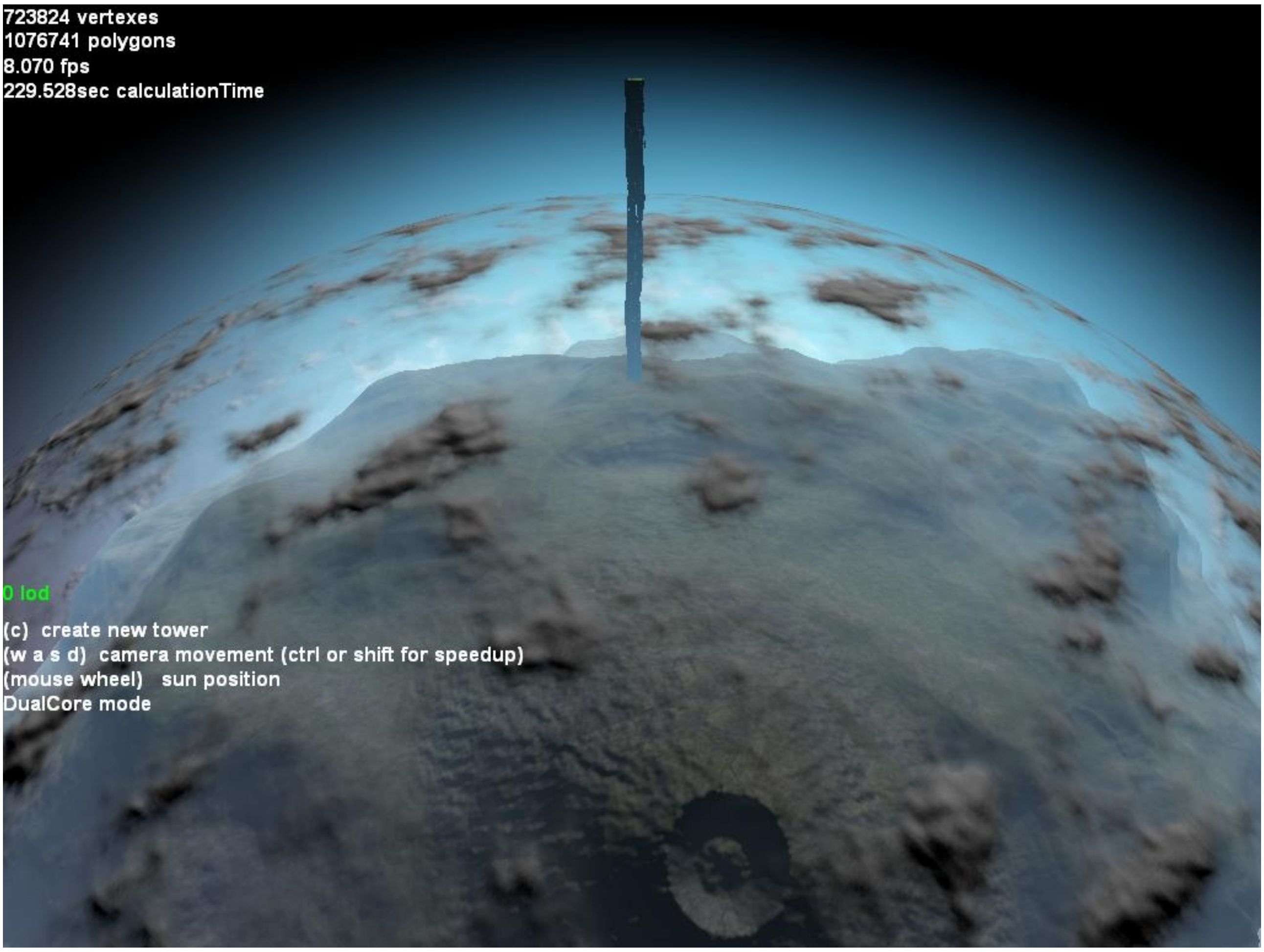
Tower on a planet

shader demo

Christopher Bläsius
august 2009
www.swollen-eyeballs.org



723824 vertexes
1076741 polygons
8.070 fps
229.528sec calculationTime

0 lod

(c) create new tower
(w a s d) camera movement (ctrl or shift for speedup)
(mouse wheel) sun position
DualCore mode

Index

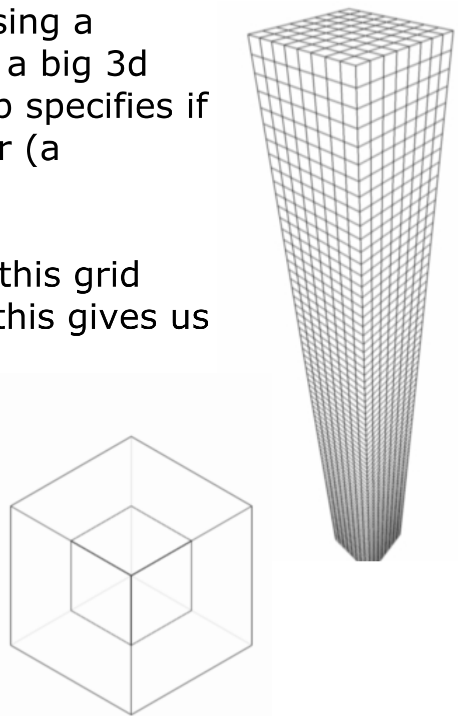
1 Creation of the tower

1.1 filling a density map

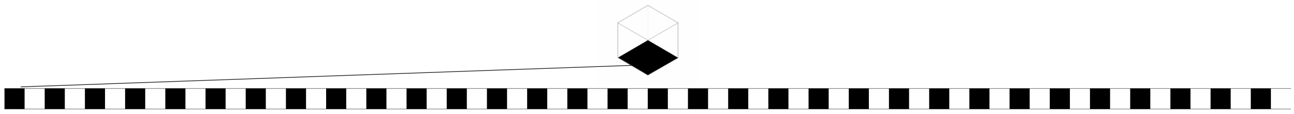
The triangle mesh of the tower is build by using a density map. Basically you can think of it as a big 3d grid (256x256x2560). Each point in this map specifies if there is material (a positive value) or just air (a negative value).

To keep memory consumptions low we split this grid into smaller parts, 32x32x32 to be precise, this gives us 8x8x80=5120 smaller grids which get calculated separately.

To avoid shading errors we let those smaller maps overlap each other by 25%, this gives us grids of 64x64x64 voxels.



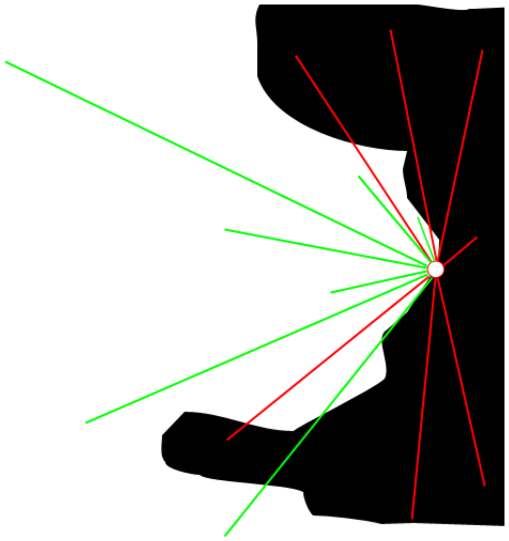
These maps get internally represented as big 2d textures (4096x64 pixels, floating point format).



This texture get filled by a shader which uses a 3d noise texture and the world position to create density values. To get a 3d position out of the 2d position on the texture a simple formula is used:

pos.y = texture.y + map.y
pos.z = floor(texture.x/64) + map.z
pos.x = texture%64 + map.x

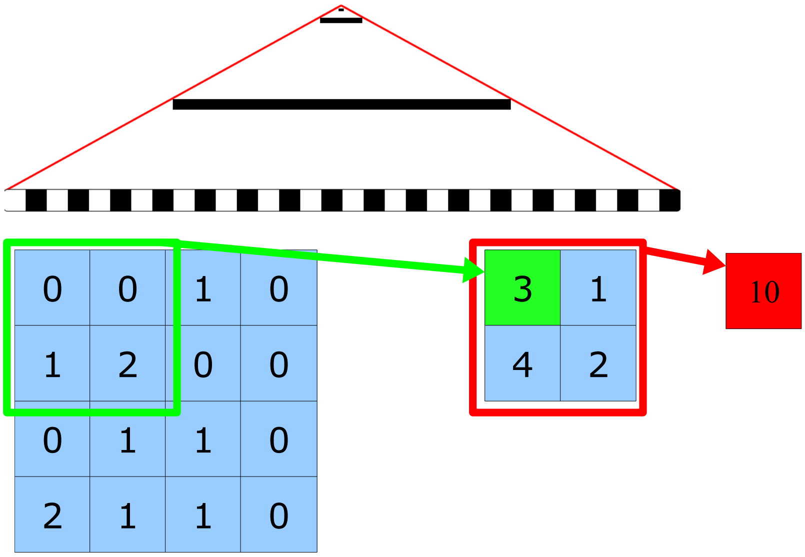
1.2 calculating ambient occlusion term



To calculate a simple ambient occlusion term we read the density map 128 times in random directions around every position. If the value is positive it blocks lights, means the ambient occlusion term gets smaller. This is done by a shader which writes the results out to a 1024x32 pixel texture (the overlapping parts gets deleted). As you can imagine this doesn't give very accurate values, thin walls are often completely ignored by the algorithm, although they also should block light. This could be solve by shooting rays instead, means reading the density map along a ray and stop if the ray hits a wall. I've tested this, but its to damn slow on a Geforce6 which doesn't have effective conditional branch support.

1.3 summing up polygons

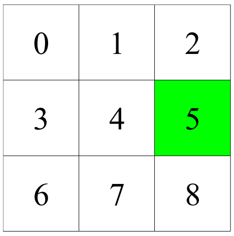
On this texture we also save the index of the marching cube (there are 256 different cube types) and the number of polygons we need to construct the mesh. (i won't explain the marching cube algorithm here, there are pretty good explanations on the web).
After this we construct a so called HistoPyramid. We sum up the number of polygons until we have a 1x1pixel texture which holds the total number of polygons we need for the complete sub map.



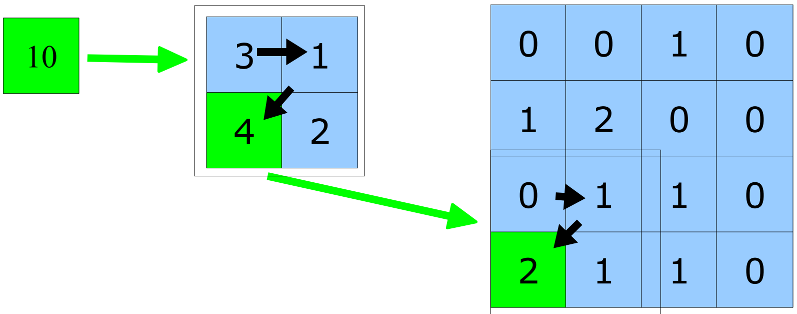
For this we create 4 extra textures, we need those for calculating the vertex positions later.

1.4 vertex creation

Each polygon is defined by 3 vertexes. The position of each vertex gets calculated by a shader writing to a 256x512px texture. Each pixel on that texture represents a vertex. Using the intermediate textures we created a step earlier we can find out to which marching cube a vertex belongs, means we can calculate the 3d world position of the vertex.



Texture holding the vertex positions (numbers are the index of each polygon (each cell consist of 3 vertexes))



The polygon with index 5 is the first polygon of the green cell

This way we get also a 2d position on the density map which we use to interpolate the vertex positions.

If you want to learn more about this i recommend reading this paper: http://www.mpi-inf.mpg.de/~gziegler/hpmarcher/techreport_histopyramid_isosurface.pdf

1.5 create vertex normals

Calculating the vertex normals is done with a shader which writes to a 512x512px texture. Each vertex now consist of 2 pixels in the textures (position x y z, normal x y z, 2*ambient occlusion term).

1.6 additional work on cpu

We read the last texture back to system memory and copy it to a vertex buffer of a mesh. Vertexes which are on the same position get merged, this resolve in a smoother shading and removes thousands of unneeded vertexes. Additional i create several lod meshes using face collapsing.

traps

The creation of the vertexes is the most difficult part. A gpu doesn't have an integer unit, so you must always work with floats and clamp the output to integers, this is needed for correct addressing of the intermediate textures of the histopyramid. Another thing are loops in shaders, the shader compiler always tries to unroll them which can result in wrong code when using loops with conditional statements. Unrolled loops are a must for Geforce6 hardware but are counterproductive on Geforce8 or newer hardware.

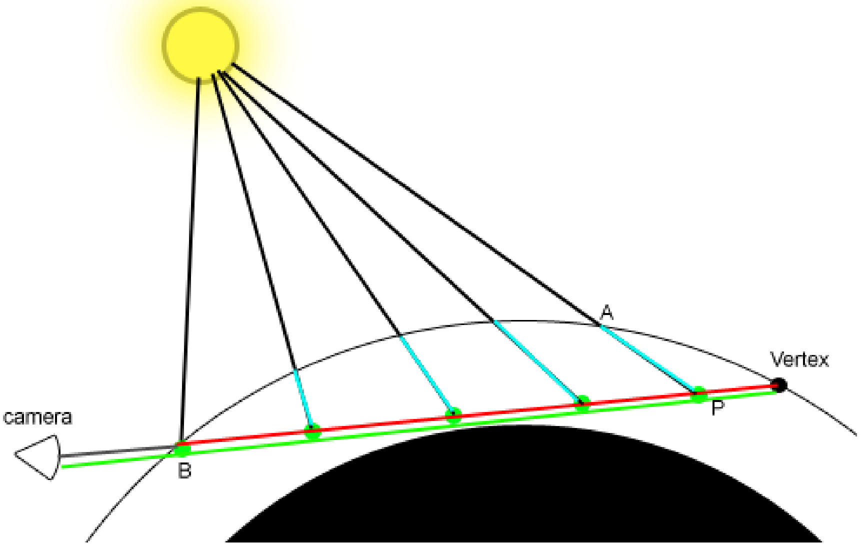
2 Rendering

2.1 mesh deformations

The terrain and the sky model are normally flat, a vertex shader deforms them to hemispheres. Nothing special but i does its job.

2.2 atmospheric scattering

The sky, the clouds, the terrain and the tower uses a simple atmospheric scattering algorithm which is in noway accurate or optimized.



For each vertex a ray from the vertex position to the camera position gets calculated. On the segment of the vertex to point B, 9 different points P gets calculated. For each point P the length of the segment $\overline{PA}$ is evaluated, the smaller this segment is the more light from the sun is reflected from small particles in the air at the point P. The blue part of the sun light gets 10 times more often reflected then the red part, this is what makes our sky blue. But the longer this scattered light must travel to the atmosphere towards the camera (the segment $\overline{PB}$) the more light gets blocked by particles in the air (more blue light gets blocked then red one). This is what makes a red sky on sunsets/risings.
Another thing you must consider is that the air consist more dirt particles at the ground.

*Light on point P = max(dirtiness*length($\overline{PA}$)*rgb(0.1, 0.5, 1.0) –*
*dirtiness*length($\overline{PB}$)*rgb(0.01, 0.4, 1.0), 0);*

I've got those values by experimenting.
The light of all points get summed up and added to the final color of the object.

2.3 simple clouds

The texture of the cloud model gets overlayed 4 times at different scales and positions to get a density. This is done 8 times, 1 time to get the transparency value, and 7 times slightly offset to calculate a normal out of the densities. This normal get used to light the model.

2.4 texturing the tower

The shader of the tower uses a simple 3 planar mapping for the texture. Sometimes called Cubemapping in modeling applications. It samples the textures 3 times, using the yz, xy and xz world position as texture coordinate. These 3 textures gets blended according to the vertex normals, this way almost no stretching errors or seams are visible.

Christopher Bläsius
chris[at]swollen-eyeballs.org
16.08.2009

Thanks for reading, and ignoring all the typos. ;)