Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)

Filip Strugar, 11 July 2010

Abstract. This paper presents a technique for GPU-based rendering of heightmap terrains, which is a refinement of several existing methods with some new ideas. It is similar to the terrain clipmap approaches [*Tanner et al. 98*, *Losasso 04*], as it draws the terrain directly from the source heightmap data. However, instead of using a set of regular nested grids, it is structured around a quadtree of regular grids, more similar to [*Ulrich 02*], which provides it with better level-of-detail distribution. The algorithm's main improvement over previous techniques is that the LOD function is the same across the whole rendered mesh and is based on the precise three-dimensional distance between the observer and the terrain. To accomplish this, a novel technique for handling transition between LOD levels is used, which gives smooth and accurate results. For these reasons the system is more predictable and reliable, with better screen-triangle distribution, cleaner transitions between levels, and no need for stitching meshes. This also simplifies integration with other LOD systems that are common in games and simulation applications. With regard to the performance, it remains favourable compared to similar GPU-based approaches and works on all graphics hardware supporting Shader Model 3.0 and above. Demo and complete source code is available online under a free software license.

Introduction

Heightmap display and interaction are a frequent requirement of game and simulation graphics engines. The simplest way to render a terrain is the brute force approach, in which every texel in the source heightmap data is represented with one vertex in the regular grid of triangles. For larger datasets this is not practical or possible: thus the need for level of detail (LOD) algorithms. An LOD system will produce a mesh of different complexity, usually as the function of the observer distance, so that the on-screen triangle distribution is relatively equal while using only a subset of data and rendering resources.

Historically these algorithms were executed on the CPU; a good example is the classic academic algorithm [*Duchainea et al. 97*]. However, since the GPU's raw (mostly parallel) processing power has been improving much faster than the CPU's, in order for the whole system to be optimally used, the terrain-rendering algorithms have changed to draw on the graphics hardware as much as possible.

Currently, in the context of modern PC and game console architecture, there is little benefit of having an algorithm that produces optimal triangle distribution on the CPU if it cannot provide enough triangles for the hardware graphics pipeline or if it uses too much CPU processing power. The API, driver, and OS layer between the CPU and GPU is also a common bottleneck, as explained in [*Wloka 03*]. Therefore, even a simplistic GPU-based approach can be faster and provide better visual results than those complex approaches executed on the CPU and formerly considered optimal.

One of the first examples of this trend is the algorithm given in [de Boer 00], which, while still essentially a CPU-based algorithm, is aimed at producing a high triangle output with less optimal distribution but also less expensive execution [Duchainea et al. 97], thus providing better results when running on early dedicated graphics hardware.

Later [*Ulrich 02*] developed one of the first completely GPU-oriented algorithms, which is still an excellent choice for rendering terrains on modern hardware due to its good detail distribution and optimally tessellated mesh. Its drawbacks are the lengthy pre-processing step involved, inability to modify terrain data in real time, and a somewhat inflexible and less correct LOD system.

A simpler heightmap-based approach is sometimes preferred; one of the most popular is the [*Asirvatham et al. 05*] and its various improvements.

This paper presents a technique that builds upon the idea of using a fixed grid mesh displaced in the vertex shader to render the terrain directly from the heightmap data source while dealing with some of the shortcomings and complexities found in [*Ulrich 02*] and [*Asirvatham et al. 05*] by using a quadtree-based structure and a novel, completely predictable continuous LOD system. This technique is intended to provide an optimal way of rendering heightmap-based terrains on graphics hardware from the Shader Model 3.0 generation and above; the technique can be extended with hardware tessellation support to fully take advantage of the latest Shader Model 5.0 generation capabilities.

LOD function

One major drawback of the basic clipmaps [*Asirvatham et al. 05*] algorithm is that the level of detail is essentially based on the two-dimensional (x; y: latitude, longitude) components of the observer position, while ignoring the height. This results in unequal distribution of mesh complexity and aliasing problems as, for example, when the observer is high above the mesh, the detail level below remains much greater than required and vice versa. This is only partially addressed in [*Asirvatham et al. 05*] by taking the current observer height above the terrain into consideration and dropping higher LOD levels if needed.

On the other hand, [*Ulrich 02*] uses an LOD function that is the same over the whole chunk (mesh block), providing only approximate three-dimensional distance-based LOD.

Such approximations can be limiting in scenarios frequently encountered in modern games or simulation systems, where terrain is commonly very uneven and the observer's height and position changes quickly, as it produces poor detail distribution or movement-induced rendering artifacts. It also causes integration difficulties with other rendering system, some of which use level-of-detail optimizations themselves, since the unpredictable terrain LOD causes rendering errors such as unwanted intersection or floating of objects placed on the terrain.

The technique presented here of continuous distance-dependent level detail (CDLOD) solves these problems by providing an LOD distribution that is a direct function of three-dimensional distance across all rendered vertices and is thus completely predictable.

LOD transition

Another drawback of the techniques of [*Ulrich 02*] and [*Asirvatham et al. 05*] is that the discontinuities between LOD levels require additional work to remove gaps and provide smooth transitions. This is addressed by using additional connecting ("stitching") strips between different LOD levels. These strips, besides adding to the rendering cost, can cause various issues such as artifacts when rendering shadow maps; unwanted overdraw when the terrain is rendered transparently (which is a problem when rendering terrain water or similar effects); etc.

Also, since the mesh swap between LOD levels happens between meshes of different triangle count and shape, differences in vertex output interpolation will result in “popping” artifacts that appear if any vertex-based effect (such as vertex lighting) is used. This also makes it a less suitable platform for hardware tessellation.

The CDLOD technique inherently avoids these problems because the algorithm used to transition between LOD levels completely transforms the mesh of a higher level into the lower detailed one before the actual swap occurs. This ensures a perfectly smooth transition with no seams or artifacts. In addition, the rendering itself is simpler and more predictable than that of [*Asirvatham et al. 05*], as only one rectangular regular grid mesh is needed to render everything.

Data organization and streaming

Storing, compressing, and streaming the terrain data can be done as with other techniques, requiring no special attention. While this topic is outside the scope of this paper, it is a necessary part of any practical large terrain rendering system implementation. Thus an example CDLOD implementation with full data streaming is provided with the *StreamingCDLOD* demo.

Algorithm implementation

Overview

CDLOD organizes the heightmap into a quadtree, which is used to select appropriate quadtree nodes from different LOD levels at run time. The selection algorithm is performed in such a way as to provide approximately the same amount of on-screen triangle complexity regardless of the distance from the observer.

The actual rendering is performed by rendering areas covered by selected nodes using only one unique grid mesh, reading the heightmap in the vertex shader, and displacing the mesh vertices accordingly.

A more detailed mesh can be smoothly morphed into the less detailed one in the vertex shader so that it can be seamlessly replaced by a lower resolution one when it goes out of range, and vice versa.

The quadtree structure is generated from the input heightmap. It is of constant depth, predetermined by memory and granularity requirements. Once created, the quadtree does not change unless the source heightmap changes. Every node has four child nodes and contains minimum and maximum height values for the rectangular heightmap area that it covers. A provided example, *BasicCDLOD*, uses a naive explicit quadtree implementation where all required quadtree data is contained in the node structure. The *StreamingCDLOD* example implements a more advanced version in which the algorithm relies only on the (partially compressed) min/max maps while all other data is implicit and generated during the quadtree traversal. This version uses far less memory but is slightly more complex.

Quadtree and node selection

The first step of the rendering process is the quadtree node selection. It is performed every time the observer moves, which usually means during every frame.

In the presented version of the algorithm, a quadtree depth level always corresponds to the level of detail. This is an artificial constraint induced for simplicity reasons because it allows for the same grid mesh with a fixed triangle count to be used to render every quadtree node. In that case every child node has four times more mesh complexity per square area unit than its parent, since a child node occupies a fourth of the area. This complexity difference scale factor of four is then used as a basis for the LOD level distribution. In other words, each successive LOD level is used to render four times as many triangles and contains four times more nodes than the previous one (see *Figure 1*).

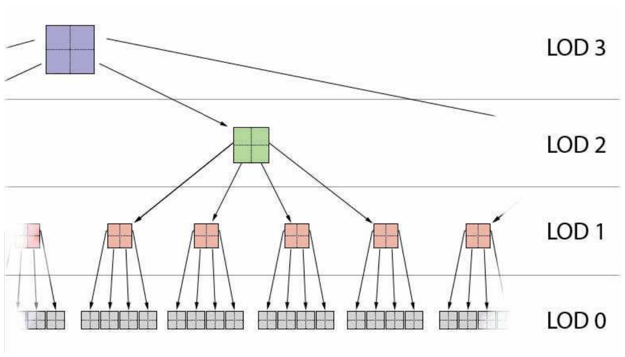

Figure 1 *Quadtree and LOD layers.*

LOD distances and morph areas

In order to know which nodes to select where, distances covered by each LOD layer are precalculated before the node selection process is performed. These are calculated with the goal of producing approximately the same average number of triangles per square unit of screen over the whole rendered terrain.

Since the complexity difference between each successive LOD level is fixed to four by the algorithm design, the difference in distance covered by them needs to be close to 2.0 to accomplish relatively even screen triangle distribution, assuming that the three-dimensional projection transform is used for rendering (due to the way projection transform works).

The array of LOD ranges is thus created (see *Figure 2*) with the range of each level being two times larger than that of the previous one, and the range of the final level (largest, least detailed) representing the required total viewing distance.

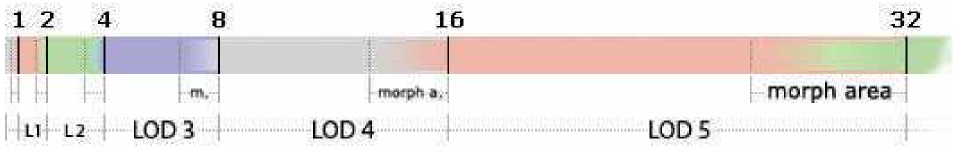

Figure 2 *The distribution of six LOD ranges with their relative sizes at the top.*

To provide smooth LOD transitions, the morph area is also defined, marking the range along which a higher complexity mesh will morph into the lower one. This morph area usually covers the last 15%-30% of every LOD range.

Quadtree traversal and node selection

Once the array of LOD ranges is calculated it is used to create a selection (subset) of nodes representing the currently observable part of the terrain. To do this, the quadtree is traversed recursively beginning from the most distant nodes of the lowest-detailed level, working down to the closest, most detailed ones. This selection then contains all dynamic metadata required to render the terrain.

The following C++ pseudocode illustrates a basic version of the algorithm:

```
// Beginning from the LODLevelCount and going down to 0; lodLevel 0 is the highest detailed level.
bool Node::LODSelect( int ranges[], int lodLevel, Frustum frustum )
{
   if( !nodeBoundingBox.IntersectsSphere( ranges[ lodLevel ] ) )
   {
      // no node or child nodes were selected; return false so that our parent node handles our area
      return false;
   }

   if( !FrustumIntersect(frustum) )
   {
      // we are out of frustum, select nothing but return true to mark this node as having been
      // correctly handled so that our parent node does not select itself over our area
      return true;
   }

   if( lodLevel == 0 )
   {
      // we are in our LOD range and we are the last LOD level
      AddWholeNodeToSelectionList( );

     return true;          // we have handled the area of our node
   }
   else
   {
      // we are in range of our LOD level and we are not the last level: if we are also in range
      // of a more detailed LOD level, then some of our child nodes will need to be selected
      // instead in order to display a more detailed mesh.
      if( !nodeBoundingBox.IntersectsSphere( ranges[lodLevel-1]) )
      {
         // we cover the required lodLevel range
         AddWholeNodeToSelectionList( ) ;
      }
      else
      {
         // we cover the more detailed lodLevel range: some or all of our four child nodes will
         // have to be selected instead
         foreach( childNode )
         {
            if( !childNode.LODSelect( ranges, lodLevel-1, frustum ) )
            {
               // if a child node is out of its LOD range, we need to make sure that the area
               // is covered by its parent
               AddPartOfNodeToSelectionList( childNode.ParentSubArea ) ;
            }
         }
      }
      return true;          // we have handled the area of our node
   }
}
```

Selecting a node involves storing its position, size, LOD level, info on partial selection, and other data if needed. This is saved in a temporary list later used for rendering.

Each node can be selected partially over the area of only some of its four child nodes. This is done so that not all child nodes need to be rendered if only a few are in their LOD range, allowing for earlier exchange between LOD levels, which increases the algorithm performance and flexibility.

Nodes are also frustum culled while traversing the quadtree, eliminating the rendering of non-visible nodes. When rendering a shadow map or a similar effect, the frustum cull is based on the shadow camera, but the actual LOD selection should still be based on the main camera settings. This is done to avoid rendering artifacts that would be caused by difference in the terrain mesh resulting from two different LOD functions (for the shadow map camera and the main camera).

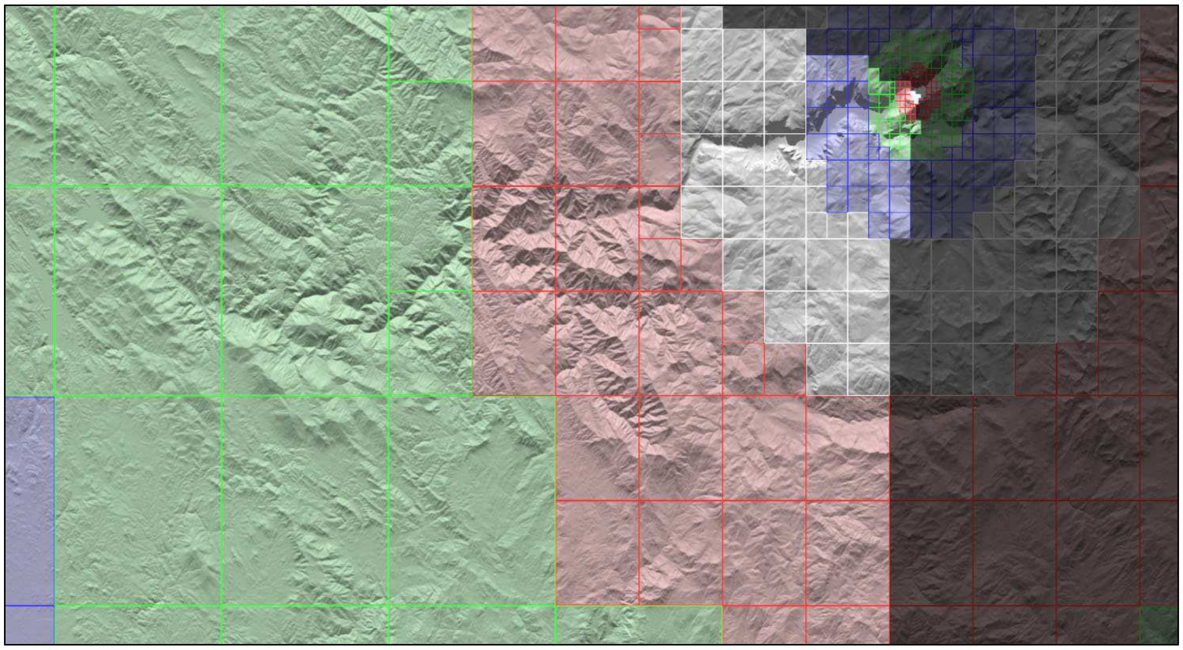

Figure 3 *LOD selection of quadtree nodes (the frustum culled section is shaded in dark).*

Since the LOD layer selection is based on the actual three-dimensional distance from the observer, it works correctly for all terrain configurations and observer heights. This results in correct detail complexity and better performance in various scenarios, as illustrated in *Figure 3* and *Figure 4*.

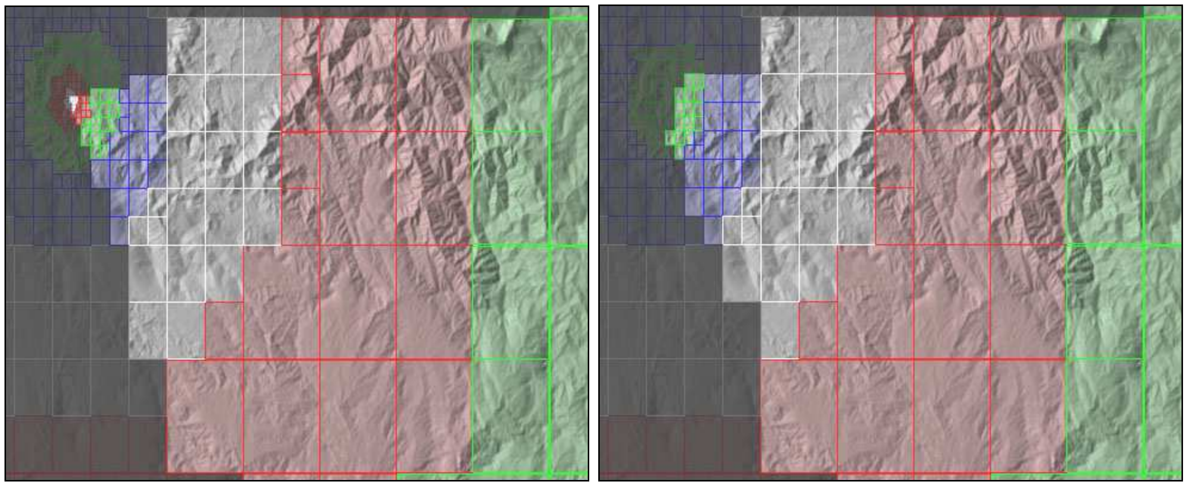

Figure 4 *LOD selection of quadtree nodes with two different observer heights (frustum culled section shaded in dark).*

Rendering

To render the terrain, we iterate through the selected nodes and their data is used to render the patch of the terrain that they cover. Continuous transition between LOD levels is done by morphing the area of each layer into a less complex neighbouring layer to achieve seamless transition between them (see *Figure 5*).

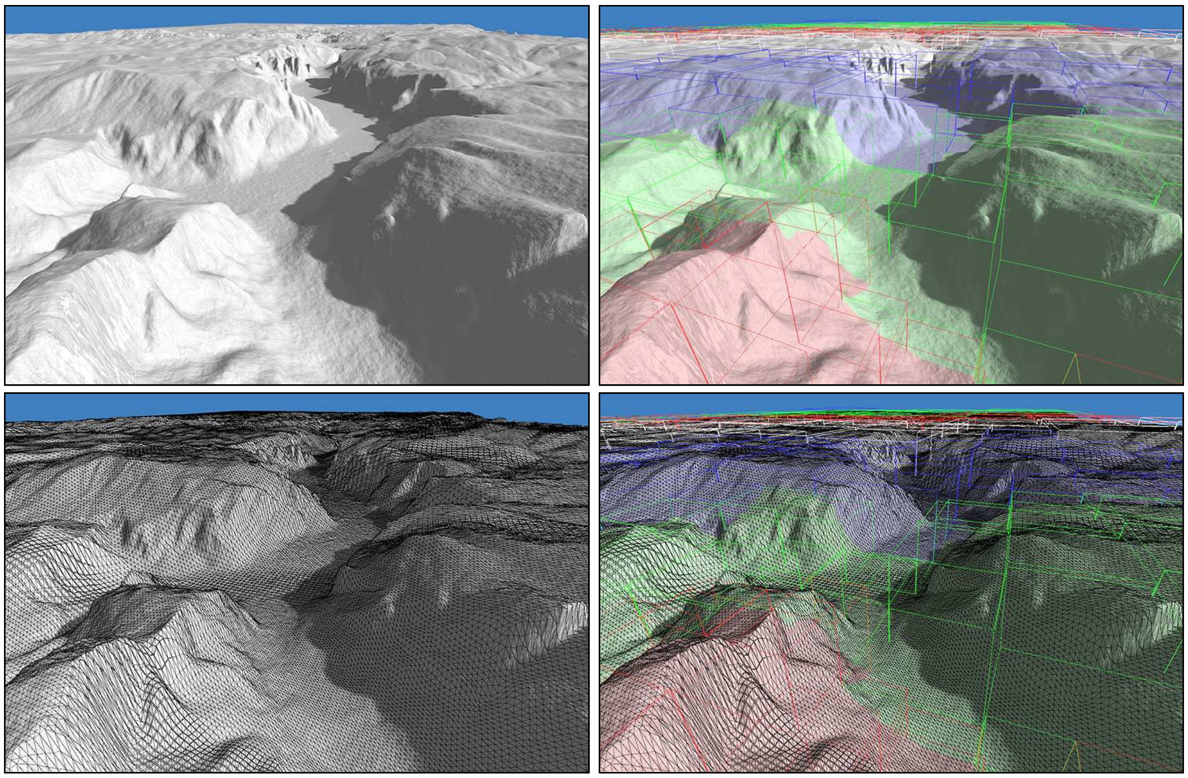

Figure 5 *Distribution of LOD levels and nodes (different colors represent different layers).*

Rendering is then fairly straightforward: a single grid mesh of fixed dimensions is transformed in the vertex shader to cover each selected node area in the world space, and vertex heights are displaced using texture fetches, thus forming the representation of the particular terrain patch. Commonly used grid-mesh dimensions are 16x16, 32x32, 64x64 or 128x128, depending on the required output complexity. Once chosen, this grid mesh resolution is constant (i.e., equal for all nodes) but can be changed at run time, which can come in handy for rendering the terrain at lower resolutions for effects requiring less detail such as reflections, secondary views, low quality shadows, etc.

Morph implementation

Using CDLOD, each vertex is morphed individually based on its own LOD metric unlike the method in [*Ulrich 02*], where the morph is performed per-node (per-chunk). Each node supports transition between two LOD layers: its own and the next larger and less complex one. Morph operation is performed in the vertex shader in such way that every block of eight triangles from the more detailed mesh is smoothly morphed into the corresponding block of two triangles on the less detailed mesh by gradual enlargement of the two triangles and reduction of the remaining six triangles into degenerate triangles that are not rasterized. This process produces smooth transitions with no seams or T-junctions (see *Figure 6* and *Figure 7*).

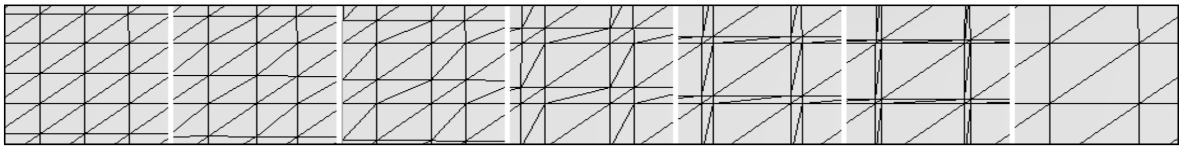

Figure 6 *Grid mesh morph steps with morphK values of 0.0, 0.2, 0.4, 0.6, 0.9, and 1.0.*

First, the approximate distance between the observer and the vertex is calculated to determine the amount of morph required. The vertex position used to calculate this distance can be an approximation as long as the approximating function is consistent over the whole dataset domain. This is necessary since, in order to prevent seams, the vertices on the node's grid mesh edges must remain exactly the same as the ones on the neighbouring nodes. In our case, *x* (latitude) and *y* (longitude) components will always be correct as the same function is used to stretch them to the world space, but z (height) must either be approximated or sampled from the heightmap using a consistent filter.

The morph value, *morphK* in the example code, ranging between 0 (no morph, high detail mesh) and 1 (full morph, four times fewer triangles), is used to gradually move each morph mesh vertex towards the corresponding no-morph one. A morph vertex is defined as a grid vertex having one or both of its grid indices (*i*, *j*) as an odd number, and its corresponding no-morph neighbour is the one with coordinates *(i – i/2, j – j/2)*.

Following is the HLSL code used to morph vertices:

```
// morphs input vertex uv from high to low detailed mesh position
//  - gridPos: normalized [0, 1] .xy grid position of the source vertex
//  - vertex:  vertex.xy components in the world space
//  - morphK:  morph value

const float2 g_gridDim = float2( 64, 64 );

float2 morphVertex( float2 gridPos, float2 vertex, float morphK )
{
   float2 fracPart = frac( gridPos.xy * g_gridDim.xy * 0.5 ) * 2.0 / g_gridDim.xy;
   return vertex.xy - fracPart * g_quadScale.xy * morphK;
}
```

Finally, the z-component is obtained by sampling the heightmap with texture coordinates calculated from x and y components using a bilinear filter (filtering is only needed for vertices in the morph region). When all vertices of a node are morphed to this low-detail state, the mesh then effectively contains four times fewer triangles and exactly matches the one from the lower LOD layer; hence it can be seamlessly replaced by it.

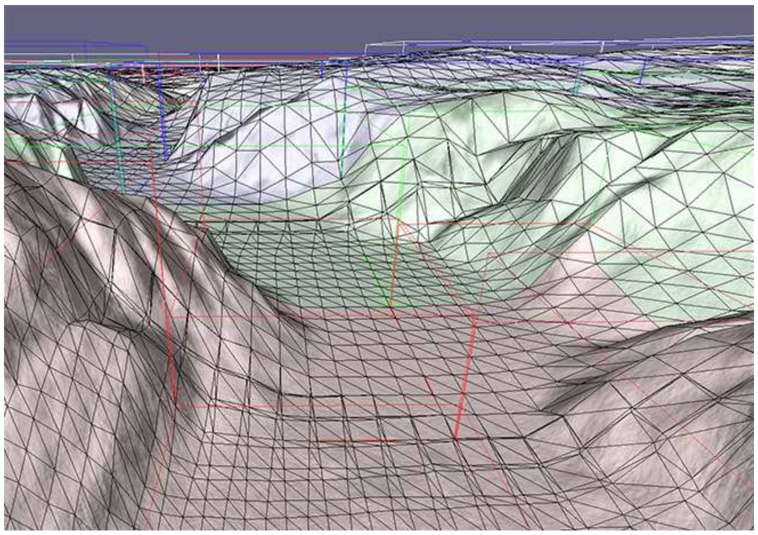

Figure 7 *LOD levels, quadtree nodes, and mesh wireframe with morphing.*

Settings

Settings used to generate the quadtree and run the algorithm need to be carefully chosen to match terrain dataset characteristics while providing the best performance. In the accompanying examples, each dataset defines its own settings. Following is a brief description of the most important ones:

The ***LeafQuadTreeNodeSize*** setting determines the depth (granularity) of the quadtree in heightmap raster size, and *LODLevelCount* defines the number of LOD levels. Using more LOD levels provides greater viewing distance and better LOD distribution, but reduces performance and increases memory use. Smaller values of *LeafQuadTreeNodeSize* allow for a greater range of viewing distances and better handling of very uneven terrains, but increases quadtree memory use and batch count [*Wloka 03*]. These two values are set during the export phase and remain fixed at run time.

The ***RenderGridResolutionMult*** defines the resolution of the static grid mesh used to render the terrain - it affects the triangle output but does not change algorithm behaviour in any other way. It is a handy tool for easily balancing rendering quality and algorithm performance at run time.

View distance can also be varied at run time and is used to change maximum rendering distance and detail level. Unlike *RenderGridResolutionMult*, changing it will affect LOD ranges, node selection, and consequently the number of render batches, required streaming data, etc.

Granularity issues

One limitation of the algorithm is that a single quadtree node can only support transition between two LOD layers. This limits the minimum possible viewing range, or the minimum quadtree depth, because only one smooth transition can be performed between two LOD layers over the area of the same node. Increasing the viewing range will move LOD transition areas further away from each other and solve the problem at the expense of having more render data to process. The other options are to reduce the number of LOD levels, which reduces the benefits of the LOD system, or to increase the quadtree depth to increase the granularity, which increases quadtree memory and CPU use. The size of the morph area can also be decreased to mitigate this problem, but that can make the transition between levels noticeable.

Since the LOD works in three dimensions, this problem will be enhanced when using extremely rough terrain with large height differences: thus, different settings might be required for each dataset.

In the provided data examples, LOD settings are tuned so that the ranges are always acceptable. In the case where different datasets and settings are used, these LOD transition problems can appear in the form of seams between LOD levels. Debug builds of the demo code will always detect a possibility of such a situation and display a warning so that settings can be corrected (detection code is overzealous, so a warning does not guarantee that the seam will be observable - just that it is theoretically possible).

Streaming and optimal quadtree data storage

Any practical implementation of a large terrain-rendering algorithm must also provide mechanics for keeping only a subset of the required terrain dataset in the working memory. This subset is usually the minimum required to render the terrain from the observer's perspective, and it is loaded (streamed) in and out as the observer's position changes. While a detailed discussion of the topic of streaming is outside the scope of this article, some basics will be covered. The *StreamingCDLOD* implementation provides all necessary implementation details if required.

The CDLOD technique requires storage for the two separately handled datasets:

- **Quadtree data**, which is the metadata required for the LOD algorithm to work.
- **Terrain data**, terrain data, which is the actual data required for the rendering and consists of a heightmap, normal map, overlay images, etc.

Quadtree data

The quadtree data in the *BasicCDLOD* implementation is a part of the node structure. As each node contains data defining its size, position, pointers to its siblings, etc., a lot of unnecessary memory is used. The *StreamingCDLOD* implementation avoids this by keeping only the necessary *minZ* and *maxZ* values, which describe the minimum and maximum height values of the heightmap area that the node covers. The rest of the data is automatically generated during quadtree traversal at some small additional computational cost and a very high (approximately 30x) memory saving.

To store this min/max data, each LOD level uses a two-dimensional matrix of two unsigned 16-bit integer values: one two-value set for each quadtree node of the corresponding quadtree level. The dimensions of the matrix for the most detailed LOD level 0 are thus [*HeightmapWidth*/*LeafQuadTreeNodeSize*, *HeightmapHeight*/*LeafQuadTreeNodeSize*], with every next LOD level requiring four times less min/max data.

To further improve memory use, the most detailed LOD level matrix is compressed by storing the min/max values in the normalized space of the parent node's min/max values, which can be stored using unsigned 8 bit integers with little precision loss. This reduces memory usage to approximately 1/45 compared to the naive implementation in the *BasicCDLOD* implementation, and is close to the maximum theoretically possible.

This min/max matrix represents all the data required by the CDLOD algorithm (aside from the actual heightmap). Since it is small compared to the rest of the terrain data, the *StreamingCDLOD* implementation keeps the whole min/max matrix in memory. For example, rendering a terrain with the source heightmap of 32K x 32K bytes and a leaf quadtree node size of 32 requires around 3.5 MiB of memory.

Terrain data

The terrain data (heightmap, normal map, overlay images, etc.) is split into streamable blocks for each LOD layer and is optionally compressed. In some cases, texture blocks might need to overlap slightly with neighbouring ones to avoid rendering artifacts near the border of the areas due to texture filtering on the edges.

At run time, the same LOD selection algorithm used for rendering is also used to select nodes in range, which are then used to mark data blocks as potentially visible and to stream them in or out accordingly. The streaming data block granularity is usually much smaller than the quadtree granularity, so one data block contains data for many quadtree nodes.

Different streaming LOD settings can be used to balance higher quality rendering and lower memory use.

Example code

Two example projects are provided: *BasicCDLOD* and *StreamingCDLOD*. Both projects are written in C++, using DirectX9 and HLSL, and should work on most GPUs that support vertex shader texture sampling, i.e., ones supporting Shader Model 3.0.

BasicCDLOD implements the CDLOD technique in its basic form. *StreamingCDLOD* implements the CDLOD technique in its practical usable form with more optimal base algorithm memory use and simple data streaming.

The source code is distributed under Zlib license; see the end of this paper for download links.

Performance

Performance measurements presented here are obtained using the *StreamingCDLOD* implementation of the algorithm.

As with other similar GPU-based techniques, performance of the CDLOD will be limited by the GPU's ability to execute the vertex shader and/or rasterize triangles. Although sampling textures in the vertex shader can be very expensive, this cost is usually not the main limiting factor due to the good screen triangle distribution, which ensures that a very high triangle number is not required to achieve good visual quality.

Comparison

When compared to the methods in [*Asirvatham et al. 05*], CDLOD produces higher quality triangle distribution and utilization, fewer or no rendering artifacts, and a similar triangle output and render batch count [*Wloka 03*]. On the downside, there is a small added memory cost required for the quadtree storage.

On the other hand, the algorithm of [*Ulrich 02*] has a potentially higher GPU performance for the similar visual quality as it renders the precalculated adaptively tessellated terrain mesh, but it is much less practical than a heightmap-based approach for the same reason. It also suffers from similar visual artifacts as [*Asirvatham et al. 05*] due to lack of smooth LOD distribution, as detailed in the *Introduction*.

Datasets

These are the datasets used for testing:

- califmtns_large: 48897 x 30465 heightmap with a normalmap and a simple texture splatting technique;
- hawaii: 13825 x 16769 heightmap with a normalmap and overlay topographic map
- puget: the classic "Puget Sound" 16385x16385 heightmap with no normalmap, no dynamic lighting, no detail map, and an overlay color map with embedded lighting

Settings used for the measurements provide balance between quality and performance. The datasets are large enough for realistic profiling of memory use (especially the califmtns_large) and CPU/GPU performance; increasing the source heightmap/image size will proportionally increase hard disk storage requirements and quadtree memory use (which is not streamed in the example code), but it will not significantly affect total memory use or CPU/GPU performance.

CPU performance

Since no data is generated on the CPU and a small amount of data is transferred to the GPU per frame, the algorithm is inherently GPU-limited in almost all scenarios. Most of the required CPU time is consumed by DirectX9 rendering API calls, while a smaller portion of it is spent in the actual CDLOD algorithm (quadtree traversal and data selection). The usual number of render batches per LOD level is ten. With seven to nine LOD levels used in demo datasets, an average of 60 to 120 render batches per frame is required to render the terrain.

Table 1 shows typical CPU time spent during one frame on quadtree traversal (together with streaming data selection) and DirectX9 rendering (where applicable) on three different CPU platforms.

The cost of the data streaming is not presented as it is directly dependent on the observer movement, compression algorithms, and data size, and it is usually executed on separate threads anyway (as in the *StreamingCDLOD* example).

As presented, the DirectX9 rendering cost should only be used as a guideline since the rendering code is not very optimized and can also vary greatly depending on the driver, OS version, etc.

Dataset	Rendered batch count	Rendered triangle count	Intel Atom330 1.6Ghz quadtree (+ rendering)	AMD AthlonXP 1.6Ghz Quadtree	Intel E8500 3.16Ghz quadtree (+ rendering)
califmt.	74	380k	0.573 ms (+ 1.208 ms)	0.295 ms	0.085 ms (+ 0.415 ms)
puget	84	105k	0.525 ms (+ 0.937 ms)	0.280 ms	0.075 ms (+ 0.268 ms)
hawaii	83	446k	0.537 ms (+ 1.189 ms)	0.287 ms	0.077 ms (+ 0.350 ms)

Table 1 *Typical CPU time spent during one frame on quadtree traversal and DirectX9 rendering on three CPU platforms.*

GPU performance

The performance bottleneck on the GPU is either the vertex texture fetch cost (used for displacement of terrain vertices), or the triangle rasterization and pixel shader cost. This mostly depends on the settings, GPU hardware, and display resolution.

Table 2 shows typical framerates (frames/second) over various display resolutions and different datasets. The datasets califmtns_large and hawaii are displayed using lighting based on one directional light and normals from the high resolution normalmap texture, with no overlay texturing. The puget dataset uses only overlay texture with baked lighting.

Measurements were performed on a Core2 Duo 3.16Ghz system, except for the NVidia ION, which uses a 1.66 GHz Atom 330 CPU.

Dataset	Rendered triangle count	NVidia ION			NVidia 8800GT			ATI 5870		
		640x480	1024x768	1680x1050	640x480	1024x768	1680x1050	1024x768	1680x1050	1920x1200
califmt.	380k	85	75	56	760	707	567	1629	1364	1253
puget	105k	192	161	95	2037	1876	1293	1997	1994	1885
hawaii	446k	78	60	53	690	638	520	1455	1364	1198

Table 2 *Typical framerates (frames/second) over various display resolutions and different datasets.*

Memory use

Memory usage depends on the quadtree granularity, streaming settings, and data. Changing the exporter LOD settings and viewing range can change the memory use by an order of magnitude. For that reason, care must be taken to establish the right balance between rendering quality and memory use. *Table 3* shows typical memory usage of demo datasets. Data includes heightmap, normal map, and image overlay textures, and they are measured using default settings and worst-case observer locations.

Dataset	Resolution (heightmap / detailmap) and coverage	Viewing range	Quadtree memory use	Texture memory use (hm+nm+om)
califmt.	10 / 5 m, 488 x 304 km	92 km	4,735 KiB	30 MiB, 26 MiB, 34 MiB (90 MiB total)
puget	10 m, 163 x 163 km	175 km	853 KiB	24 MiB, 0 MiB, 14 MiB (38 MiB total)
hawaii	10 / 5 m, 138 x 167 km	176 km	737 KiB	28 MiB, 30 MiB, 45 MiB (103 MiB total)

Table 3 *Typical memory usage of demo datasets.*

Memory used to run decompression and other algorithms that are not directly related to the terrain rendering are not presented as such usage will vary greatly based on the use scenario and implementation details. This additional memory use is usually substantially lower than the combined quadtree and texture memory use presented in the *Table 3.*

Additional thoughts

Hardware (GPU-Based) tessellation

The newer generation of graphics hardware (Shader Model 5) supports programmable hardware-based tessellation. This can be used to provide the best of both worlds when rendering a terrain: performance and good visual output of an adaptively tessellated mesh and the flexibility of a heightmap-based approach.

Since CDLOD performs the transition between LOD layers using smooth continuous morph of triangles with no discontinuities or T junctions, additional subdivision can easily be applied on top without breaking the continuity that makes it a good platform for hardware tessellation.

Performance on older hardware

In the case of older graphics cards (Shader Model 2 or lower), the heightmap sampling in the vertex shader can be too costly or even impossible due to the lack of hardware capabilities. In that case, merging the CDLOD and ChunkedLOD [*Ulrich 02*] techniques is an option in which benefits of a precalculated adaptively tessellated mesh can be used in the CDLOD framework with the benefits of a similar continuous LOD algorithm.

This can be achieved by pre-calculating a tessellated triangle mesh, simplifying it for each successive less detailed LOD level, and splitting it into the quadtree. Each vertex can contain additional morph data so that during the LOD transition some vertices can be moved into neighbouring ones in the vertex shader, eliminating triangles and effectively morphing the more detailed LOD mesh into the less detailed one. This technique provides correct continuous three-dimensional distance-based LOD, removes the need for stitching strips, and avoids the major problem that exists in the approach in [*Ulrich 02*].

The downsides of this approach, compared to the heightmap-based one, are the lengthy data pre-processing step, more difficult mesh compression, and the fact that real-time terrain modification would be possible.

References

[Asirvatham et al. 05] A. Asirvatham, H. Hoppe. "*Terrain Rendering Using GPU-Based Geometry Clipmaps*. " In GPU Gems 2, edited by M. Pharr, pp. 27-45. Reading, MA: Addison-Wesley, 2005.

[de Boer 00] Willem H. de Boer. "*Fast Terrain Rendering Using Geometrical MipMapping*. " Available at http://www.ipcode.com/tutorials/geomipmaps.pdf, October 2000.

[Duchainea et al. 97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. "*ROAMing Terrain: Real-time Optimally Adapting Meshes.* " Proceedings of the Eighth Conference on Visualization, pp. 81{88. Los Alamitos, CA: IEEE, 1997.

[Losasso 04] F. Losasso, H. Hoppe. "*Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids*." ACM Transactions on Graphics, 23:3 (2004), 769-776.

[Tanner et al. 98] Christopher Tanner, Christopher Migdal, and Michael Jones. "*The Clipmap: A Virtual Mipmap*. " In Proceedings of the 25th Annual Conference on Computer Graphics, pp. 151-158. New York: ACM, 1998.

[Ulrich 02] Thatcher Urlich. "*Rendering Massive Terrains Using Chunked Level of Detail Control*. " SIGGRAPH 2002 Course Notes. San Antonio, TX: ACM, 2002.

[Wloka 03] M. Wloka "*Batch, Batch, Batch: What Does It Really Mean?* " NVIDIA. Available at http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf, 2003.

Demos, code, data

Additional material can be found online:

Binaries and a small example dataset
http://www.vertexasylum.com/downloads/cdlod/binaries_tools_testdata.exe

Complete source code
http://www.vertexasylum.com/downloads/cdlod/source.zip

Example datasets
http://www.vertexasylum.com/downloads/cdlod/dataset_califmtns.exe,
http://www.vertexasylum.com/downloads/cdlod/dataset_hawaii.exe,
http://www.vertexasylum.com/downloads/cdlod/dataset_puget.exe

Example animations
http://www.vertexasylum.com/downloads/cdlod/cdlod_calif.wmv,
http://www.vertexasylum.com/downloads/cdlod/cdlod_hawaii.wmv,
http://www.vertexasylum.com/downloads/cdlod/cdlod_params.wmv

Paper revision 1
Originally published in the "journal of graphics, gpu and game tools":
http://jgt.akpeters.com/papers/Strugar10/

Filip Strugar, filip.strugar@gmail.com, filip.strugar@rebellion.co.uk