Terrain Rendering using Geometry Clipmaps

Robert Rose

June 7, 2005

Abstract

Geometry clipmaps present a quick, intuitive, and high performance solution to the problem of rendering terrain at varying levels of detail (LOD). Recent advances in graphics hardware allow a significant portion of geometry clipmap processing to be done on the GPU, making the performance even better. This paper explores implementing geometry clipmaps on the GPU and discusses the optimizations and pitfalls involved in doing so.

1 Introduction

Rendering terrain is a common task for many computer graphics projects. Flight simulators, Geographic Information Systems (GIS) applications, video games, etc., all need to be able to render terrain in a quick and efficient manner. For the purposes of this paper, the characteristics of the desired terrain renderer will be as follows:

1. Real-time. The terrain will need to be rendered in real-time. Moving the camera should not result in a loss of performance or quality.
2. Appropriate level of detail. Pieces of the terrain will be rendered at an appropriate level of detail. Terrain far away will have low detail, whereas terrain closer to the viewer will have higher detail.
3. Watertight. The terrain will not have any gaps in it.

To simplify the terrain rendering problem, we'll make some assumptions about the terrain we will be working with:

1. Rectilinear format. The terrain data will come on an evenly spaced grid.

2. Heightmap format. The terrain data can be represented as a heightmap (no “caves” in the terrain).

Geometry clipmaps [1] [2] provide and quick, intuitive and high performance solution to the terrain rendering requirements and assumptions described above. The reader is assumed to already be familiar with geometry clipmaps. This paper explores implementing several variations and issues involved with implementing geometry clipmaps on the GPU, including:

- View Dependent Geometry Clipmaps, a variation that fixes the clipmaps in view space instead of world space.
- Exponentially-Sized Clipmaps, another variation that uses only one vertex patch, making implementation much easier.
- Normal Blending, a different technique for generating the height map’s normals that consumes less memory.
- Single height map, a very simple approach to quickly rendering terrain using the geometry clipmap technique.

2 View Dependent Geometry Clipmaps

In the previous Geometry Clipmap work, clipmaps were oriented along the x,z plane and move only when the camera exceeds some threshold. Every frame, the positions of the clipmaps must be recomputed by the CPU and then passed to the GPU. The “view dependent” approach attempts to skip this step by fixing the clipmap meshes in view space.

Imagine the intersection of the view frustum with the plane of the clip maps. It forms a large trapezoid that extends out to infinity. The view dependent approach fills this trapezoid with triangles that become denser closer to the camera and more spaced out as you move away from the camera. On the GPU, view space is transformed back into world space and then the world space coordinates are used to sample the height map. The primary advantage of this approach is it removes the need to perform view frustum culling of the clipmap patches, which Asirvatham admitted to having to do manually each frame. An overhead view of the view dependent clipmap approach is shown in Figure 1.

I implemented the view dependent approach and ran into some interesting problems that make it apparent why Asirvatham et al choose to orient the clipmaps in the x,z plane. When the view frustum is oriented at an even 0, 90, 180, 270 degree angle everything looks perfect, but as you rotate the camera in between these angles you get a stair-stepping effect in the terrain. This is because the points sampled on the heightmap are no longer "even" sample points, and you get an effect similar to drawing a line without anti-aliasing. See Figure 2.

To combat this problem I tried solving it the same way you would if you were doing anti-aliasing: super-sampling. Instead of sampling at the point given by the transformation, I sampled 4 points around that point and averaged the results. This did indeed smooth the stair-stepping effect, but just like even anti-aliasing in a ray tracer, you're just offloading the problem further back into the scene–the stair-stepping was still there, it just wasn't as apparent.

The proper solution to deal with the stair-stepping effect was to have the vertex shader program take the floor of the transformed vertices. This guarantees sample points will only lie on "even" intervals, so the geometry will not change as the camera moves.

Pseudo-code of the shader program for the view dependent approach is given below. The shader program assumes patch data is along the x,z plane (y-up). The scaling factor `g_scale` is necessary to snap the vertices to the closest floor position–as the LOD decreases, the spacing between vertices needs to increase. `g_width` and `g_depth` are the x and z dimensions of the height map texture data in video memory.

```
VS_OUTPUT RenderVS(float4 vPos : POSITION)
{
  // transform vPos to world space
  float4 worldPos = mul(vPos, g_mWorld);

  // snap vertex to closest coordinate in height map
  worldPos.x = floor(worldPos.x / g_scale) * g_scale;
  worldPos.z = floor(worldPos.z / g_scale) * g_scale;

  // transform worldPos to texture space on the heightmap
  float4 samplePos = float4(newPos.x / g_width, newPos.z / g_depth, 0, 0);

  // sample height from texture
  worldPos.y = tex2Dlod(HMSampler, samplePos);

  // transform world space into view space
  Output.Position = mul(worldPos, g_mViewProj);
}
```

3 Exponentially-Sized Clipmaps

In the previous geometry clipmap work, care was taken to ensure that each LOD ring was exactly twice the size of the previous one, such that for every four triangles that formed a rectangle in a finer LOD there there were two triangles that formed a square in the adjacent coarser LOD. Organizing a triangle strip in this manner introduced minor complexity in that several carefully constructed vertex and index buffers needed to be created to make sure each finer set of triangles had matching coarser triangles aligned with them. Exponentially-sized clipmaps tosses this approach in favor of a brute-force approach that is much easier to implement.

Exponentially-sized clipmaps use a single vertex/index buffer structure to render the entire scene. In the middle are nine square triangle strips arranged in a square (a 3x3 tic-tac-toe). Around them are eight more of the same triangle strips twice three times the size of the previous. Around that is eight more three times the previous size, and so-on and so-forth, out to some cut-off. The end result looks like an exponentially expanding tic-tac-toe board.

Exponentially-sized clipmaps have the advantage that rendering them is greatly simpler than the previous approach. It's a trivial task to build nested for-loops to do the job. The disadvantage of this approach is instead of having a single T-junction at every set of triangles along the edges between the LOD's, you now have two. This is trivially corrected however: where the previous work stored a z_c at each vertex, it is now necessary to store a z_{c1} and a z_{c2}.

Like the view dependent approach, exponentially-sized clipmaps can fall victim to the stair-stepping effect if care is not taken to make sure the vertices fall on "even" boundaries. For this reason, it is important that the coarsest LOD moves in whole integer increments, and subsequent

4 Normal Blending

The previous geometry clipmap work handled normals by 2x super-sampling the height map and introducing fractal noise to "smooth" the terrain. This incurs a 4x overhead in memory costs to store the increased normal map, and also a minor performance decrease for the additional fractal noise texture. It would be easier and more cost-effective if normals could be stored one-to-one with their respective vertices.

Calculating normals on a per-vertex basis can be done two different ways: per-vertex in the vertex shader, or per-pixel using a normal map lookup in the pixel shader. Per-vertex creates reasonable results when the vertex density is high, such as a fine-grain LOD close to the camera, but produces undesirable results in triangle patches at coarser LOD. Per-pixel creates excellent results farther away from the camera (an effect identical to bump mapping), but closer to the camera the normals form blocks, which are undesirable.

Normal Blending is a technique that combines the per-vertex and per-pixel approaches by linearly interpolating between the per-vertex normal and the per-pixel normal. The end result is efficient and produces desirable output.

To further decrease the memory costs of the surface normals, the surface normals can be packed into the same texture as the height data. Modern graphics cards that support vertex texture sampling support a texture format known as A32B32G32R32F, a 128 bits-per-pixel format that stores four 32 bit floating point values in one texture sample. The normal blending technique I've implemented stores height data as $(height, n_x, n_y, n_z)$.

The vertex and pixel shaders for normal blending both sample the height data to obtain a normal, but they use it in different ways. The vertex shader samples the normal at a vertex and outputs it to the pipeline. It also calculates a texture coordinate for the vertex and outputs that also. Here it's important to understand a little bit about the graphics pipeline on modern GPUs. The normals and texture coordinates that are outputted by the vertex shader are not simply passed directly to the pixel shader–because the pipeline is aware of the triangle structure of the vertex data, the pipeline interpolates the normals and texture coordinates before it hands them to the pixel shader.

The pixel shader takes the translated texture coordinate and uses that to again sample the height data and obtain a new normal. (If we were doing bump mapping we would stop here and use this as the normal). The pixel shader then linearly interpolates between the translated normal and the new normal that was just sampled and uses this normal to calculate lighting for the pixel fragment. The interpolation factor is determined by the vertices distance from the camera.

The pixel shader program that performs normal blending is given below.

```
float4 RenderPS(VS_OUTPUT In) : COLOR
```

```
{
  // L: light vector, normalized
  float3 L = g_light - In.WorldPos;
  L = normalize(L);

  // Vl: distance to camera
  float3 V = g_eye - In.WorldPos;
  float Vl = saturate(length(V) / 500.0);

  float4 sN = tex2D(HMSampler, In.UV);

  // linearly interpolate between vertex N and sample N
  // based on distance to eye
  float3 N = lerp(In.Normal, sN.yzw, float3(Vl, Vl, Vl));

  // simple diffuse shading
  return saturate(dot(N, L));
}
```

5 Single Height Map

Geometry clipmaps were designed for big terrains, but most applications (i.e., games) are not going to be using gigantic terrains. The memory requirements of my implementation up until now requires 4 bytes per vertex, which includes the normal data. On a graphics card with 64 megabytes of memory, this could theoretically store a 4096x4096 height map all at once[1].

The approaches I've outlined above all work great with a single chunk of height map data. If the tiny gaps between LOD patches are not desirable, the size of the patches can be increased such that they overlap. This works great with realistic terrain data, but produces obvious artifacts when the terrain data comes from mathematical functions.

This technique suffers from aliasing when the height data is at a higher frequency than the LOD vertex width. For this reason, this technique should only be employed on low-frequency data. Again, the results are reasonable on realistic terrain data, but the aliasing is more apparent when the data comes from mathematical functions. Figure 3 shows the aliasing problem with the function $sin(x)cos(z) + sin(x/2)cos(z/2)$.

[1]Actual graphics cards have limitations on texture dimensions, so the height map would actually need to be split up into smaller chunks, such as 1024x1024.

6 Results

I have benchmarked the performance of rendering terrains using four approaches:

1. Complete mesh. The entire terrain is loaded into a vertex buffer and drawn as a triangle list.
2. 2x sub-sampled. Same as the above, except only every-other vertex is used (in both the x and z directions).
3. 4x sub-sampled. Same as the above, except only every fourth vertex is used (again, in both the x and z directions).
4. Exponentially-sized clipmaps with normal blending.

The terrain data used was a 1.39 million point height map (1392x998) of the Corvallis, Oregon area, taken from USGS digital elevation map data. The performance results are given below.

Technique	FPS
Complete mesh	1.5
2x sub-sampled	7
4x sub-sampled	45
Geometry clipmaps	65

7 Discussion

I have presented several different ways of approaching geometry clipmaps. Each approach works with varying degrees of success depending upon their application.

View Dependent Geometry Clipmaps produce artifacts, but these artifacts are only distracting to the viewer when the terrain data comes from mathematical functions. On actual terrain data the artifacts are not as noticeable, especially if the viewer is moving at a high speed. A combination of blurring or a depth-of-field effect with view dependent geometry clipmaps could produce very desirable results at little cost to the GPU.

Exponentially-Sized Clipmaps produce reasonable results for certain applications. If a linear LOD degradation is desirable then this is probably not a good approach. Implementation for this approach however is far easier, and most likely yields better performance than traditional geometry clipmaps because there is significantly less geometry and transformations involved.

Normal Blending is a quick and dirty way to avoid the expensive memory costs of storing a double density normal for 2x sampling. This technique could probably be combined with a small fractal noise texture to get even better results.

Single Height Maps are only appropriate for applications where the size of the height map is small and the height data is low frequency (to avoid aliasing, discussed previously). For a large scale terrain the height map data will need to be recalculated in a background thread and fed to the GPU.

More work needs to be done to refine these approaches, in particular view dependent geometry clipmaps. Out of the approaches described, I believe normal blending shows the most promise. Normal blending significantly reduces the video memory required for geometry clipmapping, which will also have an impact on performance.

References

[1] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps, 2005.

[2] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

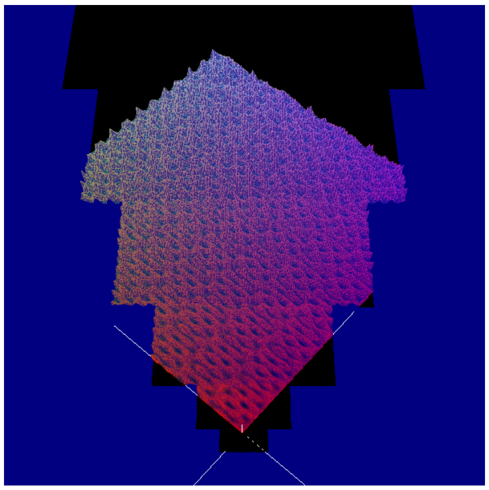

Figure 1: View Dependent Clipmaps (Overhead view)

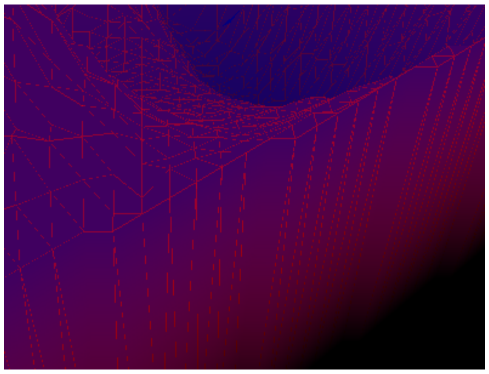

Figure 2: View Dependent Clipmaps (Before Flooring)

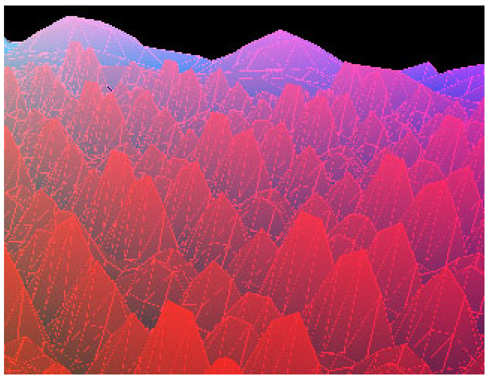

Figure 3: Single Height Map: Aliasing