Boston College
Computer Science Department

Senior Thesis 2002

Cristopher Stauffer

Real-Time Terrain Rendering and Scene Graph Management

Prof. William Ames

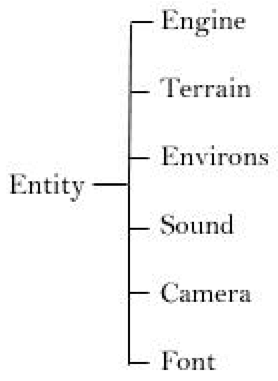
The development of computer-aided terrain rendering has over the years been the necessity of an array of applications such as topographical mapping, surgical and medical aids, as well as simulations and entertainment. The goal of terrain rendering has been redefined over the past decade due to the introduction of systems capable of real-time terrain representation. The level of detail of this terrain however has always been bound by the hardware's capability to maintain a frame-rate considered acceptable by the end user. While processing power and specialized graphics units have aided in the ability to draw an increasing amount of data with a growing amount of detail, it has been the labor of the programmer to design a software model by which the least amount of necessary data is forced into the hardware processing. The ability of the software model to render graphics must manage the data efficiently while filtering if not modifying the data in such means that the result of the execution is a perfect reflection of a hypothetical model in which infinite processing and memory capabilities were allotted.   Infinite resources within a graphics model allow the suspension of disbelief to take place, and it is towards this goal that a designer's graphics model hopes to attain.

The ability to efficiently and accurately represent terrain and therefore most forms of 3-D data has been attempted through many different models, from basic culling and visibility techniques to whole terrain models represented as space partitioning models. Basic culling and visibility determinations allow objects or sections of terrain bounded by a geometric primitive to be removed from the graphics pipeline before rendering begins, but offer no data reduction for objects within the view of the user. [Morle00] Space partitioning allows in most cases a hierarchical tree structure of the environment in order to determine visibility and possibly distance from the viewer, but again offer little data reduction for the objects contained with those visible regions. These models fail on the basis of two requirements which are demanded by any true real-time terrain world: Expandable detail and scope. While the inclusion of these models offer the ability to define with rough boundaries the viewable region, this region itself will become unmanageable as the data set grows larger due to detail or scope.

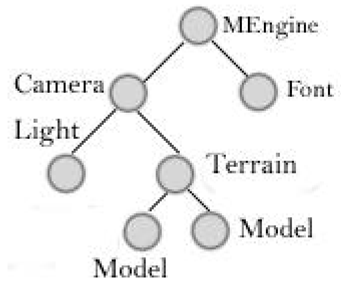
Data reduction within the viewable region is first attempted with pre-calculation in order to recreate a static mesh within fewer primitives, most likely to be triangles. Prior to the real-time rendering of the terrain, the error values, consisting of the difference between the average of two vertices and a third vertex, are measured against some constant C. If the error value is less than the constant C, then that vertex is considered non-essential, and therefore can be removed from the data set. The terrain rendering is then executed and the reduced data set will be rendered. This technique, however, fails to work on terrain that has a constant irregularity in height, in which no vertices will be removed. The memory requirements for the data set will also drastically increase, as triangles of irregular shapes are created. Data sets representing the x, y, and z locations are now necessary because the vertices are no longer guaranteed to be a constant distance apart. A problem existing within pre-rendering reduction is that these data points are physically lost within the data model, so that for applications demanding visual representation based upon constant separation, such as charting or measurements of coordinates, would be unable to easily find a data-set readily available without costly interpolation. Further the data reduction may not occur in more trafficked areas, as such is within land simulation in which travel upon water-ways is not reasonable, and therefore rarely falls into the viewable area. [Savch00]

A model and management system is necessary therefore to both manage the data, in which data reduction can occur in the correct locations, and at the same time offers reasonable memory requirements and expandability of detail and scope. In order to manage a model for terrain rendering, it is necessary to create a scene graph model in which an order of data can be efficiently managed within the program. A scene graph is the hierarchical structure by which an entire tree- representing the virtual world- is organized for efficiency and easy management. This model allows not only the efficient rendering of terrain but also the management of data and objects within the world defined by our program. [Eberl99]

The scene graph within this paper is derived from the class **Entity**, from which all other classes are derived from within the scene graph. The **Entity** object at its simplest level is a node object comprised of a parent, a child, and two siblings who share the same parent. This structure allows for a tree consisting of any object derived from the **Entity** class, and therefore allows the creation of tree of arbitrary children and siblings made up of a variety of objects, effects, and data. The **Entity** class further requires all derived classes to define a *Draw()* and *Kill()* method in which the derived object will execute either its necessary commands during the life of the program, or will clean up its allocated memory and data as well as storing any pertinent information in secondary storage. This allows for recursive calls to *Draw()* or *Kill()* to be made such that the entire scene graph will be rendered or destroyed. The scene graph management requires very few rules, and such makes it easier and much more intuitive to work with that other scene graph managers such as Sun's Java3D scene graph design. The major requirements are such that (A) An object of class **MEngine** or derivative is the root node, (B) An object of class **MCamera** or derivative is the first child of the root node, (C) An object of class **Font** or derivative is attached to the engine in order to output run-time information or debugging. Beyond the scope of these three rules, the structure of the graph is comprised of a combination of objects, which allows visual effects, weather and environmental changes, static and dynamic models, terrain, 3D music and sound effects, as well as keyboard and mouse input. Effects, such as sound or visual, usually are placed as the parent of a branch of objects which all share the common nature of the effect. Among all the objects a constant clock system as well as font engine are included in order to maintain synchronized decisions among objects as well as output debugging as well as user-useful information to the screen.



Super Class Structure



Scene Graph Tree

The hierarchical design of the tree also allows for as many children as needed to be tested for visibility through a bounding sphere test used upon the parent node. These parent nodes can quickly allow to determine the visibility of object, and because the *visible()* method within the **Entity** class can be overridden, it is possible to test for visibility using an type of geometric primitive in case the object is not successfully contained within a sphere. The allowance of an arbitrary amount of children also allows for a large list of smaller objects to be contained within the parents bounding sphere, instead of using a binary, quad, or octree design in which a large grouping of objects will cause unnecessary tests of visibility because objects are forced to be children of objects of which they should rightfully be siblings. This tree will house the model for the terrain renderer and the objects contained within. It is also possible to include just empty bounding nodes as a binary or octree structure in order to partition the space.

The model used for the terrain rendering was based upon the principle of "Level of Detail" or LOD. The concept of a LOD algorithm is to create a model by which the sharpest amount of detail is reserved solely for the closest terrain to the camera view. The level of detail, or amount of vertices used per specific region, is inversely proportional to the distance the section of terrain is from the camera view. This level of detail, in most models, is bound by a lower limit of detail in which one primitive is used to draw the entire terrain, to the upper limit in which every vertex within the data set is being represented by a point on the map [Linds96]. A LOD algorithm therefore offers a distinct advantage over other data algorithms, in that the algorithm allows the data which is immediately being viewed in the near vicinity to be rendered in its fullest detail, while saving computation and rendering time by simplifying the data set in the distance. The reason why this concept is plausible is because any data rendered at a distance, when transformed through the models perspective matrix, will resolve to a much less detailed image. Objects therefore at a distance collapse there own detail, so it is wasteful for the graphics model to attempt to draw discarded vertices. This model however, cannot be pre-calculated as could a pre-rendering vertex reduction algorithm. LOD algorithms must be dynamic, and the computation time to create this revised structure of the data set must still maintain a desirable frame rate. The LOD algorithm must also be scalable, in both detail and scope, and offer a reasonable requirement of memory. The final requirement when approaching an LOD algorithm is that it must be cost-effective to be able to integrate such features as lighting normals, primitive color blending, and texturing.

In exploring the world of LOD models, the search for such a model which coincided with the demands listed above was found in part within a quad-tree LOD algorithm originally explored by Stefan Rottger.[Rottg98] Rottger's algorithm was chosen as a base for a final LOD model because of its simplicity in design, and its ability to be easily integrated with other features of the scene graph. The description that will follow in this paper will pay close attention to the details of the author's modified quad tree algorithm, and will note those feature which were originally a part of Rottger's design, as well as those sections added or modified.

The concept of the quad-tree algorithm is the representation of a height field through series of recursive quads, for which the root quad encompasses the four corners of the height field as well as the center of the height field. By this description alone, a quad-tree then requires a height field of equal height and width, and in which the length of a side is an odd integer. Further more the recursive nature of the quad-tree, in which a quad is broken into four equal area quads, requires that the width of each quad be a power of two. Therefore the height field is bound to sizes 2^n + 1. This ensures that a height field can be recursively split using quads until each vertex is covered by the side or center of a quad. This model therefore allows for the total representation of a data set if each of these highest level quads are rendered. The most efficient choice for rendering these quads however is through the use of triangle fans. The triangle fan model with a quad has a center point located at the center vertex of the quad, and then up to 8 vertices to which it draws. Within the quad-tree model, the highest level of detail therefore is a quad that is 3 X 3 with the center point located at the location (2, 2). The triangle fan model therefore can, at highest resolution represent every vertex in the data-set.

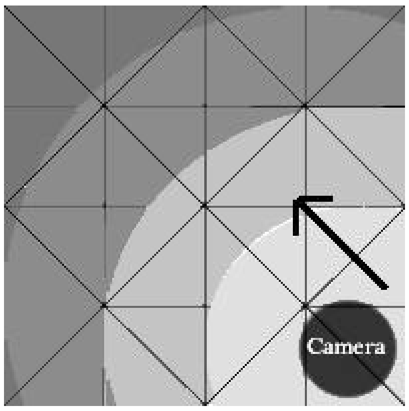
The essential concept of the quad-tree LOD algorithm though is the function which decides when a quad requires more detail, and in what way does the quad tree split to give that new detail. The method by which Rottger's model determines the need for further detail is begun through the expression:

$$L/Q < C$$

In the above expression *L* is the distance from the camera view to the center of the quad; Q is the quad width, and C is some constant intended to control the amount of detail represented in the scene. As the value of C increases, the level of detail increases as the amount of quad's requiring splits increases. Since the change in detail level is on an order of the power of two, an increase in C causes an exponential increase in the level of detail. Within the model of the modified algorithm, the expression above is expressed through the equation:

Split Value = Distance / (Quad Width * Maximum Detail)

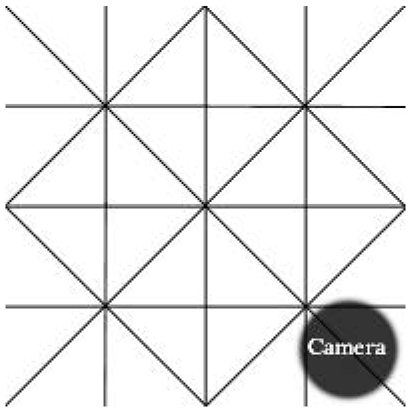
While this equation will be modified slightly in the future, it reflects the division of terrain surrounding the camera view into concentric circles of levels of detail upon which the emphasis of the computation and rendering is placed upon the closest levels, and each outer level reflecting an exponential decay in level of detail. The contents of *Split Value* is a positive real number, and by setting a floor value as the point of splitting, it is possible to know when a quad is in need of further detail. Since each width of a quad is a power of two the denominator in the split equation will always vary by a multiple of two from its higher or lower level of detail equation. This allows a model to be created, within this project, in which values from the split equation below 1.0 are considered in need of further detail, and therefore further split, but otherwise the values will be bound between an upper limit of 2.0 and a lower limit of 1.0. These values, while discarded for the moment, will be useful as data later in this project.

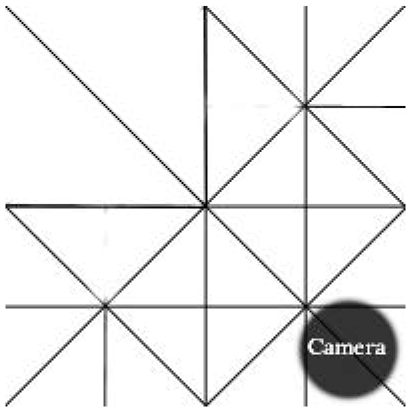


Rings Of Detail

The process within Rottger's model as well as other designs is based upon the principle of rendering the absolute minimum vertices necessary while maintaining a certain resolution and frame rate. This has led the Rottger model to follow the principle of testing for a quad's necessity to split into further detail, and then to test within each child quad whether the split within this quadrant is necessary. This process allows only some of the quads within the parent quad to be split with more detail, which appears at its creation to offer a distinct savings in rendering, but in truth it introduces many additional disadvantages. One disadvantage is that extra distance calculations must occur when the quad is split to decide if the new child quads are within the area deemed necessary for further detail. This forces the child node to decide whether it should split further, represent its new vertices without split, or simply represent the vertices visible by the parent. Further, upon a split, by testing each child to see if it should reflect the new vertices or if it should maintain the detail of the parent, it removes the children's ability to know the detail level of each other, and therefore put that information into saving computation time.

In order to simplify the quad-tree model in order to save computation time, the modified Rottger algorithm uses a winding order quad-tree. This model differs not in its decision to split, but in its action upon splitting. This project's algorithm, upon deciding a quad is in need of further, splits all four quads, without testing if the child should use the parent quad's detail level or its own. This quad is then called recursively to test if more detail is needed. During the process, a winding order, one through four, beginning in the northwest corner quad and winding clockwise is maintained to allow for a node's ability to quickly know the base value of up to 3 neighboring quads. This splitting process is continued until the *split value* is found to be greater than one, or if the minimum quad width is reached- quad width of two.





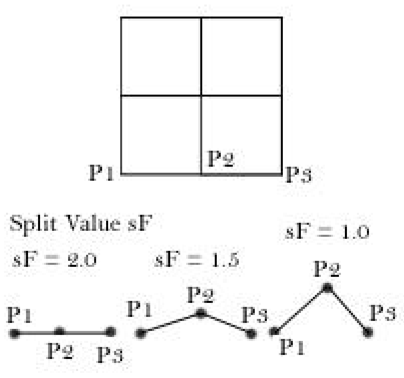
Full Data Rendering

LOD Algorithm Reduction

Within both this project's modified quad-tree algorithm and the model designed by Rottger, the level of detail algorithm creates a mesh in which distinct boundaries exist between the changes in detail. This effect is unnoticeable when rendered from a single view, but when a change of view occurs, or rather movement, detail on the mesh begins to pop in and out of view as the boundary defined by the split function passes across vertices. This function causes data previously unnoticeable to be instantly rendered, causing a popping on the screen as new data is introduced. While the quad-tree algorithm saves computation and rendering time, the popping effect makes the algorithm visually unbearable. This problem is solved through a technique called Geo-Morphing, in which the data points are manipulated to slowly introduce the new detail to the screen. [Rottg98]

The incorporation of Geo-Morphing allows the interpolation of a height value based upon the split function. The problem with rendering without Geo-Morphing, is that when a certain distance threshold is met, at which point the split function returns a value below 1.0, the algorithm then splits the quad into 4 child quads and renders 5 new height points per quad in which 4 of the new points are shared among the quads. These points however, when introduced immediately into the scene will cause a popping effect in which the new data points, if varying by a great amount from its parents' equivalent value. This effect can detract sharply from the illusion of full detail promised by the LOD algorithm. To remedy this problem, Geo-Morphing manages the newly introduced by ensuring that there initial introduction value will not vary from the average of the two points across the line which the new point is introduced. The split function supports this easily by continually producing values within the range of 1.0 - 2.0. By subtracting from these values by 1.0, a weight average can be used on the new point so that at initial introduction, where the split function S is equal to 2.0, the weighted average of point pNew across parent points p1 and p2 would be computed as such:

$$pNewActualHeight = pNew * (1 - (S\text{-}1.0) + (S - 1.0) * (((p1 + p2)/2))$$



This allows for the morphing of an individual vertex in such a way that the new data is dynamically introduced, and therefore appears to slowly grow out of the parent's data points. This technique is truthful to real world observations, in which more and more detail is slowly made visible as the distance of the object becomes closer and closer.

The use of Geo-Morphing however introduces a problem in the interpolation of new points of detail. If two quads are both rendering points at the same level of detail, there will exist a small difference in the values of their respective split functions, so that the interpolated values calculated by each quad for the shared points will be slightly different. This causes cracks in the terrain, which are considered unacceptable as a feature of an LOD algorithm. This causes a revision of both the LOD algorithm implemented within this paper as well as most other LOD algorithms. In order to solve this problem, two passes through the LOD tree is necessary. The first pass, which is considered the tessellation pass, is necessary in order to calculate for each quad the split function value. This initial pass is necessary because each quad node must have during rendering knowledge of both their neighbors as well as their neighbors parents and children. The tessellation pass stores the values of the tree in a 2-dimensional array in which 0 represents empty nodes, -1 represents parent nodes, and nodes containing values between 1.0 and 2.0 represent leaf nodes. In order to keep cracks from appearing from Geo-Morphing, when a node draws its triangle fans, it guarantees it will use the greater split value between itself and the neighboring node which it shares the point with. The rendering process of a node is therefore defined as such:

A. Include Center, Northwest, Northeast, Southwest, and Southeast data points in triangle fan.

B. Include North, West, East, and South data points if the respective neighbor is of an equal or greater level of detail.

C. For each side, if the neighbor is of equal level of detail, use the greater of the two split function values to ensure continuity.

D. For each side, if the neighbor is of a lesser level of detail, ensure that corner point of the node which is the middle point of the neighbor node is computed by using the split function of the neighbor

The necessity of knowing the level of detail of a node's neighbor is therefore critical during the rendering process. This necessity is simplified and optimized by altering Rottger's algorithm by splitting a quad into 4 individual quads when the split value is dropped below the threshold. So within the model developed within this paper, a node always has critical information always about three of its neighbors because it knows that its siblings were split as well. This becomes important when features such as texturing, primitive coloring, and lighting becomes important facets in the model design.

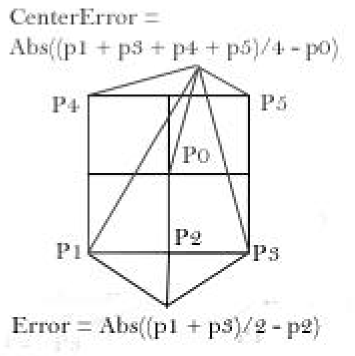
One factor introduced into the concept of LOD algorithm for which Geo-Morphing cannot solve is the case of extreme changes in height values within the height field. This can observed in the case where a quad's split value has recently fallen under the split value threshold, and in which the new data points do not yet change the shape of the surface. The problem however is that for larger and larger differences between the averaged value of the parent nodes and the final value for the new node, the greater the changes will appear for each step closer. So while Geo-morphing works well for introducing slight variations into the data field, it cannot handle extreme aberrations from the normal without compromising the guarantee of a fully detailed and realistic view.

The problem of extreme aberrations is specific to the type of data with the field. For natural representations of most geography, extreme changes in height do not normally exist. Even within such objects as waterfalls, cliffs, mountainsides, these drop offs generally only occur over many quad regions. Only in special conditions such as caves with spikes or some other unique case would a great extreme of height different would be found. However, in representations of data fields in which such constructs exist, the use of error calculations can reduce the effects of extreme height changes in those areas needed while minimizing the need for further detail in areas in which there exists little height variation.

The incorporation of error calculations is very easily done within both the Rottger model as well as the model designed within this paper. Error values are calculated in a 2-dimensional array in which each nodes error value is equal to the maximum error of its own node or one of its children. The error is calculated on the side and center of a quad using the equations:

$$\mathit{SideNodeError} = \mathbf{Abs}((\mathit{CornerHeight1} + \mathit{CornerHeight2})/2 - \mathit{nodeHeight})$$

CenterNodeError= **Abs**((*CornerHeight1* + *CornerHeight2* + *CornerHeight3* + *CornerHeight4*)/4 - *nodeHeight*)


CenterError =
Abs((p1 + p3 + p4 + p5)/4 - p0)
P4
P5
P0
P1
P2
P3
Error = Abs((p1 + p3)/2 - p2)

The maximum of these errors is then compared to the errors of the children. The total maximum is then recorded in the error field and is returned to the parent node for comparison. The distinction between most quad-tree algorithms and the one presented within this paper is that because of the application of the quad-tree model in the realm of realistic 3D terrain, it is not necessary to include error values when computing the level of detail of a quad. In circumstances that would necessitate error inclusion, the error calculations would be included in the split function through the multiplication of the denominator by the error value. Therefore by assigning a value between .1 and some maximum height difference, the split value can be modified to either draw more or less detail based upon the error condition. Error values in which there exist small error will naturally disallow further splitting of the quad because of the wastefullness of further calculations for only minor detail. The split function is therefor modified to incorporate error through the function:

*Split Value = Distance / (Quad Width * Maximum Detail * Node Error)*

Within the LOD model described in this paper, the error value has been minimized because of its lack of accuracy to described the uniqueness of the new data point introduced. This is realized because of the incorporation of both primitive coloring as well as lighting effects. Therefore while the previous computation to find the error involved in introducing a new data point provided an optimized interpretation of height data, it does not take into consideration the possible introduction of lighting and material effects or primitive coloring that would have provided further detail. Due to this lack of incorporation, it seems reasonable to further revise the split function so that it only offers more detail and never attempts to remove detail from a quad. This obviously means a higher number of triangle fans being drawn, but is a necessary step so that important detail in lighting and colors are not removed permanently. This newly revised function can be expressed as:

*Split Value = Distance / (Quad Width * Maximum Detail * (1 + Node Error))*

The use of textured primitives within the LOD algorithm can only supply a minimal amount of variation across the terrain. Even the use of multiple textures offers only a small number of possibilities when attempting to create unique and genuine terrain. The terrain transitions across different textures also create jagged boundaries from which the illusion of reality is broken. Multi-texturing, or the use of multiple passes to create primitives mixed with various textures still only offers a limited number of variations as well as causes severe performance loss. To combat this, a technique known as Geo-mipmapping is used to create a realistic lighting and color components to offer an almost unlimited combination of terrain types based on only one texture. The integration of color is used through the creation of a color map that is of the same size as the height map. Desired colors are therefore filled in to represent the expected colors as various height points. These colors are then incorporated and blended against the texture to create fluid and seamless transitions between different mediums of terrain. Lighting normals are also computed and used in a similar fashion to create realistic lighting effects based upon their angle towards the light source. Both color and lighting though suffer from the same problem of height points in LOD algorithms in that the transition of one level of detail causes not only the sudden appearance of new "physical terrain," but now also the appearance of new colors and shades of lighting.

These problems are remedied within this project in a similar fashion as height points, in that their values are weighted against the average of existing points, so that their new detail is slowly introduced into the scene. Lighting however suffers from the problem that the vertex normals in the field computed, as being the average of the face normals of the 8 triangles sharing the vertex, will incorrectly represent the lighting normals of a single vertex, based upon the nearest 8 points, instead of the computed normal based on the 8 points that the triangle fan will actually draw. The color and height components do not suffer from this problem because the height and color values at each vertex are static no matter what size the quad is, but with the normal component, each vertex will have a different normal based upon its current level of detail. This therefore makes it necessary for either (a) compute the normals dynamically, or (b) create multiple normal maps of the vertex normals at each level of detail. Since it is extremely costly to compute normals during rendering, it then is most efficent to create multiple maps of the same board, but with the vertex normals computed based upon their level of detail. This method is natively used by OpenGL and other graphics API's for producing better texture maps, and so it is used in this project to correctly incorporate lighting into the project. The correct normals, like the color and height components, will be averaged against pre-existing normal values in order to slowly introduce the new normal values into the scene.

All current LOD algorithms suffer from limitations either through computation requirements or memory requirements. The quad tree algorithm is limited in the depth of its detail by the amount of values stored within the height field. While creating a limitation in the depth of the height field, it offers very little limitation in way of memory requirements as well as processing requirements. Since the LOD algorithm ensures that objects of a certain distance away will be rendered and computed in less and less detail, height maps of greater and greater size will not strain the algorithm's speed, because of its ability to disregard the detail of distant objects. If the transition from one level of detail to the next higher level produces an exponential increase in detail, it is fair to say that for each level further introduced into the map will have a total rendering cost that is exponentially less than the previously lowest level. The LOD algorithm’s maximum detail however creates a barrier that is insurmountable within this model. If it were possible to change the source of data however from a height map in memory to function that calculated the points during run-time, it would in theory be possible to have infinite detail. Such functions as sine waves and noise functions have been sources for height values in other models. Other mathematical models however, such as fractal geometry, can offer the possibility of creating height maps that appear to have a natural terrain flow. Fractals also have the property of self-similarity, and therefore could be used to divine deeper and deeper detail within the scene. This type of infinite detail however would require that there be no memory limitations, which would demand that for each quad being rendered, it would have to compute the tessellation of its neighbors in order to have knowledge of their level of detail. This would be a computational strain on the processor.

The modifications within this project's algorithm as opposed to other LOD algorithms currently available is the move towards simplicity in design in order to stream-line lighting, coloring, and other effects calculations. The winding order design allows knowledge of neighbors' detail, and therefore allows the optimization of computations. The integration of lighting and primitive coloring through geo-mipmapping allows for the ability to slowly introduce new data, whether it is height points, color values, or shadows, into the visual experience. Various optimizations can still be incorporated into the LOD model, including storing calculated height points in a temporary array. The creation of even the most inefficient of LOD models however far surpasses a complete rendering of the data points as the size of the height map increases. Within a model created for this project, a non-LOD algorithm renders approximately 500,000 triangles within the frustum at under 1 fps, while the LOD algorithm rendered a similar scene with similar detail with only 1,000 triangles at 50 fps. This type of reduction is a major improvement in managing data. The model proposed within this paper allows for the rendering of world limited only by memory available. This allows for the bulk of the computational time to be reserved for other application related tasks such as sound, artificial intelligence, and physics. The stream-lining of terrain representation is essential if a realistic 3D world is ever attempted.

References

[Linds96] Lindstrom et al., Real- Time Continuous Level of Detail Rendering of Height Fields, *Proceedings of SIGGRAPH, 1996*

.

[Rottg98] Rottger et al., Real-Time Generation of Continuous Levels of Detail for Height Fields, *Proceedings of SIGGRAPH, 1998.*

[Eberl99] Eberly, D. *3D Game Engine Design*. *San Diego:* Academic Press, 1999.

[Savch00] Savchenko, S. *3D Graphics Programming Games and Beyond. Indianapolis:* Sams Publishing, 2000.

[Morle00] Morley, M. *Frustum Culling in OpenGL*. Http://www.markmorley.com/opengl/frustumculling.html