


Figure 12 [SHAR1999a]
(a) colour texture, (b) detail texture, (c) colour texture and detail texture blended together

4.5.3 Texture Blocks and Terrain Blocks

For a 1024x1024 colour texture, theoretically it would be ideal if the terrain engine could simply throw the entire image at the 3D hardware, and be done with it. However, there are a few reasons that this is not optimal:

- some lower-end cards on the market have maximum texture sizes (eg. 3Dfx's popular *Voodoo* line of 3D cards don't support textures larger than 256x256)
- 3D cards typically have dedicated on-board texture RAM, with ranges of between 4MB for older Voodoo cards up to 32MB for modern hardware such as the Nvidia *Riva TNT* series. As such, texture RAM is a limited resource.
- the entire terrain isn't always visible at all times, which results in a large portion of a 3D card's limited texture RAM to be wasted with the large colour map

Now consider splitting the entire 1024x1024 colour texture into smaller blocks of 64x64 textures. This would guarantee that all current 3D hardware would be properly handled, as well as allow the terrain engine to easily swap these smaller texture portions in and out of texture RAM based on visibility. But how does this affect the Binary Triangle Tree structures discussed so far?

For a 1024x1024 terrain area, the current top-down algorithm assumes the entire height map has two base triangles. What if the entire 1024x1024 terrain height map were ALSO split into blocks of 64x64, such that each block referred to its own 64x64 portion of texture from the original 1024x1024 texture? This can be done quite easily without affecting the split operations, since the binary triangle structures have connectivity information with neighbour triangles. As such, each of these 64x64 "patches" of the terrain height map would then have their own base triangulation of two triangles each, with the appropriate assignment of neighbour triangles in order to avoid T-junctions.

But what are the performance implications of this? Assuming 1m spacing, the minimum triangle size thus becomes 64m, even if the block of terrain is 1000m away from the viewer. This means that at 1000m, with a 90 degree field of view, there would be 1000 / 64 16 blocks across the screen, which actually isn't very bad. Additionally, the top-down algorithm doesn't have to start from a base triangulation of only 2 triangles for the entire terrain each frame, and can also make use of a quad-tree based culling scheme to completely eliminate patches, as well as textures, which are completely out of view (as will be discussed in the next section). As a result, the performance hit of splitting the terrain into 64x64 blocks is negligible compared to the more efficient use of texture RAM and broader hardware support. Figure 13 shows the new base triangulation scheme.

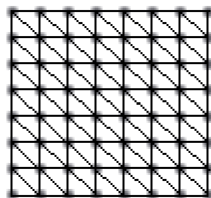


Figure 13 - Each block represents a base triangulation of a 64x64 portion of the terrain. Additionally, each block has its own 64x64 portion of the unique colour texture.

4.6 Quadtree Storage of Blocks

The final order of business before discussing our error metrics involves developing a way to store the large area of terrain in terms of its 64x64 blocks, which were discussed in Section 4.5.3. [SHAR1999a] suggests using a quadtree data structure to represent a terrain built up from a set of square Bezier patches. The approach can be easily adapted to handle our Binary Triangle Tree blocks in a similar fashion.

A quadtree is a tree where each interior node has four children. For terrain purposes, each leaf node would represent a single 64x64 block of terrain, which contains its own texture and a Binary Triangle Tree initially consisting of two base triangles. The nodes at the level above the leaves would then contain a 2x2 grouping of adjacent blocks of terrain. These nodes would then be contained within the next higher up level in a 2x2 group, and so on, at which point the root of the entire tree encompasses the whole terrain area.

The benefit in using a quadtree to store the terrain blocks is the fast culling potential it provides. At each node, if a bounding sphere around all the children is known, then this sphere can be checked against the six clipping planes of the view frustum. If the sphere is determined to be outside the view, then we can discard that node and all its children from further processing. Otherwise, recursively process the node's children until either a sphere is outside of view, or we reach a leaf node which can be tessellated using our top-down tessellation algorithm. This quick O(lgn) algorithm can be stated as follows:

```
ProcessQuadtree(node)
{
  if node is a leaf {
    Tessellate(node)
    return
  }
  if Sphere(node) is in view {
    ProcessQuadtree( child1(node) )
    ProcessQuadtree( child2(node) )
    ProcessQuadtree( child3(node) )
    ProcessQuadtree( child4(node) )
  }
}
```

The benefit in using bounding spheres as opposed to bounding boxes is that spheres can be checked against the view frustum extremely fast. They provide a trade-off between a looser fit, but a faster rejection test during culling.

4.7 LOD Error Metrics

So far, the top-down algorithm starts with the coarsest level of detail for the terrain, and continuously performs split operations based on a priority for each triangle in the queue. But how are these priorities calculated? And how does the algorithm know to stop tessellating?

4.7.1 Bounding Spheres for Triangle Prioritization

Section 4.6 showed a way to get down to each 64x64 block of terrain, but we still need to address the actual triangle prioritization as a block of terrain is tessellated. It was mentioned earlier that each binary triangle T is given a splitting priority value $p(T) \in [0, 1]$, where 0 is lowest priority, and 1 is highest. The simplest way to assign these priorities is using traditional bounding frustum checks with bounding spheres, similar to what was discussed in the previous section.

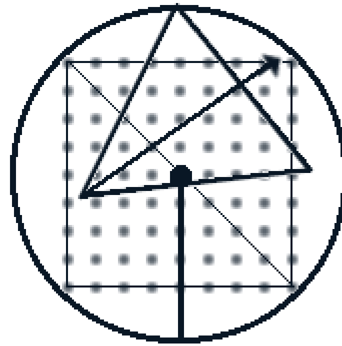


Figure 14 - Bounding sphere for base triangulation (each triangle has its own sphere)

Starting with the base triangulation, each of the two triangles are given a 0 or 1 priority based on whether a bounding sphere around the triangle is outside or within the view volume. The bounding sphere is calculated with its centre point at the centre of the triangle's hypotenuse (the split point), and a radius of half the length of the hypotenuse length (see Figure 14). Additionally, the radius is adjusted by the triangle's height variance as well (which is discussed in the next section). This provides a loose, but quick, bound around the triangle which can then be checked with the frustum planes. If the triangle is determined to be completely outside the view, it is given a priority of 0 (although it can still be split by recursive splits to other higher priority triangles). On the other hand, if the bounding sphere is even slightly *behind* the view volume, but still partially within the frustum bounds, the priority is artificially increased to 1. The motivation behind this increase is that the triangle is potentially very close to the view. Similarly, by forcing this triangle to be given maximum splitting priority, the algorithm will be better able to discard other portions of the triangle block which may definitely be completely out of view.

The bounding sphere checks outlined above provide a quick and dirty way to discard large sections of a terrain block which are completely out of view. However, for triangles which are neither completely out of view, nor making contact with the back clipping plane, the above 0 or 1 priority assignments are insufficient.

<u>4.7.2 Variance Trees</u>
In current level of detail research, there is still no *definitive* way to decide *what* geometry to tessellate, and *when* to do it. Error metrics are typically specific to the type and size of geometry being displayed, with plenty of tweaking of thresholds until the geometry generally looks "good enough". Terrain error metrics are no exception.

In Section 4.3 it was shown that as objects get further away in Z, they occupy less and less pixel space on the screen. Ideally, this pixel variance should be the main basis for the triangle priority computations, such that we have an accurate measurement of how detailed a terrain block is at any given point. Interestingly, since terrain variations are largely based on their height (Y) values, its possible to develop just such an error metric quite easily.

Consider the base triangulation for a block, where each of the two triangles are defined completely using the four corner points of the terrain grid (Figure 7b). While these base triangles will cover all inner points of the terrain block, the rendering of these triangles completely ignore the height (Y) values. For example, consider a terrain block whereby the four corner points are at an elevation of 0m, and the centre point of the terrain is at an elevation of 100m. Clearly, our algorithm should not stop tessellation at the base triangulation since there is a large variance within the *area* of the base triangles. Somehow our priority computation needs to consider the height values within the area of the triangle as well.

Now assume that at each step of our triangulation, our algorithm has knowledge of this variance within each triangle's area. So in the example above, the base triangulation would know that the height values within the area of each triangle varied by 100m. If this variance of height values within the area of the triangle vs. the flat plane of the triangle (as scaled by the distance from the camera) is greater than some pre-defined maximum-variance quality setting, we would need to tessellate more. Similarly, if this projected variance were below our quality threshold, tessellation for the triangle could be stopped.

In other words, we have the following variance calculation:
variance = (max_height – min_height) *within the <u>area</u> of the current triangle*

Now we can "project" this height variance into screen space using the standard projection equation:

projected_variance = variance / Z

where *Z* is defined as the centre of the triangle's hypotenuse (ie. the same centre point used for the bounding sphere checks).

This projected variance value would thus represent the height difference, *in pixels*, between the current triangle and the actual height variance within the area of the triangle.

As a result, we can set triangle priorities such that:

priority = projected_variance / pixel_threshold

where *pixel_threshold* >= 1 is the desired screen space accuracy that the tessellated terrain triangles should achieve. Additionally, to force the triangle priority between 0 and 1, the *projected_variance* can be capped to a maximum of *pixel_threshold.*

Our priority computation can be summarized in pseudo code as follows:

```
T = current_triangle

if sphere(T) is completely out of view {
  priority(T) = 0
}
else if sphere(T) touches back clipping plane {
  priority(T) = 1
}
else {
   projected_variance = variance(T) / Z(Sphere(T))

  if projected_variance > pixel_threshold
     projected_variance = pixel_threshold

  priority(T) = projected_variance / pixel_threshold
}
```

For example, assuming that *pixel_threshold* = 8 pixels, and the *projected_variance* for some triangle was only 2 pixels, the splitting priority for this triangle would be 2 / 8 = 0.25 (relatively low priority). Similarly, if some triangle had a *projected_variance* of 12 pixels, the priority computation would set *projected_variance* = *pixel_threshold* = 8. Thus this 12 pixel varying triangle would be given priority of 1.0 (highest priority).

One remaining issue that remains to be solved is how to retrieve the *variance* values for each triangle. Clearly, calculating the variance at run time would involve traversing all the points in the area of the triangle to find the minimum and maximum height values. At the base block size of 64x64, the base triangulation would involve checking all (64x64)/2 = 2048 points for each of the two triangles, which we do **not** want to do!

The other alternative is to *precompute* the variances for each binary triangle. The following pseudo code will calculate the variance for any binary triangle T, assuming the Binary Triangle Tree exists :

```
CalculateVariance(T) {
 real_height = terrain height at the middle of the hypotenuse
 average_height = average of terrain heights at two ends of the hypotenuse
 variance = | real_height - average_height |

 if LeftChild(T) is valid
  variance = max( variance, CalculateVariance(LeftChild(T)) )
```

```
 if RightChild(T) is valid
  variance = max( variance, CalculateVariance(RightChild(T)) )

 return variance
}
```

The final piece of the puzzle involves *storing* these variance values. Consider the base block size of 64x64 points. Starting with the base triangulation, for each triangle that can be split, 2 additional variance values are required. Continuing down to the finest detail level for the terrain, this 64x64 grid of values will contain 64 different levels of binary triangles. In general, an nxn grid of values contains close to n levels of triangles. Thus, to store our variance values for each level would require 2^n entries. Therefore for a single 64x64 block, this requires 2^{64} entries, which is ridiculous!

Clearly there must be a better way to store these variance values. The `CalculateVariance` function outlined above shows an interesting property: each variance at any given level is calculated as the maximum of its own variance and that of its 2 children's variances. So each variance in every child is actually a *subset* of its parent's variance value. Thus, would it be possible to *cap* the variance tree down to a fixed size? Suppose the 64x64 terrain block had its variance tree capped up to the first 8 detail levels. This would give us 2^8 = 256 entries which, at 4 bytes per pixel, occupies 2K of memory (1K for each of the base triangles). This is much better! But how does this perform in terms of capturing variances? In the 64x64 case, every 2 levels divide the area covered by a triangle by almost half. Thus, after 8 levels, the variance levels will represent approximately (64/4) x (64/4)   16x16 grid points. With a 1m pixel spacing, a 16x16 grid coverage is actually quite detailed. In the best case, such as a sudden peak within the 16x16 area, the variance level at the 8th level would completely take the peak into consideration (due to the subset property). In the worst case, such as rather flat areas below the 16x16 grid resolution, the variance levels would be *too* accurate, thus causing some extra tessellation in flat portions of the terrain which don't require it. So in the end, capping the variance at 8 levels for a 64x64 terrain block is quite sufficient.

The following *high-level* pseudo code summarizes the complete top-down rendering algorithm:

```
For each visible block in quadtree
   Tessellate the block and build its Binary Triangle Tree
   Add the block to a visible block list

For each block in visible block list
   Bind the block's 64x64 texture
   Traverse the block's Binary Triangle Tree, and render the leaf triangles
```

4.8 Achieving Target Triangle Counts

For real-time applications, such as fast-moving games or simulations, sometimes fluid frame rates and visual responsiveness are more important than the visual quality of the 3D world. Similarly, in a world of ever increasing processing power and triangles per second, its important to provide a 3D rendering system with a bit of scalability to account for lower end hardware.

[DUCH1997] describes a simple and elegant way to achieve strict frame rates, by checking the amount of time that has passed after each split operation. However, this assumes a global tessellation algorithm, which clearly doesn't fit well into our block-by-block tessellation scheme.

On the other hand, frame rates and the triangle count in a scene are inversely related to one another. If a scene increases in triangle complexity, the frame rate usually decreases, and if the triangle complexity decreases, the frame rate increases. So it follows that if we use our triangle count, rather than time, as our metric each frame, it should be possible to achieve a certain level of "smoothness" or fluidity in our frame rate.

Consider the pixel threshold error metric discussed in Section 4.7, which is the primary variable that can be used to control the amount of tessellation. While typically this value is fixed to some predetermined value, [MCNA1999b] suggests algorithmically modifying it at run-time in order to progressively achieve a target triangle count.

At the end of each frame, a ratio between the drawn triangle count and desired triangle count can then be calculated, and then used to modify the pixel threshold for the next frame [MCNA1999b]. Progressively, as frames are rendered, the pixel threshold will descend upon a value which matches the desired triangle count.

The following pseudo code shows how this could be done:

```
RenderFrame(pixel_threshold)

threshold_adjust = current_triangle_count / target_triangle_count

if(threshold_adjust < 0.9 || threshold_adjust > 1.1) {
   // Modify the threshold
   pixel_threshold *= CLAMP(threshold_adjust, 0.8, 1.4);

   // Clamp the threshold to avoid unwanted values
   pixel_threshold = CLAMP(pixel_threshold, 1.0, 32.0);
}
```

The above code modifies the pixel threshold if the *current* triangle count varies from the *target* triangle count by ±10%. Additionally, its important to clamp the amount of adjustment to the threshold each frame and instead allow the change to occur over subsequent frames in order to avoid sudden "popping" of geometry as much as

possible. Finally, the pseudo code shows the pixel threshold being clamped between 1 and 32 pixels, in order to avoid extremely low or extremely high threshold values.

4.9 Handling Dynamic Objects

Up to this point, the terrain algorithms have assumed a rather barren landscape, void of any interesting objects such as trees, people, or vehicles. However, most games or virtual worlds consist of small dynamic objects moving around the landscape, so its important to consider the implications that the dynamic level of detail algorithms place on them.

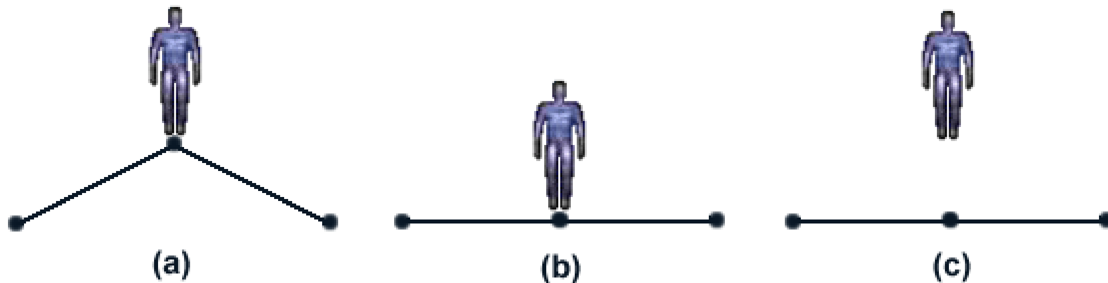


Figure 15 - Handling Dynamic Objects
(a) Ideal object position with geometry, (b) Position based on tessellation, (c) Position based on original height map

Figure 15 shows the key issue that needs to be considered with respect to dynamic objects. The object in Figure 15a shows the ideal situation, where the terrain is at the highest detail level, and the object simply chooses its Y position based on the original height map. However, when tessellating terrain, things aren't so simple. Figure 15b shows one approach where the object is positioned based on the current underlying tessellation level. As vertices are added or removed during splitting, the object similarly chooses its height position based on the current tessellation level. Figure 15c shows the opposite approach, whereby the object position is based entirely on the original, underlying height map, with no regard to the current tessellation level.

The major advantage in using the Figure 15b approach is the avoidance of the "gaps" which are apparent in Figure 15c. Objects which are supposed to be on the ground will always be on the ground, no matter what tessellation level the terrain is at. The disadvantage with this approach is the constantly changing object positions, which can cause serious synchronization problems when dealing with multi-user worlds, or in simulations where accurate physics models are employed.

Thus the Figure 15c approach seems most desirable, as long as the gap problem could be addressed. With the split queue algorithm from [DUCH1997], the gap problem can easily be remedied by *artificially increasing* the priorities of triangles surrounding an object's extents such that the geometry underneath the object mirrors the underlying height map as close as possible. But with our top-down algorithm, since priorities aren't handled in the same way, this method is not feasible. However, similar to the way we modified pixel thresholds to achieve certain triangle counts, we can also temporarily modify the pixel threshold on a block by block basis, such that

pixel thresholds decrease in blocks where objects are located (thus increasing triangle counts).

While this approach doesn't guarantee that an object will not have a gap underneath it, it does reduce the gap problem significantly. The drawback is the increased tessellation in areas that may not necessarily require it, such as areas extremely far from the view, or flat areas that aren't causing gap problems in the first place.

4.10 Handling Popping via Vertex Morphing

The traditional level of detail algorithms from Section 4.2, which involve a series of fixed resolution meshes, suffer from a sudden "pop" effect when meshes are swapped. While not as sudden, the Binary Triangle Tree tessellation algorithm results in similar anomalies. Figure 16 shows the popping effect that can occur from frame to frame as the view changes.

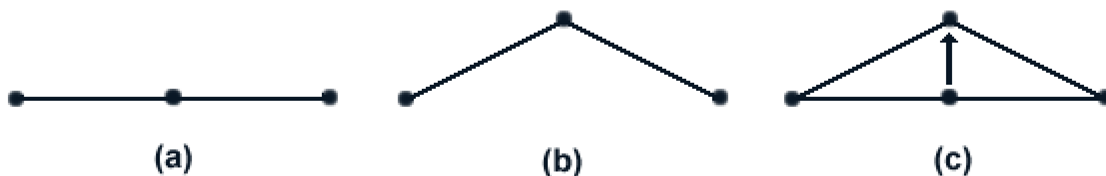


Figure 16 - Cross-section view of vertex popping (a) cross-section at frame f, (b) cross-section at frame f+1, (c) vertex morphing over time

Figure 16a shows a cross section view of a set of vertices at some frame time f. At this time, only the end points will be used, and the middle point is thus the interpolated position based on the two endpoints. However, as can be seen in Figure 16b, the actual height at the midpoint is different. Thus at frame f+1, if the triangle from Figure 16a were to split to give us Figure 16b, a "popping" effect would be seen in the terrain upon rendering. For small height variances, the popping is nearly negligible. Additionally, if the pixel error threshold is set close to 1, popping can also be almost completely eliminated, but at the expense of increased triangle counts and decreased frame rates [HOPP1997]. Thus, when trying to achieve high frame rates in terrain with many peaks and valleys, the popping effect can become quite noticeable to the point of reducing any perceived realism that the virtual terrain is able to provide.

Figure 16c shows a conceptually simple way to reduce the popping effect. Rather than suddenly switching a vertex to its new position, the vertex can be *animated* or morphed to its destination position over a few frames.

In our dynamically tessellating terrain, this vertex morphing can be achieved by declaring a Z range *just past* the point of the triangle being split which determines the amount of vertex morph, from none (looks the same as if the triangle isn't split) to full (looks as if the triangle is split normally). This value, which goes from 0 to 1, can be stored in a member of the binary triangle structure, along with two more variables

which define the triangle's current Z distance from the camera, as well as the triangle's split Z distance. The way to get around bottom-neighbouring triangles having different morph amounts is simply to average their morph amounts together when making the rendering pass (to avoid cracks).

So the algorithm would look as follows:

```
void calculateGeomorph(BTT_Triangle *pTriangle)
{
   t = pTriangle->m_geomorphZ - pTriangle->m_currentZ;

   if(t < 0)
      t = 0;

   // See if the triangle is in the morph range
   if(t < TB_MORPH_RANGE) {
      pTriangle->m_geomorph = t / TB_MORPH_RANGE;
   }
   // Otherwise use the sampled height, no morph
   else {
      pTriangle->m_geomorph = 1.0f;
   }
}
```

where *TB_MORPH_RANGE* is some arbitrarily set range of Z distance in which the morph occurs (eg. 1.5 meters), and *currentZ* and *geomorphZ* are, respectively, a triangle's current Z distance and typical split distance from the camera.

The *currentZ* can be easily calculated during the triangle's priority computation, by simply setting the *currentZ* value to the Z value of the centre of the triangle's hypotenuse (since this is where the split occurs). To calculate the *geomorphZ* value, we need to take the current pixel threshold into account and figure out at which Z point the triangle would normally be split. From our projection equation, we know that the split will occur when the projected variance value is greater than or equal to the current pixel threshold. From Section 4.7.2 we had:

projected_variance = variance / Z

Thus to find the split Z value, we want *projected_variance* = *pixel_threshold*:

pixel_threshold = variance / Z

Solving for Z, we have:

Z = variance / pixel_threshold

where Z becomes our *geomorphZ* value.

Now in order to apply the morph values to the geometry, the geomorph values for each triangle can simply be multiplied with the triangle's split height point during the rendering pass of the Binary Triangle Tree, with the morphed height values being passed down to the split children as we recursively traverse the tree down to the leaves.

4.11 Generating Triangle Strips

In Section 4.7.2 we outlined the high-level tessellation and rendering algorithm, where it was mentioned that the rendering pass of the Binary Triangle Tree simply traversed the tree and rendered the leaf nodes. Typically, the traversal uses a fixed left-right or right-left traversal order, and then simply renders a single triangle at the leaf node.

Now consider the overhead of simply rendering a single triangle at a time. For each triangle, three vertices will be transformed, and then passed onto the rendering hardware, which will perform further calculations for texturing and rasterization, and then the triangle will be output to the screen.

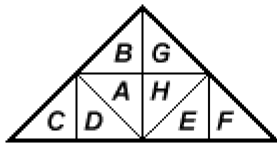


Figure 17 - Rendering order assuming a left-right tree traversal (A = first, B = second, etc.)

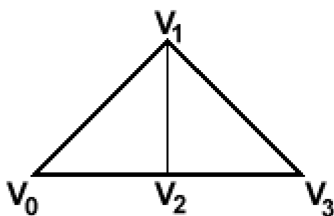


Figure 18 - Triangle strip consisting of two triangles and four vertices

Figure 17 shows the typical rendering order for a left-right tree traversal. It is clear to see that every pair of left-right leaf triangles can be rendered together as *triangle strips*, such that the two triangles can be passed to the rasterization system as one unit, requiring only four vertices instead of six (see Figure 18).

For consumer level rendering hardware, this results in a 2/6 = 33% performance gain with respect to vertex transformation overhead, which can be significant in highly detailed scenes.

Looking at Figure 17 again, it seems as though it may be possible to increase our strip depth even further if we were clever about our tree traversal. [MCNA1999b] notes that by alternating the recursive diving order, triangle strips consisting of three or four triangles each can be easily achieved. That is, for the first level we visit left child, right child, for the second level we visit right child, left child, etc. A slight overhead exists with respect to extra conditionals to determine which vertex becomes the fan point, and when a triangle isn't part of the fan, but the alternating traversal method should still end up being a speed win [MCNA1999b].

5 Representing Arbitrary World Geometry

Section 4 described how Binary Triangle Trees could be used to represent detailed, dynamically tessellating terrain. However, the ultimate goal of any 3D system is to be able to represent any arbitrarily oriented geometry, such as a highly detailed castle overlooking a rolling green landscape, or a highly detailed cave featuring realistic stalagmites and stalactites. This section describes a few simple ways the Binary Triangle Tree structures can be extended to handle such arbitrary 3D worlds.

5.1 Binary Triangle Tree Patches

Section 4.5.3 showed how a large height map of terrain data could be split into blocks of 64x64 for texturing purposes. Additionally, these 64x64 blocks posed no major difficulties with respect to T-junctions, since the general form of binary triangles take neighbouring triangles from other blocks into account.

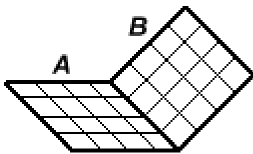


Figure 19 - attaching arbitrarily oriented patches together

Extending this neighbouring attachment idea, it should be perfectly feasible to attach *any* arbitrarily oriented block to any other block, as long as they share a common edge. Figure 19 shows two arbitrary height fields, or *patches*, being attached. Thus, building tunnels or caves with detailed walls and ceilings is simply a matter of translating and rotating the entire height field to the desired position, and making the appropriate attachments to neighbouring patches.

This simple extension of Binary Triangle Tree patches has the following advantages:

- We can now represent detailed indoor environments such as caves, with dynamic level of detail
- Merging arbitrary patches together is a simple matter of assigning the appropriate neighbour pointers, and completely avoids cracks
- Different colour textures can be applied to each patch, removing the reliance upon a single colour texture for an entire terrain

However, this method has some major downfalls as well:

- Neighbouring patches must have edges which are at the exact same location (ie. they must be at height 0 with respect to the patch's flat plane)
- Neighbouring patches must be the exact same size, in order to have edges meet up properly.
- Certain indoor settings, such as office buildings, don't require the density of vertex and triangle information that a Binary Triangle Tree patch provides, since floors and walls are usually flat and planar

Clearly, this patch method seems ideal when representing large scale areas, such as terrain and caves, but fails when representing any sort of indoor human structures such as buildings. Section 5.2 touches on some future possibilities when representing this sort of indoor geometry.

5.1.1 Comparison to Bezier Patches

[SHAR1999a] implements a dynamically tessellating terrain using a similar patch-based approach, but rather with Bezier patches instead of Binary Triangle Trees. Its worth quickly comparing the two approaches since they both attempt to achieve similar goals with respect to dynamic level of detail and arbitrary geometry representation.

Advantages of Binary Triangle Tree (BTT) Patches over Bezier Patches

- Handling cracks (T-junctions) is implicit in the BTT representation, while Bezier patches require special case testing
- The BTT variance tree allows an elegant pixel based error threshold based on Z distance from the camera, while Bezier patches must be tessellated globally for the entire patch using less intuitive schemes
- Adjustable BTT pixel threshold allows easy implementation of strict triangle targets
- Building desired geometry is intuitive, since we can simply use greyscale displacement maps or vertex manipulations to assign our height fields (see Appendix A for more detail) as opposed to manipulating Bezier control points.

Advantages of Bezier Patches over Binary Triangle Tree (BTT) Patches

- Bezier patches can tessellate "infinitely", since they're parametric as opposed to the fixed height-field coarseness when using BTTs
- Bezier patches require much less memory overhead, since they use control points to define the geometry rather than exact height data

Each of the two methods provide various benefits and trade-offs, and thus neither one is clearly better than the other. The Binary Triangle Tree method may have a slight advantage due to the more intuitive and exact height field creation process it provides, but at the expense of increased memory requirements for large scale areas. Of course, there's no reason why the two methods can't be used together.

5.2 Future Considerations

5.2.1 Merging Traditional Indoor Representations

The Binary Triangle Tree representation of terrain is clearly an elegant way to obtain dynamically tessellating landscapes consisting of hills and valleys. Now consider a typical indoor area consisting of large, flat walls connected at 90 degree junctions. Level of detail algorithms clearly aren't required in such situations, since the geometry is already very coarse and planar in the first place. Additionally, terrain and indoor structures both have largely conflicting goals; indoor 3D systems can make use of occluding geometry and visibility algorithms in order to determine what needs to be drawn, while outdoor 3D systems typically need to see right up to the horizon, with very little occlusion. Thus when it comes to representing indoor structures such as office buildings, structures such as BSP trees or portal engines are more suited to the task.

This brings up the question of whether such traditional indoor representation methods can be combined seamlessly with Binary Triangle Tree patches. Clearly, placing an entire BSP based structure on a flat area of terrain poses no major problems, except for the overdraw which would occur with the terrain geometry underneath. Issues dealing with gaps (such as if a large BSP structure was placed on a flat mountain top and was viewed from a far distance) could easily be solved using methods similar to the dynamic object handling that was discussed in Section 4.9, whereby the underlying blocks would artificially increase in detail.

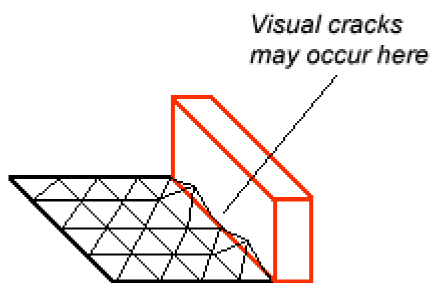


Figure 20 - Edge issues when merging traditional geometric structures with BTT patches

The only major issue would be the crack/T-junction problems which we were so careful to avoid when generating the level of detail terrain geometry. Figure 20 shows such a situation, where the rectangular BSP slab doesn't share the same vertices as the terrain patch which meets up with it. As a result, the merge area between the patch and the slab could show visual anomalies such as cracks when rendered, reducing the realism of the virtual world. Possible solutions for the cracks include either restricting terrain patches such that merged edges must all be flat, or including the extra terrain vertices as part of the slab's geometry. The latter method seems overly complex however, since the terrain patch may tessellate to any level and possibly cause some of the slab's faces to become concave.

5.2.2 Handling Details using Continuous Meshes

Many indoor 3D worlds are built using a Constructive Solid Geometry (CSG) approach, where geometry is built up from simple 3D primitives such as cubes, spheres, and cylinders [FOLE1990]. Using a BSP like structure, the CSG world is then partitioned to create structures suitable for run-time traversal. The naïve approach to building such worlds involves placing all static geometry, regardless of size, into the BSP tree. However, due to the way BSP trees work, highly tessellated static geometry such as curvaceous pillars or furniture would completely decimate the tree. [BAGW1999] suggests a simple solution to this problem: build the large sections of an indoor world, such as walls and rooms, using the CSG-BSP approach, and leave the details to dynamic mesh objects. Intuitively, this makes sense, since not only does it keep the BSP tree efficient, but it also allows the details to be defined using progressive mesh representations [HOPP1996]. Thus localized details such as the pillars could be highly detailed without affecting neighbouring geometry. The downfall to this approach is the added overdraw that may occur, but with 3D hardware fill-rates increasing exponentially, the overdraw becomes negligible.

5.2.3 Increasing Detail at the Texture Level

In Section 4.5.2 we discussed detail texturing as it applies to terrain, such that we can increase the perceived resolution of our stretched colour texture using a tiling "noise" texture. In a sense, this detail texturing is a form of level of detail control, where detail is increased at the texture level rather than the geometry level, and can thus be applied to any sort of geometry, not just terrain. When a player walks directly up to a wall in a first generation 3D game such as Doom, the view becomes filled with large blocky pixels. The second generation 3D games such as Quake improved on this by making use of 3D hardware and bilinear filtering, where the blockiness is reduced by performing subpixel blending. Going further, the next generation of 3D games can apply the detail texture approach, such that when a player gets too close to a wall, a high-frequency texture is blended with the original texture, thus increasing the perceived level of detail of a polygonal surface without actually increasing geometry.

5.2.4 Temporally Coherent Tessellation Algorithm

On a typical flythrough of a Binary Triangle Tree terrain, the viewpoint usually changes slowly and smoothly. As a result, it follows that the triangulation for any two consecutive frames will tend to be similar to one another. Thus, if for frame ***f*** we could use the triangulation from frame ***f-1*** as a starting point, we could decrease the number of splits required per frame and thus increase performance. [DUCH1997] employs two priority queues during the tessellation phase, one containing the best splits, and one containing the best merges (the inverse of a split). Merging can be applied to a triangle T and its bottom neighbour T_B (if it exists) only if the children of both T and T_B are all in the current triangulation. While our top-down algorithm doesn't make use of split or merge queues, the concept of merging should be easily extendible to our recursive splitting algorithm, since the nodes at the level immediately above a leaf level potentially define a mergeable triangle. Further experimentation is required to make any conclusions regarding frame to frame coherence for our Binary Triangle Tree algorithm.

6 Implementation Results

The following results were obtained on a regular, consumer-level Intel Pentium 2 PC, 350MHz, with 128MB of RAM, running under Microsoft Windows 98, with an Nvidia Riva 128 3D graphics accelerator with 4MB of onboard texture RAM.

The demo was implemented using Microsoft Visual C++ 6.0, using both the Microsoft DirectX 6.0 API as well as OpenGL as follows:

- DirectInput: used for keyboard and mouse input
- Direct3D: used for interfacing to the 3D hardware
- OpenGL: used for interfacing to the 3D hardware

The implementation allows the selection of either OpenGL or Direct3D as the rendering API, but all the performance tests below were done using Direct3D.

6.1 Top-down Algorithm Analysis

Table 1 – Basic Attribute Memory Usage

This table outlines the important basic attributes used in the implementations, along with their respective memory usage:

Basic Attribute	Size
Height Map Values	32 bit float
Variance values	32 bit float
Binary Triangle structure	32 bytes
Texture Colour Depth	16 bits per pixel
Each 64x64 block, with 8 level variance cap for each base triangle	2K

Table 2 – Memory Usage Comparisons

This table shows the memory usage of the different worlds used during testing.
Note: The implementation made use of a fixed size store of Binary Triangles, from which the tessellation algorithms could draw triangles from at run-time. This helps avoid relying on operating system specific memory allocations every frame. Since the store is constant for all worlds, the triangle memory isn't specified in the following table.

World	Height Map Dimensions	Spacing	Height Map RAM	Total Blocks	Block RAM
Mountain	256x256	1m	256K	4x4	32K
Quarry	512x512	1m	1MB	8x8	128K
Grand Canyon	1024x1024	1m	4MB	16x16	512K

As can be seen, the growth rate with respect to memory usage is linear as the size of the world increases.

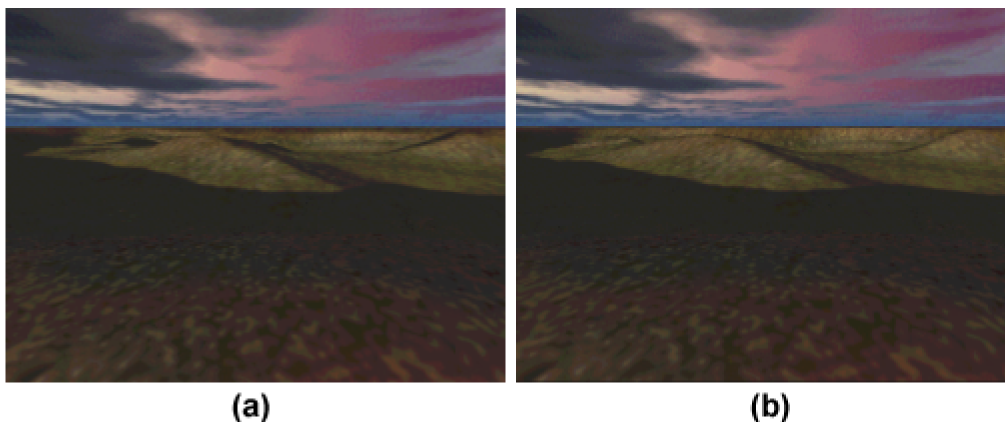
Table 3 – Performance Comparisons
This table shows the performance of the algorithms in terms of tessellation and rendering time. Each of the following tests were performed with *vertex morphing*, *dynamic object handling*, and *collision detection* all enabled. The camera was approximately positioned in the centre of the world, and then a 360 degree rotation was performed in place.

World	Average Frame Rate (Hz)	Pixel Threshold	View Distance	Tessellation Time (ms)	Render Time (ms)	Average Splits Per Frame
Mountain	30	8 pixels	1000m	1.7	17	910
Quarry	29	8 pixels	1000m	1.3	19	550
Grand Canyon	18	8 pixels	1000m	3.8	39	1050

6.2 Summary

World Descriptions
Mountain – small world, with lots of jagged terrain
Quarry – medium sized world, with smooth rolling hills and valleys
Grand Canyon – large world, with some areas which are extremely flat, other areas which have deep, narrow valleys

In Table 3, its interesting to see that the Mountain world (which was a quarter the size of the Quarry world) actually had more splits on average. This is largely due to the jaggedness of its terrain compared to the smoothness of the Quarry world. Figure 22 shows two scenes of the Quarry, with different threshold amounts. Figure 22a shows a scene with a pixel threshold of 8 pixels, resulting in 652 rendered triangles. Figure 22b shows the same scene, but with the threshold at 2 pixels, resulting in 3194 triangles. Notice that large-scale details in geometry are maintained in Figure 22a compared to Figure 22b, even though there are almost 5 times less triangles.



(a) (b)

**Figure 22 - Quarry scene with:
(a) 652 triangles (threshold = 8), (b) 3194 triangles (threshold = 2)**

Another interesting thing to notice in Table 3 is the number of splits in the Grand Canyon world compared to the Mountain world. While the split count was quite similar, the Grand Canyon world still took more than twice as long to tessellate. This is attributable to the larger world size for the Grand Canyon; even though the number of splits was similar to the Mountain world, the fact that the world was 16 times larger had a large effect on the recursive tessellation algorithm in general. A temporally coherent algorithm, as mentioned in Section 5.2.4, could potentially reduce such world-size related tessellation times to almost a constant amount, if implemented correctly.

Figure 23 shows some scenes from the Grand Canyon world. Each of the rendered scenes is shown from the same viewpoint, but with different target triangle counts. Figure 23a shows the scene with 1047 triangles, which can easily be played back at interactive frame rates on almost any consumer level 3D hardware. Figure 23b shows the same scene with 3609 triangles. Notice how the 1047 triangle scene maintains many of the close-up details as the 3609 triangle scene. Only at extremely far distances can we see a noticeable change in the geometry. Similarly, Figure 23c shows the scene with 7213 triangles, which is almost indistinguishable from the 3609 triangle scene. Clearly, the algorithm is cleverly maintaining geometric detail where its needed the most, while removing triangle counts from the areas which are irrelevant. The associated wireframe images in Figure 23 show the triangle allocations and densities.

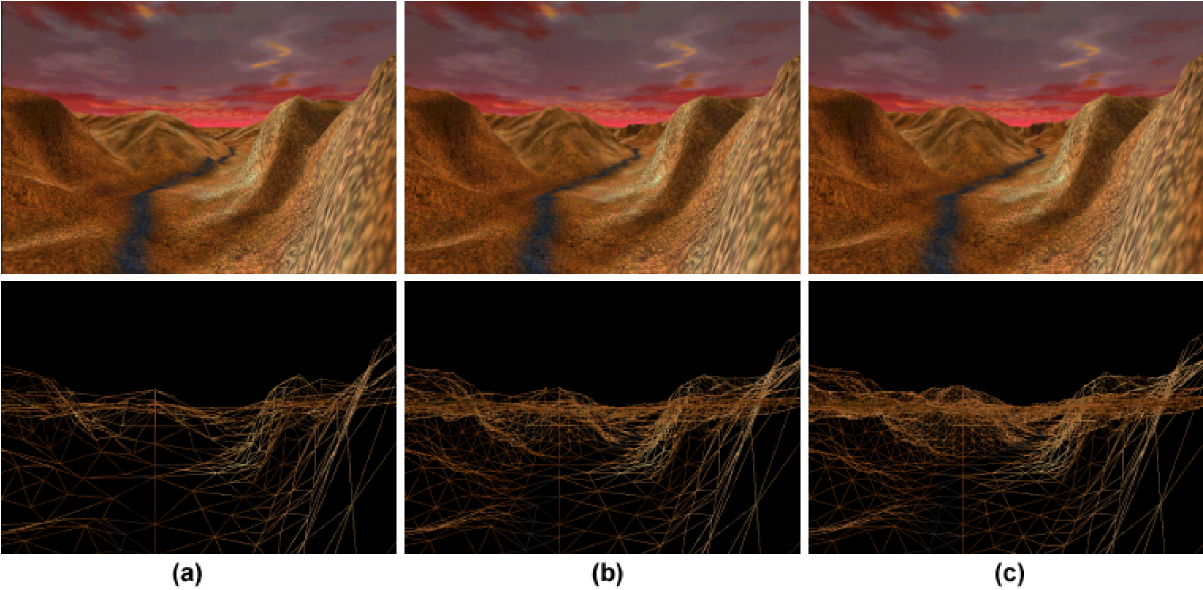


Figure 23 - Grand Canyon world with different triangle densities:
(a) 1047 triangles, (b) 3609 triangles, (c) 7213 triangles

7 Conclusion

This report presented a dynamic level of detail representation of both terrains and arbitrarily oriented “patches” of elevation information, based on Binary Triangle Trees. While the Binary Triangle Tree approach can easily represent worlds such as rolling landscapes and caves, it doesn’t scale well to general-purpose geometry such as office buildings or other indoor urban structures. In such cases, dedicated indoor rendering systems such as BSP trees were suggested, with a brief discussion of how to incorporate them with the Binary Triangle Tree height maps.

8 References

Note: *Some of the web links may no longer be active.*

[BAGW1999] Bagwell, W. Private Communication (email), Valve Software, 1999

[BENT1997] Bentley, C. Rendering Cubic Bezier Patches. http://cs.wpi.edu/~matt/courses/cs563/talks/surface/bez_surf.html

[CORM1990] Cormen, T., Leiserson, C., Rivest, R. Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts, 1990.

[DUCH1997] Duchaineau, M., Wolinsky, M., Sigeti, D., and Miller, M. ROAMing Terrain: Real-time Optimally Adapting Meshes. Visualization '97 Proceedings (1997), IEEE, pp. 81-88.

[FOLE1990] Foley, J., van Dam, A., Feiner, S., Hughes, J. Computer Graphics: Principles and Practice. Addison-Wesley Publishing, Reading, Massachusetts, 1990.

[HOPP1996] Hoppe, H. Progressive Meshes. Proceedings of SIGGRAPH 96, pp. 99-108, August 1996.

[HOPP1997] Hoppe, H. View-dependent Refinement of Progressive Meshes. Proceedings of SIGGRAPH 97, August 1997.

[KRUS1997] Krus, M., Bourdot, P., Guisnel, F., and Thibault, G. Levels of Detail & Polygonal Simplification. http://www.limsi.fr/Individu/krus/CG/LODS/xrds/

[LIND1996] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L., Faust, N., and Turner, G. Real-Time Continuous Level of Detail Rendering of Height Fields. Proceedings of SIGGRAPH 96, pp. 109-118 http://www.cc.gatech.edu/gvu/people/peter.lindstrom/

[MCNA1999a] McNally, S. Binary Triangle Trees and Terrain Tessellation. Longbow Digital Arts web page, http://www.longbowdigitalarts.com/seumas/progbintri.html

[MCNA1999b] McNally, S. Private Communication (email), Longbow Digital Arts, 1999

[SHAR1999a] Sharp, B. Optimizing Curved Surface Geometry. Game Developer Magazine, July 1999.

[SHAR1999b] Sharp, B. Implementing Curved Surface Geometry. Game Developer Magazine, June 1999.

9 Appendices

9.1 Appendix A –Displacement Maps

The sample terrain data sets used for Section 6 were created using a combination of 3 textures each. The first texture, which was common for all terrains, was a 256x256 detail texture. The second texture was a colour texture, which was unique for each terrain. Finally, each colour texture also had a corresponding greyscale *displacement map*. This 24bit greyscale image was where elevation information could easily be defined. White areas - RGB(255,255,255) - represent the highest regions of the terrain, while black areas - RGB(0,0,0) - represent the lowest regions of the terrain. All the greyscale colours in between thus represent the heights between the highest and lowest points. As an example, consider the Quarry world[1]. Figure 21 shows its 512x512 displacement map, along with its associated 512x512 colour map. Each texel in the greyscale displacement map represents 1 meter, and thus the entire Quarry world will be 512x512 meters. In order to go from the greyscale image to elevation data, each texel in the displacement map is broken into its RGB components, and then averaged into a single number between 0 and 255. This number then becomes the height value for the corresponding height field location. If height values larger than the 0-255 range are desired, a scale factor can be easily brought into the equation.

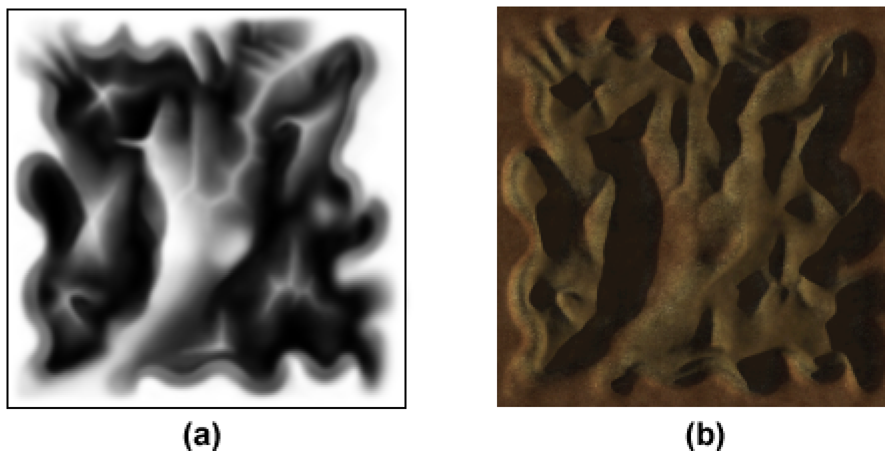


Figure 21 - (a) 512x512 "Quarry" displacement map (b) 512x512 "Quarry" colour map

[1] Quarry textures are Copyright © 1999 by Microsoft Corporation and Rainbow Studios