Multi-Resolution Deformation in Out-of-Core Terrain Rendering

William E. Brandstetter III[1,2] Joseph D. Mahsman[1] Cody J. White[1]
Sergiu M. Dascalu[1] Frederick C. Harris, Jr.[1,2]

Computer Science and Engr.[1] CAVCaM[2]
University of Nevada, Reno Desert Research Institute
Reno, NV 89557 Reno, NV 89512
{brandste,mahsman,cjwhite,dascalus,Fred.Harris}@cse.unr.edu

Abstract

Large scale terrain rendering in real-time is a well known problem across the computer graphics community which has garnered many solutions relying on dynamic level of detail changes to the terrain. These algorithms typically fit into two categories: in-core and out-of-core. Out-of-core algorithms usually require data to remain static, thus disallowing terrain modification whereas in-core algorithms allow for deformation, but usually require updating of modified data through a data hierarchy which can potentially be a slow process. This paper presents a solution for out-of-core deformable terrain rendering that works in real-time. Since the requirements of in-core deformable terrain do not scale to an out-of-core system, the need for data propagation and recalculation of error bounds has been eliminated.

Keywords: level-of-detail (LOD), deformable

1 Introduction

Terrain rendering is a highly researched area due to demand from the military, scientific visualization, and computer gaming communities. Even as advances in graphics hardware continue to be released, these applications will always push the current technology to the limit such that a brute force method will never be practical. Level-of-detail (LOD) rendering algorithms are one of the applications which continue to be developed to give the best visual representation of large-scale landscapes in real-time.

The size of datasets is one of the major problems in terrain rendering. First, brute force rendering is not an option when dealing with large datasets, so a LOD approach needs to be taken. Second, given a large heightmap, quite a bit of memory can be taken up and thus out-of-core (outside of system memory) rendering needs to be supported. The most common approach is to extract a good view-dependent approximation of the mesh in real-time. This is accomplished by storing data in a specific hierarchical structure, in which terrain can usually be categorized. Terrain can be represented in many different data structures such as a triangulated irregular mesh (TIN) [7], which gives the best approximation, a regular grid, which uses somewhat more triangles to represent a surface, quadtrees [14], binary triangle trees [4], or directed acyclic graphs [9].

Refinement may take place on a per-triangle basis, or tessellate aggregates of polygons. Some existing algorithms refine the terrain every frame, having a "split-only" approach. Others may merge and split from previous frames' work. Refinement can be accomplished using a nested-error bound metric, or as in [10] solely the viewing position. Some terrain algorithms only support in-core (inside of system memory) [4], while others support out-of-core rendering [9] and dynamic addition of procedural detail [11].

Since terrain data can consume such a large memory footprint, out-of-core algorithms often limit their datasets to be static (unchanging). Large amounts of terrain data are usually processed in a way that leaves the geometry optimal for video hardware and is not expected to ever change. When dealing with an out-of-core terrain system that handles dynamic updates of its height values, things aren't so trivial. For the most part, the areas of the mesh that need to be rendered stay in memory, while areas that aren't visible can be discarded to the hard-drive until needed. With deformable terrain, updates to the mesh could be made outside the viewing frustum, in which case those areas would need to be loaded, updated, and cached back to disk. If a hierarchy of LOD mesh representations were preprocessed, then updated data may need to be propagated up through the tree or reprocessed altogether. The idea of dealing with large amounts of data in a dynamic terrain algorithm can quickly become unmanageable, thus when combined with the first problem, a second problem of terrain rendering is presented: dynamic terrain. Therefore, presented here is an out-of-core terrain algorithm which supports dynamic updates to the heightfield in real-time, allowing for deformable terrain.

Contribution: We present an out-of-core terrain LOD algorithm that supports real-time deformation, building upon the features of several algorithms discussed in Section 2. [11] presents a LOD algorithm that combines the quadtree structure of [16] with the detail addition properties of [3]. We adapt this data structure to support terrain deformation at any resolution throughout the terrain hierarchy with large-scale data stored out-of-core.

Overview: After the previous work (Section 2), this paper is arranged as follows: A quadtree is constructed using a coarse-grained simplification (Section 3.1). During runtime, the refinement algorithm determines the currently active regions of the terrain based on view-dependent and deformation refinement criteria (Section 3.2). A separate thread is responsible for loading and writing out-of-core portions of the quadtree (Section 3.3). Finally, a brush is used to select a rectangular region of the terrain at a particular resolution to be deformed (Section 3.4), possibly extending the quadtree by adding levels. The new data is treated the same as the rest of the terrain mesh, and is paged in and out of main memory as needed. Finally, we present out texturing scheme (Section 3.5).

2 Related Work

2.1 ROAM

ROAMing Terrain: Real-time Optimally Adapting Meshes [4] is a well known level of detail algorithm utilizing a binary triangle tree (bintree) which stores all of the triangles for a given mesh. Instead of dealing with a complete terrain system that performs out-of-core paging for geometry, textures, and selection of LOD blocks, the authors focus on in-core geometry management. Given a bintree, split and merge operations are performed using a dual priority-queue system to achieve a LOD representation for the underlying data.

ROAM starts with a preprocessing step that produces a nested view-independent error-bounds that works along side the bintree. When deciding to split or merge a specific triangle in a bintree, the pre-computed error bound is taken into consideration along with the view-dependent metric. The algorithm uses a metric based on nested world-space bounds, where a world-space volume (called a wedgie) contains the points of the triangle. World-space bounds are computed bottom-up, such that a node's error-bounds is the maximum of its children's world-space bounds.

This algorithm also realizes that neighboring triangles could be at different resolutions, either coarser or finer by one level. In this case, before a split is made, neighboring triangles may be force-split to eliminate cracks or T-junctions in the mesh. This is done recursively until the base neighbor is at the same resolution level (i.e. a diamond is created). Doing this recursive step ensures a single continuous mesh.

Top-down refinement of a terrain mesh is a simple and widely used concept where detail resolution can be added easily by extending the leaf nodes of the binary triangle tree with some adjustments to the nested error-bounds. The authors state that ROAM is suitable for dynamic terrain since the preprocessing of error-bounds computation is localized and fast. However, the algorithm only handles data that can fit into system memory. Reprocessing large amounts (more than can fit into memory) of terrain data is unacceptable for extremely large datasets, especially if many deformations are occurring and requiring error-bounds to be recomputed every frame.

2.2 Geomipmapping

With advances in graphics hardware, it is common to spend less work on the CPU to find a "perfect" mesh and send more triangles to the GPU, even if they aren't needed. Since sometimes it is faster (and easier) to render a triangle than determine if it should be culled, there is a balance between brute force and dynamic refinement algorithms. In 2000, de Boer wrote the paper *Fast Terrain Rendering Using Geometrical MipMapping* [3], a new approach that exploits graphics hardware instead of computing perfect tessellation on the CPU. De Boer states that the goal is to send as many triangles to the hardware as it can handle. Since terrain data can be represented as a 2-dimensional heightmap, the analogy of texture mipmapping was used and applied to geometry.

Geomipmapping makes use of a regular grid of evenly spaced height values, that must have $2^N + 1$ samples on each side. A preprocessing step is performed that cuts the terrain into blocks, called GeoMipMaps, also with $2^N + 1$ vertices on each side (e.g. a 257×257 regular grid may be divided into 16×16 blocks of 17×17 vertices). Vertices on the edge are duplicated for each block where each block is given a bounding box and is suitable to be stored in a quadtree for quick frustum culling. Finally, a series of mipmaps are created by simplifying the mesh which is done by removing every other row and column vertex. The author suggests that out-of-core rendering could be supported by having only visible blocks or those near the camera in memory while others can be discarded to the hard disk until needed.

Each geomipmap level has an associated geometrical error. For each vertex removed during the simplification step, a world space error is calculated as the distance between the vertex and the line of the interpolated simplified mesh. The maximum error of all vertices is assigned as the geometrical error to the block. Then, to decide which geomipmap to use, it is projected to screen pixel space and compared to a user-defined threshold. When the current geometrical error is too high, a higher detailed block is used.

After each geomipmap block has been chosen, there will be several neighboring blocks that reside at a different LOD. As such, cracks will appear between these blocks since one patch holds more detail than the other. De Boer fixes this problem by omitting vertices on the edge of a higher detailed block to identically match its lower detailed neighbor.

This algorithm is extremely easy to understand, implement, and also exploits the benefits of the graphics hardware. Adding detail is trivial by simply reversing the simplification step described in the algorithm. Deformation could be supported, but geomipmaps would have to be recreated and geometrical errors recalculated, which could hinder real-time deformation. The downside is that the number of geomipmaps increases quadratically (N^2) based on the size of the terrain; therefore, possibly resulting in slow computation and rendering.

2.3 Chunked LOD

At *SIGGRAPH'02* Ulrich presented a hardware friendly algorithm based on the concept of a chunked quadtree, which is described in [16]. This algorithm, also referred to as Chunked LOD, is somewhat similar to GeoMipMapping; however, it scales much better due to the quadtree structure. There is often confusion of the differences between Chunked LOD and Geomipmapping since the algorithms are similar. However, Chunked LOD exploits a quadtree data structure of mipmapped geometry. Therefore the number of rendered nodes does not quadratically increase due to the size of the terrain.

A requirement of this algorithm is to have a view-dependent LOD algorithm that refines aggregates of polygons, instead of individual polygons. As ROAM tessellates down to a single triangle, Chunked LOD refines chunks of geometry that have been preprocessed using a view-independent metric. Since chunks are stored in a quadtree, the root node is stored as a very low polygon representation of the entire terrain. Every node can be split recursively into four children, where each child represents a quadrant of the terrain at higher detail than its parent. Every node is referred to as a chunk, and can be rendered independent of any other node in the quadtree. Having such a feature allows for easy out-of-core support.

In order to create the chunked quadtree, a non-trivial preprocessing step must first be performed. Given a large heightmap dataset, height samples are partitioned into a quadtree and simplified based on the properties of the mesh and not the viewer. This can be done using any per-triangle tessellation algorithm, such as binary tree tessellation as illustrated in ROAM [4]. Depending on the depth of the chunk, more detail is given to the final mesh.

Each chunk holds a list of renderable vertices, a bounding volume, and a maximum geometrical error. Starting with the root node of the quadtree, nodes are culled and recursively split based on the viewing position and its geometrical error. Neighboring chunks at different levels of detail are addressed by creating a skirt of extra geometry that eventually, with tweaks of texture coordinates, fills in the frame buffer so no artifacts are noticed. Utilizing skirts keeps chunks independent of each other, which makes out-of-core support trivial.

The Chunked LOD quadtree structure is one of the best known hardware friendly LOD algorithms since it can be utilized for very large out-of-core terrain. Adding detail resolution requires extending the chunked quadtree, which could be easily done in a preprocessing step. However, deformation isn't trivial since the algorithm requires a static mesh; if any height samples were changed, it would require reprocessing the entire quadtree, which is unacceptable for real-time deformation.

3 Proposed Approach

The following sections present our out-of-core deformable terrain algorithm for preprocessing and rendering of large-scale terrain datasets. We start with an overview of the hierarchical representation of the terrain data and then describe the runtime algorithm for mesh refinement, rendering, memory management, and deformation. Figure 1 illustrates the flow of data through the system from program initialization to rendering.

3.1 Hierarchical Representation

The hierarchical representation of the original mesh is built during a preprocessing step. For a $n \times n$ input mesh, a quadtree is used to organize the data such that the root node defines a low-detail representation of the entire mesh. Each subsequent child contains more detail at the scale of one quarter of its parent's mesh, while the leaf nodes constitute the original mesh. Every node is of size $m \times m$, and therefore each node uses the same amount of vertices. The dimensions n of the input mesh and m of the nodes must be one greater than a power of two to allow for optimization of the construction and refinement algorithms.

The quadtree is constructed using a simplification process similar to [8]. First, the input mesh is partitioned into leaf nodes of size $m \times m$, where each node overlaps neighboring nodes by one row and one column. Nodes are combined into 2×2 blocks and upsampled by removing every other row and column vertex. This is repeated recursively until 2×2 blocks

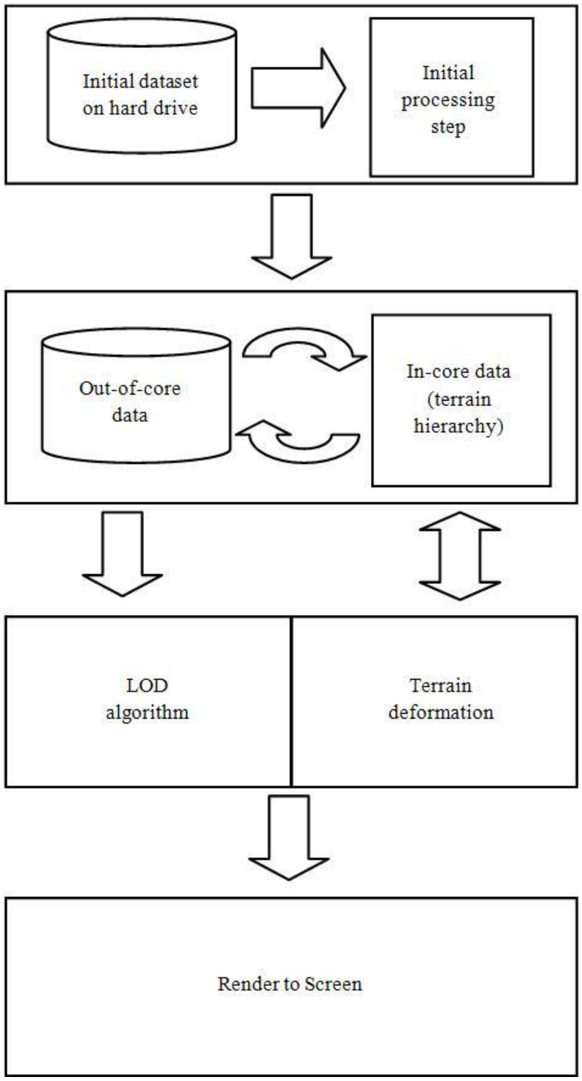
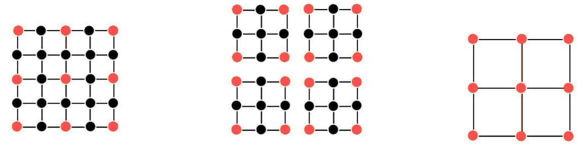


Figure 1: Block diagram demonstrating the flow of data through the system.

can no longer be made. Each node is given a bounding box that encapsulates the entire mesh, as shown in Figure 2.

The process of removing every other row and column vertex when creating parent nodes implies that the data for each node, except for the root, comprises its parent's data (the excluded rows and columns) and its own data. During terrain deformation, this property obviates the need to propagate changes through the tree. In addition to a node's individual data, it contains pointers to its parent's data. To guarantee this property holds true, when a node is loaded into memory all of its ancestors must be in memory as well. The memory layout for any given node is shown in Figure 3.

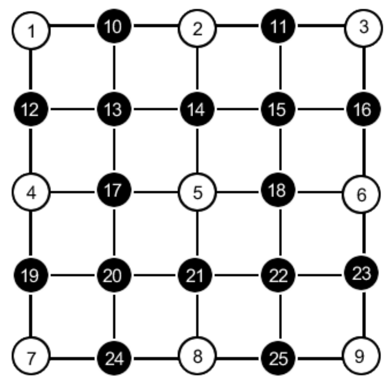
For example, the bottom left vertex of an underlying heightfield belongs to the root node. Child nodes receive a pointer to this vertex in order to access it. This



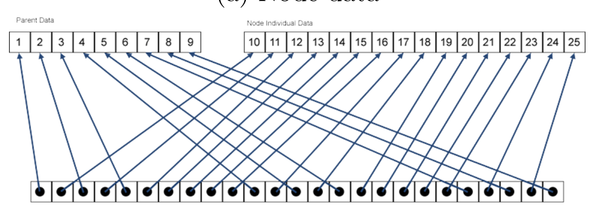
(a) Original Mesh (b) Child Nodes (c) Parent Node

Figure 2: The simplification process to create a quadtree. The original mesh (a) is separated into $m \times m$ nodes ($m = 5$) (b), where each 2×2 block creates a parent node (c).

is similar to the wavelet compression scheme from [2]. However, we do not encode the child data within the parent's node. Instead the individual data for each node is stored in a file that can be loaded on demand. This eliminates the need to decode node information at runtime and allows for deformation without encoding new vertices into the quadtree. In order to query a node's data, it must simply dereference the vertices it points to.



(a) Node data



(b) Memory layout

Figure 3: Node data shown in (a) represents the memory layout shown in (b).

3.2 Runtime Algorithm

3.2.1 Mesh Refinement

The goal of any terrain rendering algorithm is to quickly create the best approximation mesh for each frame. Our approach uses a split-only top-down refinement. Previous algorithms use properties of the underlying geometry (e.g. nested error bounds) during refinement as in [16] and [3]. Deformation of the terrain requires recalculation and propagation of these properties throughout the tree. We take an approach similar to [10] and use only the view position and frustum as refinement criteria. Although this approach looks awkward for high-frequency data (e.g. a steep mountain consisting of a few vertices), natural terrain datasets often feature a smooth gradient.

Refinement begins at the root node and proceeds recursively for each child node. A breadth-first traversal is required for linking neighboring nodes. For every node, if the node's bounding box is inside the view frustum and the center of the bounding box is closer than a predefined threshold, the node is refined by traversing its four children, otherwise it is prepared for rendering. A threshold should be chosen such that a nested regular grid surrounds the viewer. Since no other metrics are taken into account during refinement, this will yield the best visual fidelity. Note that the LOD of neighboring nodes is never limited, as in a restricted quadtree where nodes are forced to split based on the level of its neighbors as in [13]. Figure 4 shows bounding boxes of the hierarchy.

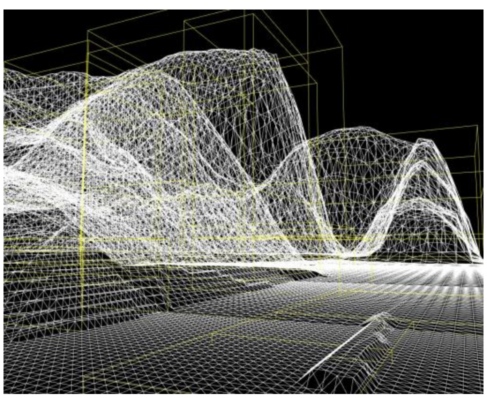


Figure 4: A quadtree displaying the bounding boxes of each node. Bounding boxes are culled against the viewing frustum to quickly eliminate nodes during refinement.

3.2.2 Neighboring Nodes

Smooth transitions between nodes of different LOD must be rendered correctly, otherwise seams will be visible due to gaps in the rendered mesh or inconsistent shading from incorrect normal calculations. Also, since each neighbor holds its own copy of edge vertices, care must be taken while deforming edges or edge boundaries. To handle these variations, a node must be aware of its neighbors. Since quadtree refinement isn't restricted, the difference between two nodes may be one level or more. Since a node's LOD may change from frame to frame, neighboring links are recreated during refinement.

When linking nodes together, a node is only allowed to point to a neighbor of equal level or higher. Enforcing this rule allows each node to store no more than four neighbor references. When a node is split during refinement, the node is responsible for updating its children with the correct neighborhood information. This cannot be accomplished with a depth-first traversal, commonly used in LOD algorithms. Instead, a breadth-first traversal is performed.

Neighbor links play an important role for correct normal calculation. Normals are needed to simulate a realistic lighting model, and can also be used for collision response. The biggest problem of normal calculation presents itself on the seams of terrain patches. Vertices on an edge need the height values of neighboring nodes.

The most common approach to calculate a normal is to compute a normal for each vertex in the heightfield by taking the average normal of all faces that contain the vertex [17]. This process consists of several costly mathematical operations, such as square roots. Several optimizations can be made by exploiting properties of the heightfield. The method we use is described in [15], which only requires the four neighboring heightsamples of a vertex. In order to create a smooth transition across a patch seam, neighboring vertices must be queried and the computed normal is then stored for each edge.

3.2.3 Detail Addition

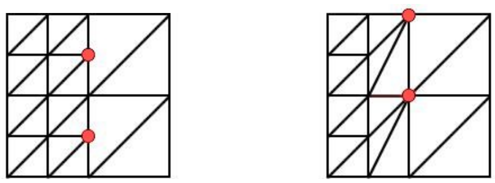
To improve the appearance of the terrain without increasing the size of the data on disk, procedural detail is added at runtime for leaf nodes that meet the refinement criteria. The detail is added to the hierarchy in the form of new leaf nodes that extend the quadtree until a user-specified level is reached. When creating a new node, a reverse process of adding rows and columns is performed and the new node is linked to its parent, which was previously a leaf node. The new vertices are then assigned procedural data.

Linear interpolation is not sufficient for creating additional detail because the resulting data is uniform. Instead, fractals are used to give the data a non-uniform appearance. Each interpolated vertex is shifted a random amount such that it stays within the bounds of the surrounding vertices. Since detail addition is subtle, the process does not need to be deterministic, therefore detail can be randomized each time it is created.

3.2.4 Rendering

The result of refinement is a list of patches to be rendered. Before rendering, indices can be recalculated for nodes whose neighbor's LOD have changed and normals can be recalculated if deformation had occurred. Each node must then dereference its pointer data to create a vertex list. Finally, each node can be transformed into world space and the data sent across the bus to be rendered. The rendering process is decoupled from the updating and disk I/O methods, allowing for smooth loads of data and no hiccups in the system regardless of how fast the viewer is moving around the terrain.

Stitching is accomplished by having the finer detail node omit vertices on its edge to match that of its coarser neighbor. This is done by rendering degenerate triangles. Geometrical skirts [16] were not chosen since the size of the skirt may change after deformation. Recalculation of the skirt can become tedious and slow. Figure 5 illustrates the removal of T-junctions by utilizing degenerate triangles.



(a) T-Junction (b) Degenerate Triangles

Figure 5: T-Junctions appear at the neighboring nodes of different levels of detail. Omitting vertices via degenerate triangles removes any possible cracks from the mesh.

3.3 Memory Management

Our algorithm supports large datasets stored out of core, i.e. data that resides outside of main memory [11]. A separate loading and caching thread is fed patches to load or write to disk. The patches to load are based on refinement, while the patches to write are based on a least recently used (LRU) algorithm.

During refinement, if a parent cannot be split because the data for its children is not in-core, a request for the data is made to the loading and caching thread. The system never stops to wait for data to load; until the data for the children is loaded, the parent's data is rendered.

When the memory footprint exceeds a predefined threshold, LRU patches are discarded to disk until the used memory falls below the threshold. Each node is given a timestamp representing the last time the node was either rendered or deformed. A priority queue is used to efficiently determine which nodes should be discarded. Since every node relies on its parent for some of its data, discarding a parent to disk will invalidate memory references for its children, therefore only leaf nodes of the currently refined mesh are considered for caching.

Depending on the actions of the user (fast movement or several deformations) and the current memory footprint, nodes may require continuous allocation and deallocation. Instead of using operators such as new or delete which are notoriously slow for small and frequent allocations, a freelist is used as in [6].

3.4 Deformation

Real-time modifications are applied to the terrain by refining the currently active mesh based on a rectangular selection of the terrain, called a brush, in addition to the view-dependent refinement criteria described earlier. The vertex data for each refined node is modified to fit the brush specification.

A brush defines the rectangular extent (defined by position, width, and height) and the resolution of deformation (defined by a level in the hierarchy, which may not exist). In addition, a brush holds an array of pointers to vertices in the terrain, allowing deformations to cross node boundaries. Nodes that intersect the brush are selected during refinement and vertices from each node are given to the brush. Dereferencing the brush gives access to vertex data which can be overwritten with new data. Since vertices on edges are duplicated for each patch, care must be taken for deformations across boundaries by syncing adjacent vertices. This is done in a pre-rendering step that compares dirty flags of neighboring nodes in the quadtree.

Refinement is based on brush extent and resolution as well as view-dependent criteria. Therefore, a node may be refined even though it is not sufficiently close to the viewer or inside the view frustum. Depending on the resolution of the brush, data for nodes deep into the hierarchy may be requested for loading. Only when all of the data required by the brush's resolution has been loaded can deformation be applied.

As described in Section 3.2.3, procedural detail is added for leaf nodes that meet the view-dependent refinement criteria. If a brush alters a node with procedural data, disk space is allocated for the node and it is allowed to be discarded to disk by the memory manager.

When a node is chosen for rendering, it is possible that an ancestor has previously been deformed. Time stamps are compared, and if a node's last modification is older than its parent's, its data is adapted to the parent mesh by creating procedural detail.

3.5 Texturing

Textures are processed similarly to the terrain data. A large texture can be cut into user-defined partitions and merged into 2×2 blocks before being mipmapped. This process continues until an entire quadtree is built over the original texture data.

Nodes in the terrain quadtree directly map to nodes in the texture quadtree. However, with modifications to the terrain quadtree (deformation and procedural detail), it becomes impractical to create a texture quadtree of the same depth. If a node in the terrain quadtree cannot be mapped directly to a node in the texture quadtree, the parent's texture and texture coordinates are used. When a node is being loaded or deleted it can also load or delete its texture.

Just as the terrain quadtree presented issues at seams, so does the texture quadtree. This is due to the kind of texture filtering used to generate the texture quadtree. Although no seams are visible with nearest filtering, this type of filtering is not visually appealing. With linear filtering, seams appear at the texture edges because the edge texels are not being blended with the correct neighbor texel. This is solved by overlapping adjacent textures during texture quadtree construction such that neighboring nodes have exact texels on shared edges. Clamping the texture edges during rendering causes these texels to blend, removing the seam. There is no perfect solution, and the amount of pixels to overlap can vary.

3.6 Supporting Large Scenes

Extending terrain detail causes the terrain as a whole to be resized to a different resolution. Instead of having a 1-meter resolution map, detail can be added so now its a 1-millimeter map. In order to properly represent detail, the entire terrain needs to be increased in scale.

As worlds get substantially bigger, the amount of precision to reprensent world coordinates gets smaller and smaller. For example, if units represent meters and our world is 100km square, at the farther corner of the world a 32-bit number will allow us to represent 7.8mm granularity [5]. The larger the world coordinates, the less accurate they will be at the farthest extent. Therefore floats usually have to be converted to doubles since there are not enough bits to represent a large number with high precision.

Unfortunately the graphics hardware only performs floating point operations, so precision is lost during operations such as matrix transformations. Traveling to the farthest extent of your terrain will result in seams between patches or jittering movement when the camera moves. It is best to partition the world into a user defined segment space, where each node belongs to a segment and is given an offset.

Instead of transforming the camera and translating patches during rendering, thus losing precision, transforms can be made in segment space, such that the world now revolves around the camera within a segment. This will solve any and all accuracy problems that large scenes come with. Unfortunately, this doesnt come without a burden. If this method is used, then all objects need to be represented in segment space which can prove difficult, especially when using a scenegraph.

Along with large world coordinates, precision is also an issue with the depth buffer. Ideally the user will want to see detail an inch from his nose while also seeing the moon in the sky. Unfortunately a 24 or 32-bit depth buffer cant support such a task. Since the depth buffer is not linear, there are many more bits allocated to the precision of objects closer to the camera than further. Precision is gained exponentially as you push out the near clipping plane. However, in our terrain we will sometimes want to view millimeter resolution, without culling the mountiains in the horizon.

Several methods can fix this problem, such as using imposters [12], or a multipass rendering system that renders the scene in sections, clearing the depth buffer while altering the frustum each time. In our implementation we allow for multiple rendering passes. Table 1 shows the steps required to perform such a task.

1. Set zFar to the maximum value needed.
2. Set zNear = zFar / ratio.
3. Clear z-buffer.
4. Render scene and cull to the adjusted frustum.
5. Set zFar = zNear.
6. If zNear isnt close enough yet go back to Step 2.

Table 1: Process for multipass rendering

4 Results

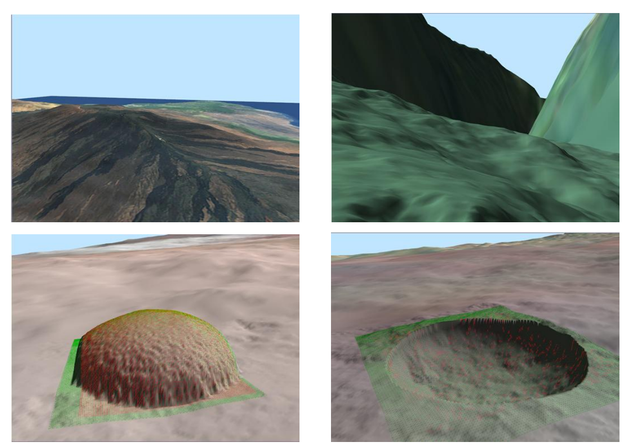


Figure 6: Screenshots from the visualization of the Hawaii dataset.

The following tests were performed on a machine with an Intel Core2 Quad Q9450 processor with 8GB of DDR2 RAM and a NVIDIA GeForce GTX 275 under Windows XP Service Pack 2, which can only utilize 3.5GB of RAM.

The data used for these results was obtained from [1], which holds 10-meter elevation data of the big island of Hawaii along with a 4096×4096 texture. The terrain file has dimensions of 8193×8193 and was already

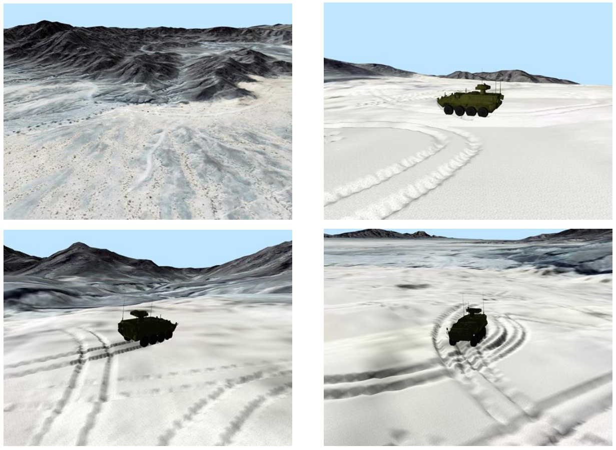


Figure 7: Screenshots from the visualization of Yuma, Arizona with tire track deformation.

in binary terrain (.bt) format. It was first converted into the internal .ter file and the texture image (.jpg) was converted into a .tex file. Since these files were so small, the preprocessing took less than five minutes.

With this application, the user is able to move around the scene via keyboard and mouse input. By clicking and dragging the mouse, the user can select a single axis-aligned brush, and change the resolution of that brush via keyboard input. Once a brush is selected with the desired resolution, the user may create a hill or crater by raising or lowering the terrain. Any changes to the terrain are automatically saved to the .ter file and will be loaded back in when the application restarts. Figure 6 shows screen shots of this application.

In order to determine how well this algorithm runs, we ran various operations of it as illustrated in Table 2 with a frame buffer size of 1024×768. The file tested was a ten meter resolution digital elevation map (DEM) of Hawaii of raster size 8193×8193 which can be freely downloaded over the Internet [1]. The first test was to simply move over the terrain with no deformation occurring. This tested the LOD refinement algorithm used to render the terrain in real-time. The next section of results in the table show the speeds of deformation of the terrain in terms of frames-per-second. Using different brush sizes, we deformed the terrain over the same part of the dataset. For all of the brush sizes used, the algorithm demonstrated interactive framerates. The largest brush size used exhibited a relatively low framerate due to the increased amount of refining of the mesh down to the deepest parts of the terrain hierarchy, which can be considered a worst-case scenario.

Another application of the algorithm has been used for tire track deformation from a military vehicle navigating the terrain in a dataset from Yuma Proving Ground, an Army installation in Arizona. Screenshots are shown in Figure 7, and an example can be seen in

Operation	**FPS**
Arbitrarily moving over the dataset	48.41
Deformation with brush of size 32×32	38.37
Deformation with brush of size 64×64	23.54
Deformation with brush of size 128×128	10.96

Table 2: Average frames per second over different operations

the accompanying video.

We were unable to compare these results to any prior terrain rendering algorithms because no algorithm supports deformation out-of-core.

5 Conclusions

We have presented a complete LOD terrain algorithm including the major features of deformation and out-of-core rendering. To the best of our knowledge, this is the first out-of-core deformable terrain algorithm. Refinement is not only based upon the viewing frustum, but also takes into account the selected deformation brushes. This allows data that is not being viewed to remain in memory and be subject to deformation. Previous methods that allow out-of-core rendering usually preprocess the geometry into a triangulated irregular mesh for optimal polygon throughput, and require that the terrain mesh remain static. Other in-core algorithms support changes to the underlying heightmap, but need to recalculate and propagate nested error-bounds through a hierarchal structure. Our approach eliminates the need for any geometry tessellation or propagation after a modification to the terrain heightmap. By exploiting the features of a regular grid, x and z coordinates will never change requiring only updates to the y coordinate (height offset). The quadtree structure exploits a child-parent relationship in which child nodes actually point to their parents data. In this way, when the data of children nodes are modified, the pointer actually dereferences some parent data completely eliminating any propagation back up through the quadtree. The need for nested error-bounds is also eliminated by depending solely on the view position for refinement. Even though this results in a less accurate refinement, the tessellation is tolerable and the tradeoff of propagation removal is well worth it.

Deformation is allowed to be done at any resolution within the extended quadtree. The quadtree may be extended to a user specified resolution by scaling up the original terrain and adding procedural fractal detail to the leaf nodes. These extra nodes are created on the fly in real-time and only need to be saved to disk if deformed. Since detail addition is so subtle, the extra nodes do not need to be spatially deterministic and can be randomly created each time. By comparing the time stamp of a nodes parent, data may procedurally adapt to a low resolution modification using this same method to create detail.

Along with our algorithm, we have presented support for large texture maps, fast normal calculation, and dealing with large world coordinate and depth buffer precision.

Limitations: The terrain is represented as a heightmap, precluding such features as caves and overhangs. The dimensions of the input heightmap are required to be $2^N + 1$ on each side to allow for optimizations. Additionally, the preprocessing step to build the terrain hierarchy is non-trivial for large datasets.

6 Future Work

For simplicity, not all optimizations were used when implementing this algorithm. It would be possible, with some effort, to port the entire algorithm to the GPU. Terrain data would reside completely in video memory in the form of a texture, and a quadtree structure could be mimicked via indices to a memory location. Vertex lists can easily be generated due to the regular grid layout, and indices could properly be generated with triangles in a vertex shader.

Creating disk space for nodes of added detail disrupts data coherency when layed out on disk. Though this isn't seen as a huge problem, it could be looked into further.

Currently the algorithm only allows for a single brush to be created at any given time. Ideally, it should accept a myriad of brushes at various resolutions that can be placed throughout the terrain and referenced by a specific identifier.

Often a terrain dataset is too large for deformations to be occurring everywhere. Deformations are sometimes limited to a specific region of interest even though terrain is present (the tank track deformation demo for example). Since the current algorithm supports deformation anywhere at any given time, the polygon throughput is not optimal. It would be possible to detect if a region of the quadtree hasn't been touched for a period of time, and if so, start to process the vertices into an optimal triangulated irregular network. The mesh could toggle back to a regular grid if deformation in that region was ever needed. This would result in faster rendering and somewhat more distinct feature preserving since nested error-bounds would be used within a triangulated irregular network. Note that the memory usage would have to remain the same since a toggle to a regular grid could happen at anytime, but the indices would change to allow for faster rendering of a patch.

Finally, the algorithm could be modified for rendering terrain at a planetary scale, which would require a specialized acceleration structure for ellipsoidal geometry.

Acknowledgements

The work shown has been partially sponsored by the Department of the Army, Army Research Office; the contents of the information does not necessarily reflect the position or the policy of the federal government, and no official endorsement should be inferred. This work is partially funded by the CAVE Project (ARO# N61339-04-C-0072) at the Desert Research Institute.

References

[1] Virtual terrain project. http://www.vterrain.org/. Last accessed: October 21, 2009.

[2] Samuel Atlan and Michael Garland. Interactive multiresolution editing and display of large terrains. *Computer Graphics Forum*, 25(2):211–223, June 2006.

[3] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. http://www.flipcode.com/archives/Fast_Terrain_Rendering_Using_Geometrical_MipMapping.shtml, 2000. Last accessed: 10/08/2009.

[4] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Mille, Charles Aldrich, and Mark B. Mineev-weinstein. Roaming terrain: Real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[5] Peter Freese. Solving accuracy problems in large world coordinates. In *Game Programming Gems 4*. Charles River Media, March 2004.

[6] Paul Glinker. Flight memory fragmentation with templated freelists. In *Game Programming Gems 4*. Charles River Media, 2004.

[7] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[8] Peter Lindstrom, David Koller, William Ribarsky, Larry F Hodges, Nick Faust, and Gregory Turner. Real-time, continuous level of detail rendering of height fields. pages 109–118, 1996.

[9] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 363–371, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM.

[11] Steen Lund Nielsen and Thomas Lauritsen. Rendering very large, very detailed terrains. http://www.terrain.dk/, 2005. Last accessed: 10/08/2009.

[12] Sean O'Neil. A real-time procedural universe, part three: Matters of scale. http://www.gamasutra.com/features/20020712/oneil01.htm. Last accessed: November 2007.

[13] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. page pages, 1998.

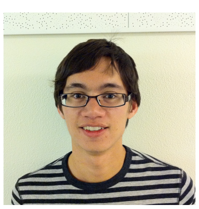
[14] Stefan Rottger, Wolfgang Heidrich, Philipp Slusallek, Hans peter Seidel, Graphische Datenverarbeitung (immd, and Universitt Erlangen-nrnberg). Real-time generation of continuous levels of detail for height fields. pages 315–322, 1998.

[15] Jason Shankel. Fast heightfield normal calculation. In *Game Programming Gems 3*. Charles River Media, 2002.

[16] Thatcher Ulrich. Rendering massive terrain using chunked level of detail control. ACM SIGGRAPH 2002: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, 2002.

[17] Huanxi Zhao. Fast accurate normal calculation for heightfield lighting on a non-isometric grid. In *CGIV 06: Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation*, pages 408–413. IEEE Computer Society, 2006.

William E. Brandstetter III Billy Brandstetter graduated from the University of Nevada Reno with a bachelors of science in computer science in 2005. He went on to graduate with a master of science with a specialization in computer graphics in 2007, where he wrote his thesis 'Multi-resolution Deformation in Out-of-Core Terrain Rendering'. His interests include research in computer graphics, scientific visualization, and game programming.



Joeseph Mahsman received his Bachelor's degree in 2008 and his Master's degree in 2010 from the University of Nevada, Reno. He is a researcher in the High-Powered Computing and Visualization Laboratory. His research interests include interactive planetary visualization, real-time 3D graphics, and building virtual worlds.

Cody White received his Bachelor's degree in 2009 from the University of Nevada, Reno and is currently working on his Masters. He works as a researcher in the High-Powered Computing and Visualization Laboratory at UNR and his research interests include real-time 3D graphics and photorealistic rendering.

Dr. Sergiu Dascalu is an Associate Professor in the Department of Computer Science and Engineering at the University of Nevada, Reno, USA, which he joined in 2002. In 1982 he received a Masters degree in Automatic Control and Computers from the Polytechnic University of Bucharest, Romania and in 2001 a PhD in Computer Science from Dalhousie University, Halifax, Nova Scotia, Canada. His main research interests are in the areas of software engineering and human-computer interaction. Sergiu has published over 100 peer-reviewed journal and conference papers and has been involved in numerous projects funded by industrial companies as well as US federal agencies such as NSF, NASA, and ONR.

Dr. Frederick C Harris, Jr. is currently a Professor in the Department of Computer Science and Engineering and the Director of the High Performance Computation and Visualization Lab at the the University of Nevada, Reno, USA. He received his BS and MS in Mathematics and Educational Administration from Bob Jones University in 1986 and 1988 respectively, his MS and Ph.D. in Computer Science from Clemson University in 1991 and 1994 respectively. He is a member of ACM, IEEE, and ISCA. His research interests are in Parallel Computation, Graphics and Virtual Reality, and Bioinformatics.