GPU Terrain Rendering

A Paper by Harald Vistnes (herald@vistnes.org)

Presented for CS525, April 11, 2007

Michael Schuld

4/11/2007

GPU TERRAIN RENDERING

A Paper by Harald Vistnes (herald@vistnes.org)

OVERVIEW

In general terms and in most algorithms of the past, rendering of terrain with levels of detail is done with a single update to all of the vertices in the terrain's buffer every frame or every time the camera moves. To accomplish this sort of updating in real time requires a lot of processing by the CPU and uses up much of the needed frame time that could be better allocated elsewhere in a game or simulation. To solve this problem, the author of this paper has moved all height field calculations of terrain vertices to a vertex shader running on the GPU at render time, which allows for a much smaller amount of data to be stored in the terrain system, and for much quicker calculation of the height values overall while leaving the CPU free to work on other important tasks.

ALGORITHM EXPLAINED

The basic idea of the algorithm uses the same general methods of breaking up terrain that is seen elsewhere in other terrain rendering algorithms. The terrain is broken up into blocks of (2k+1) by (2k+1) which easily maps each vertex in the sub blocks to one of (2n+1) by (2n+1) vertices in the overall terrain. Because of this method of breaking up the blocks, a single vertex buffer can be used to represent the whole set of blocks along with a much optimized index buffer that lists the triangles in a strip (the fastest method of rendering a group of triangles.)

Two factors in the actual shader are needed as well to make the algorithm work. One of the two factors is the scale of the current terrain block. Using this scale value, any size block can always be sent to the shader, and the correct vertex positions based on that scale can be used in selecting the vertex heights within the shader. This allows for a great number of possible level of detail algorithms to be implemented using different sized blocks based on any number of parameters. The second factor is a simple offset value for the top left corner of each sub block of terrain. Here is a simple vertex shader for the offset and scale transformation:

```
float4 VS(float3 pos : POSITION) : POSITION {
    float s = pos.x;
    float t = pos.y;
    float u = s * uScale + uBias;
    float v = t * vScale + vBias;
    float4 tex = float4(u, v, 0.0f, 0.0f);
    float h = tex2Dlod(heightfield, tex).x;
    pos = float(u, h, v);
    float4 pos0 = mul(float4(pos, 1.0f), matWorldViewProj);

    return pos0;
}
```

Level of Detail

As a result of the method of block dividing the terrain, and because of the size constraint on the blocks, the terrain can actually be divided up using a quadtree type of approach, which maps very well to different level of detail algorithms. Little detail control in this algorithm is done by recursively evaluating each block and deciding whether to render or divide it into four smaller blocks. A simple distance based method is used by the author of this paper to allow for faster development, however many other methods could be used to decide the level of detail of each block. The equation used for distance evaluation in this example is:

```
l/d < C
```

where l is the distance from the center of the block to the camera and d is the world-space extent of a single triangle. C is a constant the controls the quality of the terrain. Here is some sample code using DirectX the shows how you can render the terrain with level of detail control.

```
void Render(float fminU, float fMinV,
    float fMaxU, float fMaxV,
    int iLevel, float fScale)
{
    float fHalfU = (fMinU + fMaxU) * 0.5f;
    float fHalfV = (fMinV + fMaxV) * 0.5f;
    float d = (fMaxU-fMinU)*m_matWorld._11/(m_iBlockSize-1.0f);

    D3DXVECTOR3 c(fHalfU*m_matWorld._11,0,fHalfV*m_matWorld._33);
    D3DXVECTOR3 v = c - g_Camera.GetPos();
    float l = D3DXVec3Length(&v);

    float f = l / d;

    if(f > m_fLOD || iLevel < 1){
        Draw(fMinU, fMinV, fMaxU, fMaxV, iLevel);
    } else {
        Render(fMinU, fMinV, fHalfU, fHalfV, iLevel-1, fScale/2);
        Render(fHalfU, fMinV, fMaxU, fHalfV, iLevel-1, fScale/2);
        Render(fMinU, fHalfV, fHalfU, fMaxV, iLevel-1, fScale/2);
        Render(fHalfU, fHalfV, fMaxU, fMaxV, iLevel-1, fScale/2);
    }
}
```

Because of the method of determining the level of each block, some popping will occur as blocks move from one terrain level to another. A way that is discussed to solve this problem is to use multiple texture lookups and a linear interpolation of the values in the shader to make a smooth transition between detail levels. Another problem with the current method is that levels next to each other that are different will show cracks. A simple, and what the author calls ‘elegant’ solution is used with skirts to fill in the holes.

Culling

Frustum culling is built in to the system fairly simply by adding a call to a camera or scene manager with the coordinates of the current block and its size. A very simple test can be used to tell if the block's rendering or recursion should be called.

Normal Calculation

One of the major issues with level of detail terrain algorithms is that the normals on the terrain change every time the location of a vertex changes hen a level is incremented or decremented. To help with this calculation, the GPU is once again relied on and implements a previously known equation that uses neighboring height values to get a vertex's normal. The equation used is as follows:

```
N = {(w-e), 2d, (s-n)}
```

In this equation, d is the distance between the vertices, and the directions: n, s, e, w are their obvious counterparts. This method of calculating the normals works very well. It correctly allows for dynamic lighting algorithms to show the whole terrain. One problem however, is that the terrain actually changes its lighting with the levels of detail, which may be drastic and noticeable even with the interpolated changes discussed above.

The suggested method of solving this problem is to fall back on a more common method of getting normals for a terrain map and calculating them before hand into a normal map (another texture) and simply looking up the value there. Using a normal map there is one (or two with interpolation) texture lookup per vertex, where the calculated version used four making the normal map method much preferred over the former. The only downside of the normal map is that it requires more texture memory.

Collision Detection

As the height map's data and the final outcome of its use are now being fully separated into different systems, some of the problems that need to have both available to it arise. The major problem that needs both the data behind the height and the final outcome is collision detection. Since the actual heights of the terrain are not being stored in any format directly accessible by the CPU, an alternative method of passing this information has to be developed. In this paper, the author has chosen to use a simplified version of collision processing at a specific single point. This point's coordinates are sent into the shader like every other parameter, and the height at that point is written out to a separate one dimensional texture that can be checked against by the CPU separate from the main rendering system. Other similar methods with larger point grids can be used as well, but only a single point was necessary to satisfy the needs of the author's collision problem.

Difficulties in Implementation

- Texture size vs. Memory Size
- Height accuracy vs. Data Types
- Texture filtering and Smoothing

Summary and Results

Using the method presented in this paper, the author was able to successfully demonstrate height map based terrain rendering of images up to 2049x2049 resolution with multiple different block sizes at speeds comparably equivalent or better than those of other algorithms and methods in use today, while leaving the CPU essentially free to work on other tasks.

Bibliography

Vistnes, Harald, "GPU Terrain Rendering." *Game Programming Gems 6,* Charles River Media: pp.461-471, Color Plate 8, 2006.