Fast Realistic Rendering of Global Worlds using Programmable Graphics Hardware

Nick Gebbie
University of California San Diego

Mike Bailey
Oregon State University

Abstract

Interactively rendering geo-spatial scenery is a recurring theme in game development. This paper demonstrates a method of handling much of that rendering on programmable graphics hardware. This makes better use of the power available on the GPU, and frees the CPU processing time for user interaction, simulation, and data acquisition and manipulation.

Introduction and Previous Work

There have been many methods proposed to rendering large terrains in real-time computer graphics systems. The amount of data available to render in terrains has always exceeded the available rendering time, making the issue of terrain level-of-detail an old problem. Most algorithms for terrain rendering focus on optimizing mesh-based algorithms for removing or inserting vertices based on some error criteria.

A common method of handling terrain LOD is to build a large mesh, and selectively pick a vertex to add or remove from the mesh. When using a View Independent Progressive Mesh (VIPM) [6], the order of vertices to remove or add is precomputed, and does not depend on the camera position. For large terrains, where large areas may not be visible at all, VIPMs are less applicable. VIPMs are more typically applied to non-terrain meshes. View Dependent Progressive Meshes (VDPMs), as the name suggests, determine at run-time which vertices to remove or add. Typically, vertices are selected by determine the screen-space error induced by removing or adding the point. There have been many papers on VIPMs and VDPMS, such as [4], [7], [9].

VIPMs and VDPMs continually alter the rendered geometry, so the mesh must be retransmitted to the GPU whenever the detail needs to change. Transmitting this data takes a noticeable amount of time, and so we prefer to use static geometry, stored in Vertex Buffer Objects (VBOs) as often as possible. When using VBOs, OpenGL can move the geometry to the GPU's memory, and avoid the transfer time from the CPU. A goal of this project was to ensure that our scheme can utilize the massive performance boost gained by using static VBOs.

Rather than operate at the level of a single vertex, geomipmapping [3] breaks the large terrain into square chunks, also called zones. Various levels of detail are pre-generated for each zone, independent of the view. At run-time, a simple error metric is calculated for each zone to determine which LOD to render. Typically, this metric is based solely on the distance from the eye position to the center of the zone. As different zones may be operating at different levels of detail, care must be taken to ensure that there are no T-intersections[1] between zones, which can cause cracks to appear between zones.

Unlike progressive meshes, each LOD can be stored in a static VBO, as it never changes. This method uses a large quantity of GPU memory, so very large terrains require more elaborate managerial schemes. LODs for zones can be paged in at appropriate times, so that only the required level of detail is present. The zones can be maintained in a quad-tree, rather than a regular grid, which greatly aids GPU storage of larger terrains.

Geomipmapping uses a very simple decimation of the geometry. The highest LOD contains every vertex, while every successively lower LOD skips a vertex. Each LOD, then, has roughly four times fewer points than the level higher. The name is an allusion to texture mip-mapping [5] which is appropriate, considering the similarity of the technique. Like progressive meshes, there have been other papers on this type of LOD, such as [22].

[1] A T-intersection occurs when a vertex is placed along an edge of an adjacent triangle, as opposed to being placed as a vertex of both triangles. This can cause rendering artifacts due to inaccuracies in the rasterizer. In the case of heightfields, large cracks will appear where T-intersections are allowed.

Geoclipmapping [10] introduced very recently, takes the zone-based approach of varying LODs, but avoids the precomputation. A set of nested square regions are maintained around the camera position. Each layer represents a successively lower level of detail. The smallest square, centered at the eye position, will be the highest level of detail. Geoclipmapping makes use of VBOs by maintaining each of these regions in a separate GPU buffer. When the camera moves, the buffers are updated incrementally, rather than switching to a different zone's buffer. In this way, the only GPU buffers in GPU memory are the buffers rendered, so GPU memory is not wasted. The update process can be costly in terms of bandwidth requirements, so geoclipmapping supports the option of not completing any given update. If the eye position is moving too fast, the highest level of detail may not be updated. This allows maintaining interactive rates very nicely.

Clipmapping [21], developed by SGI, is a proven method for handling textures that will not fit into available memory. It is a simple extension to mipmapping. Unlike mipmapping, the highest detail regions of the texture are only loaded in a small specified region of the texture. This region is the same texel size at every level, unless the mipmapped level is smaller than this region. If there are fewer texels available than the set texel size, the entire mipmap level is loaded. Geoclipmapping, in the last section, is the geometrical equivalent of this technique. The technique used in this paper is inspired by clipmapping, but is fundamentally quite different.

The OpenGL Shading Language

Originally, shaders had to be written in microcode, tailored specifically to the target architecture. As with CPU programming, high-level languages and compilers were developed. Microsoft's DirectX 9 supports a high-level language called HLSL [12]. Nvidia supports their own language Cg [11], which can be used with either OpenGL or DirectX. The OpenGL Architectural Review Board (ARB) has adopted a language called the OpenGL Shading Language (GLSL) [8,19]. The capabilities of all three languages are dependent on the hardware, and so are very similar. The differences are in the languages themselves and the interfaces used to access them.

GLSL was designed to look and feel like C, reducing the learning curve of the language. Variable types intrinsic to graphics, such as vectors and matrices, are supported. Likewise, common functions such as clamping, mixing, dot products, cross products and matrix multiplication are included. Many of these functions are implemented in hardware, and having a direct call in the language enables these functions to be compiled into single instructions.

GLSL allows two types of variables to be passed into the vertex processor. Vertex attributes such as normal, color, texture coordinates, and the vertex position are accordingly called *attribute*s. Run constants, such as light locations, textures used, and material properties are called *uniform* variables. *Attribute*s are expected to be different per vertex. Uniforms are typically constant for a set of vertices, and are more expensive to update. These values are read-only to the shader. Variables passed to the rasterizer, which are interpolated and passed to the fragment processor are called *varying*. These variables are write-only in the vertex processor, and read-only in the fragment processor. These variables are the only link between the two shaders.

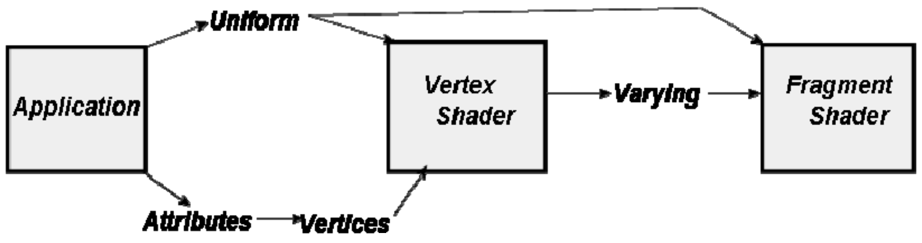

Figure 1: Variable Types in GLSL

The Spiral

Instead of breaking the terrain into regions, we generate a single abstract mesh (in a single VBO), then wrap it around the target geometry using custom functions written in GLSL. For example, if we wish to render terrain on a globe, we can interpret the single mesh's positions as latitude and longitude coordinates, and generate the vertex position using a polar coordinate transform. We can just as easily render the terrain to a flat surface by altering this transform. Using a single geometry with this naïve system, however, means that the geometry rendering will never change. This is equivalent to just rendering the entire terrain at a single LOD, and so needs some improvement.

The first improvement is to alter our projection to account for a focal point, derived from the camera's position. We supply this focal point, in latitude and longitude, to the vertex shader, and add this coordinate to the positions provided by the abstract mesh before projecting the coordinate into world-space. This means that the rendered geometry is always positioned relative to the focal point. If we make the focal point the location of the geometry directly below the camera, the rendered geometry will follow the camera.

The next step is to handle varying levels of detail. In general, we will want the highest detail to be the closest geometry to the camera. Since our geometry is all projected relative to the camera, we merely need to increase the detail of our abstract input mesh near the origin. Furthermore, since we want our single mesh to handle all levels of detail, we'd like the mesh to retain this relationship as the camera moves further away from the input.

An example of a mesh that fits these criteria is an exponentially increasing spiral. We select which region of the spiral to render based on the distance from the camera to the focal point. The spiral is wound tighter near the focal point, so as we zoom in, more geometry will appear. As we zoom out, we don't render the smallest geometry, and increase the maximum rendered. This leaves a hole in the center, but that can be easily and cheaply filled with a single triangle fan. The exponentially increasing nature matches the rate triangles shrink as the camera changes zoom, so the spiral appears the same density and renders the same number of triangles at all distances.

All this together means that we can render a terrain at wildly varying distances using the same number of triangles, preserving triangle density, at any location over our terrain.

Creating the Spiral

To generate the spiral, we define a variable called *spin*. *Spin* corresponds to how far around the spiral, in total degrees or radians, we have gone. The spiral function is given in terms of spin only, and outputs the vertex to be rendered.

$$f_x(spin) = \sin(spin) \bullet 2^{spin}$$
$$f_y(spin) = \cos(spin) \bullet 2^{spin}$$

This equation alone, however, does not yield a useful spiral, because it increases radius too quickly. We introduce a number of controls to the spiral generation, which allow us to tweak the spiral for optimal speed and geometric density.

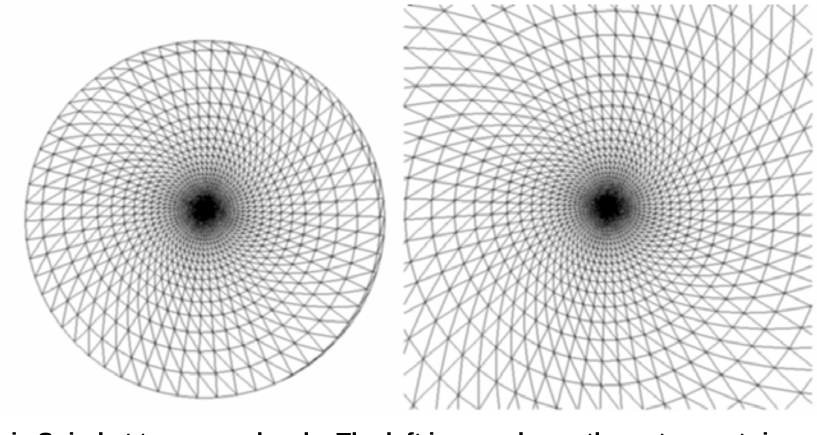

Figure 3: The Basic Spiral at two zoom levels. The left image shows the outermost ring of the spiral. Note that the triangle density in both images is the same, independent of the zoom level.

The constant *spiralrate* alters how fast the spiral expands outwards by modulating the power of the exponent. The value of *spiralrate* itself is hard to quantify, so we instead calculate it from a parameter *spintodouble*, which specifies how many revolutions the spiral must undergo to double its radius. The constant *baseradius* determines the smallest density we will need. Since we are dealing with latitude and longitude, this value may be quite small if we want high precision. Our new equations are:

$$spiralrate = 1/spintodouble$$

$$f_x(spin) = \sin(spin) \bullet baseradius \bullet 2^{spin \bullet spiralrate}$$

$$f_y(spin) = \cos(spin) \bullet baseradius \bullet 2^{spin \bullet spiralrate}$$

Rendering the Spiral

Generating geometry using the spiral equation is performed as follows. We first define another variable *spinstep*, which determines the amount of increase in spin between each point. This value also determines the number of triangles in a full revolution around the origin. We generate a pair of triangles with the spin values *spin*, *spin+spinstep*, *spin+360*, and *spin+spinstep*+360, for successively increasing values of *spin*. Every triangle created can be connected by a single triangle strip. If we do not ensure that *spinstep* is a whole number divisor of 360, adjacent revolutions around the spiral will not match, causing cracks in the terrain (see Figure 4). This geometry is only created once, then moved to GPU memory via the OpenGL extension *vertex_buffer_objects*.

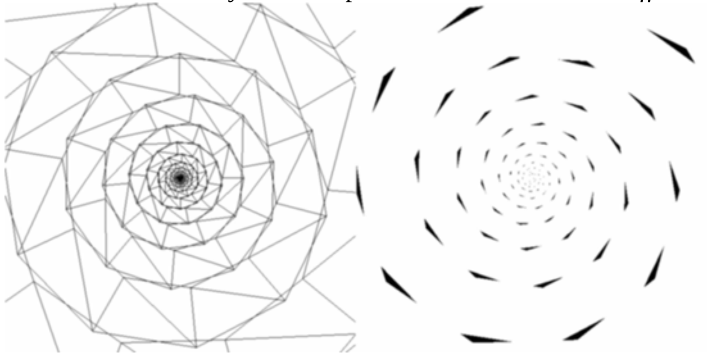

Figure 4: A spiral created with a *spinstep* that is not a even divisor of 360. The right image shows, in black, locations where no geometry is rendered.

To render, we only need to select a region of the spiral to submit to the graphics pipeline. We can select a region in terms of spin or radius, although radius makes more intuitive sense. From either parameter, we simply determine which index in our vertex buffer matches closest to the given spin/radius minimum and maximum, and then render that segment of the vertex buffer. But, how do we know which part of the spiral to render? Calculating the exact required minimum and maximum radii is not possible, given the constraint of arbitrary projection. We need to find some other means of feedback to determine which region to render. Our first approach was to simply control the min and max radii by some simple function dependant on the distance to the camera. This only works, however, if you know which projection is active, because differing projections have different effects on the zoom level. On the positive side, the amount of data per projection was minimal, requiring a single float with which to multiply both the min and max. This is functional, but not perfect.

Radius Correction

The spiral introduces a notable bias in the balance of triangles rendered. Even when cropping the smallest, most detailed triangles, the triangles rendering in the center of the screen are far smaller than those near the edge. Ideally we would like to balance this without incurring performance penalties. Since we are using vertex shaders, we could simply recalculate the radius of the point on the spiral before projecting the point, based on the current range of radii being rendered. Even a simple formula has noteworthy benefits to the balance of triangles. The following formula was used to generate Figure 5.

$$newradius = \sqrt{\frac{radius - minzoom}{maxzoom - minzoom}} \bullet (maxzoom - minzoom) + minzoom$$

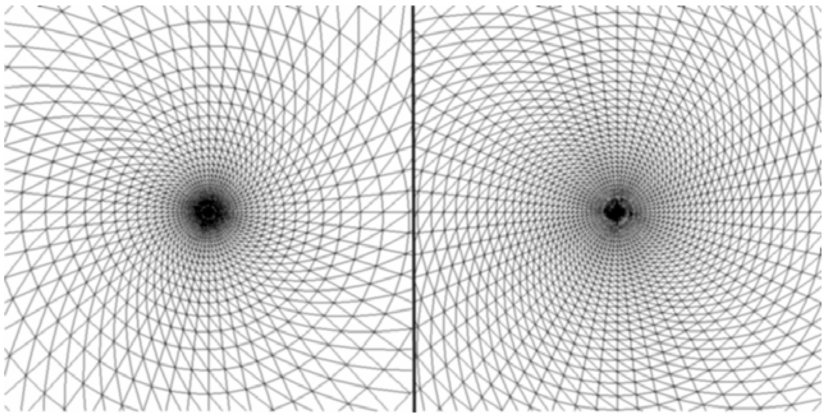

Figure 5: The spiral before and after radius correction. Note the more even triangle density in the right image. In comparison, the outside triangles are smaller, and the central triangles larger.

We considered using a logarithmic function to offset the exponential nature of the spiral. However, square roots have good hardware support on GPUs, due to the frequent use of square roots for normalizing vectors. The results of this formula are good enough that the cost of using a log is too high for the benefit gained. We calculated the radius from the input latitude/longitude offsets in the vertex shader, modified the radius, and then recalculated the latitude/longitude offsets. We found that this approach is straightforward to implement, but expensive to run, because it relies on a square root. When we removed the first step we were able to greatly improve the speed, as this step required inverse trigonometric functions.

In order to eliminate the initial radius calculation, we supplied the normalized direction of the latitude/longitude offsets in the x and y coordinates, and the radius in the z coordinate. The shader performed the formula directly on the z coordinate, then multiplied the result into the x and y coordinates, thus removing any necessary trigonometric functions. The bandwidth used by the vertices was slightly increased, but since the vertices are stored on the graphics card and the size of the buffer does not approach the size of modern graphics memory, this cost is minimal. The new spiral function became:

$$f_x(spin) = \sin(spin)$$
$$f_y(spin) = \cos(spin)$$
$$f_z(spin) = baseradius \bullet 2^{spin \bullet spiralrate}$$

The globe projection creates a hemisphere from the latitude/longitude offsets, then rotates the hemisphere to the focal point. This has a number of other benefits. The hemisphere was being created with the range of -90 degrees to 90 degrees only, and the focal point is at zero. This means that we can use a relatively low order Taylor expansion in place of the built-in sine and cosine functions, and save some processing time. This results in artifacts near the edge of the screen, but in practice these artifacts are not noticeable unless the camera is rotated without informing the projection vertex shader. Unfortunately, if we drop the strict offset restriction, as we have done, we no longer have easy access to the rendered latitude and longitude. These values are extremely useful for generating texture coordinates.

Managing Elevation

So far we have ignored the elevation component of the terrain. Because the vertices in the spiral move over the terrain, the elevation component cannot be contained with the vertex itself. Instead, we encode the height into a texture, perform a texture lookup in the vertex shader, and modify the projected coordinate based on the elevation. We modify the output vertex based on the elevation. However, this requires a texture lookup in the vertex shader. This feature, called *vertex textures*, is fully supported by the OpenGL Shading Language [20], but only very new GPUs support it. At the time of this writing, no hardware was available that could perform vertex texture fetches, so we were unable to include this in our work. Instead, we utilized bump-mapping, which provides the illusion of height without actually perturbing the vertices, and therefore does not require a vertex texture fetch.

Globe Textures
Rendering the earth in many projections is useless if the resulting product is not recognizable as earth. When we place annotations on top of the globe, we will want to know where the annotations lie. We will shade the globe with multiple textures given the constraints provided by the geometric level of detail, and our goals of efficiency and realism.

Cube Mapping
Cube map textures [15] are designed for lookup by directional vectors. Cube maps consist of six textures, one for each axial direction. The six textures compose a cube. When a 3D texture coordinate is used to lookup in a cube map, it is as if the vector is placed in the center of this cube, and the color at which the vector points is the resulting color. Cube maps are often used for reflection mapping as a replacement for environment maps. Unlike environment maps, cube maps are view independent when used for reflection mapping. Also, cube maps can be generated easily on the fly, as there are no non-linear distortions involved. Cube maps are slightly distorted in the edges of each texture, but this is because the texture is represented as a cube and not a sphere. The corners of the cube appear stretched to compensate for the distortion. Cube maps can be generated easily by performing six renders, once each at 90 degree field of view and down each axis and its negation. The perspective calculations handle the distortion automatically.

Textures as Lookup Tables
Graphics hardware provides very fast texture lookups. Using a texture as a function lookup is extremely fast compared with most approximations. The drawback of this method is the necessary mapping to texture coordinates, and issues with numerical precision. Mapping to texture coordinates is not an issue in our case. We are actually using the texture as the mapping mechanism. This leaves the problem of precision.

There are two types of precision to be concerned with when using lookup textures. The first is the total number of texels in the lookup texture. This determines the granularity of the output data. The texture lookup can use bilinear filtering to produce intermediary results between texels, but this does not create any additional data. The other type of precision has to do with the number of bits per texel component. A typical RGB texture only has eight bits per component, leaving only 256 available values. OpenGL does have a texture format with sixteen or thirty-two bits of precision, but not all hardware yet supports it. Both forms of precision need to be high enough to produce good results.

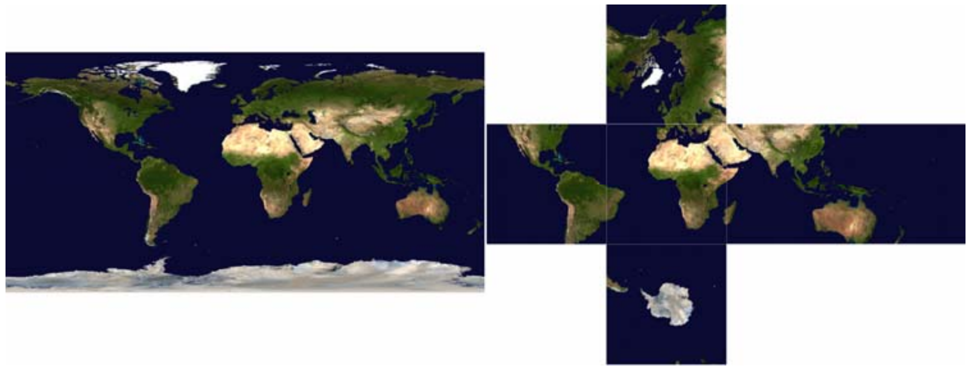

Figure 6: The Earth as a latitude/longitude texture and as a Cube map texture. Note that the distortions near the north and south poles in the latitude/longitude texture are gone in the cube map, but the distortions at the boundaries of each cube face are introduced, as is most visible on the Arabian Peninsula and Australia.

The precision required to map simple textures onto the surface of our globe is fairly small. Our globe textures are typically 1024x512 or 2048x1024, which requires only eleven bits of accuracy. We need two of these because we need two outputs, for latitude and longitude. The lookup function is quite linear, so we probably do not need many texels in the lookup texture. For this purpose, a lookup texture may be costly in terms of texture memory, but the function should be accurate enough.

Using Cubemaps as Globe Textures

Instead of generating two-dimensional texture coordinates using a cube map as a lookup, we can convert the latitude/longitude textures into cube map textures and use them directly. This avoids singularities at the poles, and spreads the texel detail more evenly throughout the globe.

Because we are using a sphere, generating the texture coordinates for the cube map is well-defined for the globe projection. We copy the output coordinate into the texture coordinate and normalize it. The flat projection, however, needs to generate this coordinate, although it did not have to originally. The flat data projection suffers, but the globe data projection is enhanced. Because the globe data projection was already more complicated, this actually balances the two, and improves the situation. Increasing the complexity of the globe projection would cause problems by exceeding the available resources to run the shader.

Converting a latitude/longitude texture to a cube map is not a difficult process. If load time is important, this can be done in a preprocessing step and saved as six pieces of the cube map texture. To generate every texel in the cube map, we calculated the 3D texture coordinate of the texel. This coordinate is a vector, so we reinterpreted this vector as a direction, and calculated polar coordinates for it. This gives us a latitude and longitude with which to lookup the latitude/longitude texture. Because this is done as a preprocessing step, we use bicubic interpolation when blending between texels in the original sample.

Shading

The vertex shader is responsible for transformation and projections, as well as generating texture coordinates. The fragment shader is responsible for shading. In fact, the shading model is completely independent of the projection and transformation, once texture coordinates are generated. This means that we only need to write a single fragment shader, and the shading at any given latitude/longitude vertex will be identical throughout every projection. We have selected to render the surface as if it were a sphere, as this is the most realistic lighting setup among the projections. The GLSL fragment shader code is given in Appendix A.

We selected five textures to layer together to create the final image, which was then modulated by per-pixel lighting calculations. Since we used GLSL, we were not forced to resort to multi-pass methods. The five textures are:

- A color map of the earth
- An elevation map of the earth
- A color image of clouds on the globe
- An image of earth at night (showing city lights)
- A water map, used as a specular mask.

The images were all obtained through NASA's blue marble website [13]

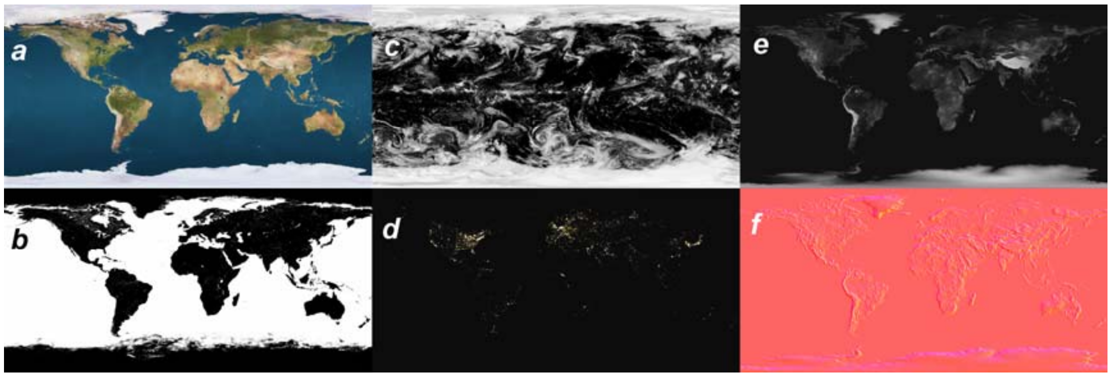

Figure 7: The textures used to render the Earth. a. Color map. b. Specular Mask. c. Clouds. d. Night light map. e. Elevation map. f. Processed elevation map into a normal map.

Bump Mapping

Because the vertices rotate over the surface of the globe, the normals of the surface cannot be included in the vertex buffer. Instead, we calculated the normals within the shader, because only the shader can calculate where the resulting vertex is. This is done using the elevation map and bumpmapping.

Bumpmapping [2] is a technique that takes a texture and uses the color channels to perturb the actual normal of the surface. The perturbed normal is used for lighting calculations. We process the elevation map into three components (stored in the red, green, and blue channels of the texture). The first component specifies how much of the original normal is in the final normal, and the second and third specify how much of the tangent and binormal vectors to include. The second and third values should range from -1 to 1, but textures only allow 0 to 1, so we need to divide the value by two and add .5 when creating the texture. The opposite of this needed to be done in the shader. We keep the original elevation in the alpha channel.

The normal, tangent and binormal vectors form an orthonormal basis typically called tangent-space. The tangent must be the vector tangent to the surface, yet increasing in the direction of the s-texture coordinate over the surface. The binormal increases in the direction of the t-texture coordinate, and is typically perpendicular to the tangent and normal.

We implemented bump mapping using the OpenGL Shading Language. The default normal and tangent of the surface needed to be provided by the projection. For the globe projection, the normal is equivalent to the normalized position. Calculating the tangent based on the increasing s-direction of the cubemap is difficult, however, considering the s-direction is different for every face of the cubemap. Instead, we processed the elevation map into the bump map before converting it to the cubemap, so that the s-direction is that of the latitude/longitude texture. This direction is the same as that of the lines of longitude, and is calculated given the normal. Given a normal (x,y,z), the tangent is (-y,x,0). The binormal follows lines of latitude, and is equivalent to the cross product of the normal and tangent.

Lighting and Texture Composition

The fragment shader is given the location of the light and the eye (through uniform variables) for calculating diffuse and specular lighting coefficients using the bump-mapped normal. The color map is modulated using the diffuse coefficient. A Phong [17] specular highlight is calculated using the bump-mapped normal as well, but this highlight is multiplied by the water mask texture, so that only water areas show the highlight. The clouds are not dependant on the surface elevation, so the clouds are modulated by the diffuse coefficient from the original normal. The clouds are also animated over time by multiplying the lookup texture coordinate by a rotation matrix provided by the application. Lastly, the night texture is added in using the negation of the diffuse coefficient from the original normal, resulting in it being blended into the dark region of the earth.

Despite the complexity of the textures used, writing the shader code to handle this sort of shading is well defined. The first version of this shader worked well, but was very inefficient. Optimizing the fragment shader is vital for efficiency, considering the number of times the fragment shader is run. For a small window at 640x480 resolution, this shader will typically be run up to 307,200 times per frame, and 18,432,000 times per second for smooth frame rates!

High-Resolution Texturing Level of Detail

Simple Texture Annotations

The amount of data collected for annotating the Earth is truly immense [23]. Single datasets now must be shipped on multiple DVDs. Clearly this presents a problem in displaying such data in real-time. Powerful desktops at the moment may have 1GB of CPU memory and 256 MBs of GPU memory. Detailed textures clearly will not fit within these memory constraints. Fortunately, rendering every detail simultaneously is never required. A large framebuffer, at resolution 2048x1536, with 32 bit color, is still only 12 MB. This implies that the largest amount of texture memory to be active every frame, for such a framebuffer, is only 12 MB. No detail beyond this could possibly be presented to the user. Keeping exactly this amount of texture in use is, of course, impossible, due to the limited bandwidth to the graphics board, and processing limitations on the CPU for determining the exact required segment of the texture.

Typically the texture is broken into pieces, and for each piece the detail level is determined via standard LOD mechanisms. Of course, arbitrary projection ruins this concept, so once again we must find another way to determine the detail necessary. Before going into these details, we first discuss how a simple, small texture is rendered within the application.

Texture Application

Our first approach to applying textures to the surface geometry was a single-pass method. The vertex shader was given information regarding the location of the decal texture, and generated texture coordinates for it based on the vertex latitude and longitude. The fragment shader looked up the texture color and combined it with the base colormap. We set the alpha value of the edge texels in the decal to zero (completely transparent), and blended using this alpha value. The end result was that the decal texture replaces the color map in its location.

Unfortunately, the latitude and longitude after projection are difficult to calculate with significant precision, and the precision required in this case can be very high! A testing texture used in this project is a roughly 9000x8000 image of the UCSD campus, with details down to roughly one or two feet. The precision requirements on the latitude and longitude values are roughly 4.5×10^{-6} values over 360 degrees, implying the need for 26 bits of accuracy in the final result. Not only would a 16 bit lookup texture be insufficient, but the resolution of this texture would be tremendous. Furthermore, a single pass method would limit the number of textures to the multitexturing limit inherent to the GPU. When segmenting large textures into multiple textures comes into play, this method is clearly impossible. Rendering the decal textures in a second pass is required.

For simple textures (ones that are small enough to be a legal texture size), we simply render a quad at the texture's latitude/longitude limits, and use the vertex shader's projections to transform it to the proper place. This has some inherent inaccuracies with floating point math, so we need to do the same trick we used with the spiral rendering. Instead of rendering at the target latitude and longitude, we render offsets from a focal point. In this case, we set the focal point to be the bottom left corner of the texture. Using the offsets added a degree of complication, but the resulting accuracy was much improved. This increased accuracy once again allowed us to use low order Taylor expansions in the data projections. Because the offsets are most likely very close to zero, first or second order expansions are accurate enough for our uses.

Non-linearity

As previously mentioned, many useful projections are non-linear in nature. Rendering a simple quad with non-linear transformations causes a problem. If the quad is large, the geometry of the texture will cut through the Earth geometry. In the presence of non-linear transforms, the quad needs to be tessellated into smaller quads. The number of divisions to tessellate into is dependent on the severity of the non-linearity.

One method to determine the degree of required tessellation was to simply increase the tessellation until the texture looked right. Since we don't want to have to interact with the program in such a way, we need to find a way to automate “looking right”. The extension *occlusion_query* [14] can tell us how many pixels were rendered, as mentioned before. If the number of pixels doesn't increase when increasing the tessellation, we've tessellated enough. This method has a number of obvious drawbacks. First, in certain projections, it could fail entirely. If the projection is concave, for example, the texture might get smaller as the tessellation increases. Second, if the distortion caused by the projection is not constant, such as the hyperbolic or dome view projections, the result is only appropriate to the tested location.

However, in many ways, GPUs are now fast enough to not warrant spending a great deal of effort on reducing tessellation. Multiplying the number of vertices by two is not a severe drawback in this case. We use a 10x10 tessellation of each simple quad, which seems to work without problems. Textures that cover larger area on the globe will need appropriately increased tessellation.

Rendering the Texture

Each texture is rendered explicitly as a decal. No lighting or blending with the background is performed. This is in the interest of not disturbing the data. In the region a decal texture is rendered, only the decal texture is visible. This simplifies the work done by both the vertex shader and the fragment shader. The vertex shader no longer needs to generate a normal or texture coordinates. The fragment shader is now only a very simple texture lookup, rather than a complicated set of texture combinations, lookups, and lighting calculations.

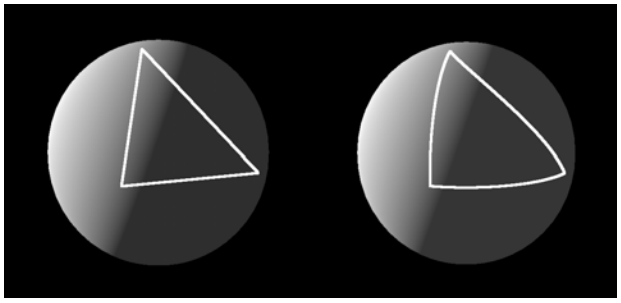

Figure 8: A line drawn on the globe with and without additional tessellation. The lines on the left fail to match the curvature of the surface. If depth-testing was enabled, these lines would pass through the globe and almost entirely disappear.

Large Textures

As previously mentioned, many textures are simply too large to load them entirely into texture memory. The experimental texture used in this project is roughly 178 MB, and is roughly 8000x9000. While large GPUs have this much memory texture sizes of this magnitude are not permitted by OpenGL. Our task was to break the large texture down into simple textures, and render each simple texture as discussed in the previous section.

Regular Grids

Perhaps the simplest approach is to break the image into cells of known size. For each cell, generate a texture in CPU memory based on its segment of the original image. When that cell is not culled, load that texture into GPU memory and render. If all the cells are on screen, however (when the camera is zoomed out, for instance), the entire texture will be loaded, which is not acceptable. Instead of using mipmapping for each texture, we preprocessed each texture detail level into a separate image. Then, depending on how much of the texture is needed, we brought in the appropriate level of detail. If the camera is zoomed out, textures with less detail, and thereby smaller texture, will be loaded, as opposed to the full detail image. Because the entire image could never possibly be needed (due to the limit of framebuffer size), the entire image will never be loaded. Ideally, the number of texels loaded would be as close as possible to the framebuffer size.

This approach has some drawbacks, of course. Every cell needs to be checked, independent of the relative detail of its neighbors. Every cell must render at least a very small texture. From far enough away, every cell will have a single texel texture loaded. Considering for our example texture, if every cell were 256x256 maximum, there would be over 1,000 cells, which is too much. A hierarchical approach is required to solve this problem.

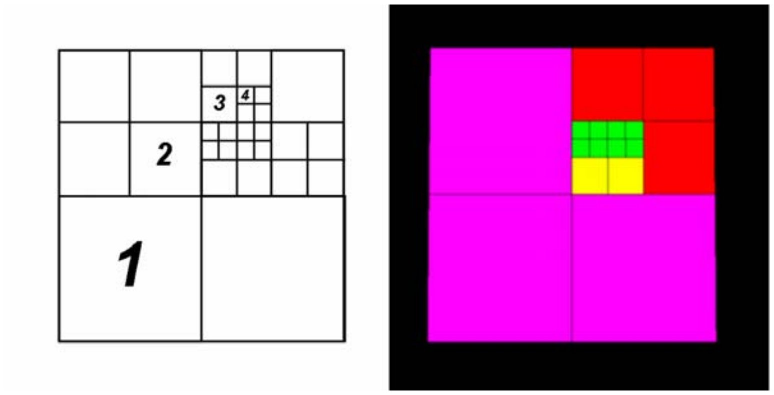

Figure 9: An illustrated and rendered quadtree. The left image shows an example quadtree with the level of selected nodes labeled. The right image is rendered by the project, using different colors for each level.

Quadtrees

Quadtrees provided a hierarchical means to subdivide the texture, solving the problems of the regular grid in the previous section. Every node in the tree, starting with the entire image, is an object responsible for rendering a texture at a constant resolution. Nodes split into multiple nodes when they need to increase their level of detail. Nodes merge back into one when all four children need to reduce their level of detail. This resulted in a local concentration of nodes where the camera is viewing, and very little detail elsewhere.

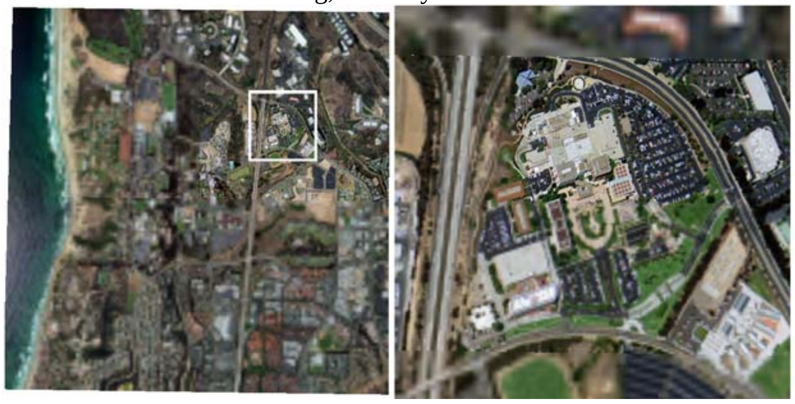

Figure 10: Shows the quad-tree texture at run-time. The camera was positioned very near the surface in the white box. The right image shows the white box region. Note that the texture detail is very high in the center of the white box, where the camera was viewing, but low everywhere else.

Instead of subdividing until the nodes have reached the maximum resolution of the image, we terminated the subdivision early, and allowed leaves to contain multiple levels of detail. When a leaf needs more detail, it pages in the next mipmap level, instead of subdividing. This allows us to use less memory for the quadtree, in exchange for slightly higher texture memory usage. The detail levels of a leaf vary from the detail level of a node down to the maximum detail of the image for the segment the leaf represents.

Selecting LOD for each Chunk

The previous sections have left the topic of level of detail alone, just mentioning that nodes will be split of merged based on a LOD calculation. A simple means to determine when a node has either too much or too little detail is to compare the number of pixels rendered to the number of texels in the texture. Ideally they will be identical, or perhaps there will be slightly more texels. Of course, calculating the number of pixels is impossible without knowing the projection. However, we can use the *occlusion_query* [14] extension to determine exactly the number of pixels rendered. With this information in hand, we can determine when to merge or split nodes.

Occlusion_query allows the programmer to specify a start point and an end point for a query. OpenGL will count the number of samples get rendered to the framebuffer between the two points. The extension provides a set of functions for retrieving data from a query. Since this call involves acquiring data from the GPU, however, care must be taken when using it. Any time data needs to be readback from the GPU, all graphics operations need to complete and the pipeline needs to flush. This can potentially remove any possible parallelism and greatly slow down your rendering. The simple solution is to delay reading the occlusion results until the pipeline has already flushed anyway. For example, reading the queries just before rendering the next frame or during the update processing loop of the system will be substantially faster than reading the queries immediately after issuing the draw commands.

There are some significant drawbacks to using *occlusion_query* as the sole means of level of detail. First, textures that occupy the edge of the screen will render fewer pixels than their neighbors, due to screen culling. Textures at the edge will reduce their detail, resulting in less texture depth at the edges of the screen. This is not a serious flaw when viewing the texture from above, but if the camera is positioned nearly on top of a texture, this can become troublesome. The texture the camera is closest to may be lower detail than others because of clipping, and yet should probably be at the highest detail level available. Secondly, even if the number of pixels rendered is zero, the texture cannot completely cull itself. If it does so, the result can never grow larger than zero, and so the texture will never reappear. Every node must render something when using this technique.

There are significant benefits to using *occlusion_query*. First, the number of texels is kept quite close to the framebuffer size. The calculations are not prone to error. *occlusion_query* is not a terrible performance penalty, as long as the results are not required instantly. We defer the results gathering until the entire render cycle is complete. Lastly, this technique will automatically scale with the resolution. If the target machine is running at 200x200, very few texels will be paged in. If the target is running a 32 MB framebuffer, just as many texels will be paged in to fill the screen with detail.

Other Uses for *occlusion_query* for LOD

In the spiral, the minimum and maximum radii are controlled independently. To determine the minimum radius to render, we obtain the number of pixels rendered by the central triangle fan. This fan fills the center gap in the spiral, and grows larger when the minimum radius increases. We simply set a target pixel value for this triangle fan, and if more pixels are rendered, we reduce the minimum radius by some constant amount. Likewise, if the pixels are too low, we increase the radius equally. The amount of change can be controlled by the pixel difference, but special care would need to be taken here to avoid chaotic situations. If the result jumped past the target radius, noticeable popping could occur. If it jumps so far as to be farther away (in terms of pixels rendered) than the original value, the radius' value will exponentially explode. To avoid these situations, we play it safe and use a small value to control the minimum value.

The maximum radius is harder to control. Ideally we always want to be rendering to the complete screen. However, *occlusion_query* doesn't return any information regarding off-screen pixels, as they are clipped. This means that we have no information regarding how many extra pixels are rendered. If the camera zooms in, we have no clue when to reduce the maximum radius, as the total number of pixels has not changed! We will need to be actively attempting to reduce the radius until we know we've gone too far, and the number of pixels rendered drops. Of course, if the number of pixels drops, the user will see it! This causes a flicker in the corners of the image, which may or may not be acceptable. When the camera quickly pans out, the number of pixels will drop, and so we increase the radius to match. It is difficult to determine how far to jump, however. This symptom is similar to the case of the minimum radius. In this case, however, choosing a slow response rate causes problems in itself. If the user zooms out quickly, the spiral will not increase its maximum radius fast enough, and the user will see the spiral falling behind.

Using *occlusion_query* for Heightfield Terrains

Because there is no displacement mapping in this situation, regions of the texture never occlude other regions of the texture. This technique is applicable to rendering with displacement mapping, with internal occlusion, as well.

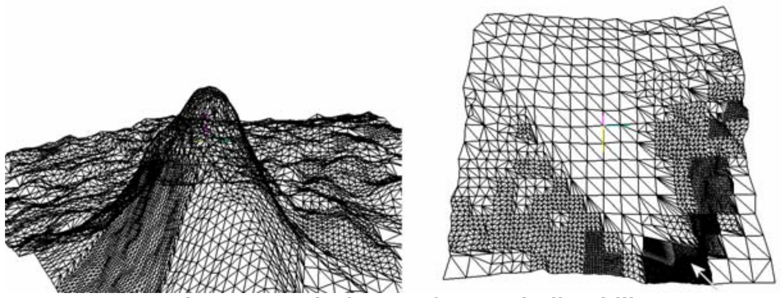

Figure 11: Wireframe of an occluding hill

Figure 11 shows a wireframe of a large occluding hill near the camera. Using pixels rendered as the LOD mechanism will automatically handle large occluders nicely. The right image shows the rendered wireframe from the left viewpoint. Note the large area of low LOD behind the hill, because this area is not visible in the original image. We used this technique on a simple height-field rendering, splitting the terrain into a regular grid, and using an occlusion query per grid area. The final pixels rendered per zone gives a very good indication of how much geometry should be rendered within it. In order for the occlusion query to give us the final pixels rendered, however, we will need to ensure that any pixels that get covered up later do not count. This can be implemented by sorting the zones front to back.

With the LOD controlling the rendered elevation of the heightfield, an interesting form of feedback can occur. If there are sharp changes in the elevation in a zone, decreasing the LOD could remove such a sharp spike, which could increase the number of pixels rendered, which then causes the LOD to increase for the next frame. We already have controlled the threshold pixel counts to avoid popping, using hysteresis, but this effect can bypass even large buffer thresholds. This results in visually-irritating oscillations. We've considered several approaches to this problem.

The geometry for separate levels of detail are generated using geomipmapping [3]. To reduce LOD levels, every other row and column of the heightfield geometry are dropped, reducing the triangle count by a factor of four. More intelligent schemes that take the geometry into count will reduce this problem significantly. Such algorithms are already documented and well known.

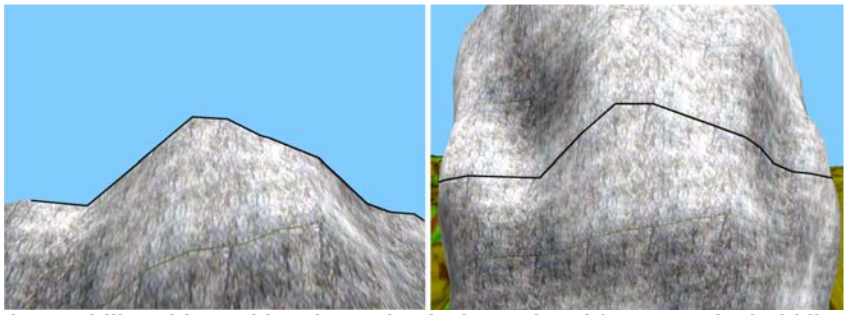
Figure 12: The edge-on hill problem with using only pixels rendered for LOD. The bold line is the edge of a zone. Rotating the camera slightly causes the higher region to spontaneously appear.

The camera's approach to any given zone can also cause problems. Take, for example, a camera walking up a hill. While walking up the hill, if the next zone up the hill was occluded previously, it will not show up until the camera gets very close. This is because the low detail geometry is nearly edge-on to the camera, and so very few pixels get rendered. Once the camera gets closer, the geometry increases, which causes the LOD to quickly rise, and the entire hill to show up. Despite the problems, this algorithm very cleanly and quickly handles occlusion by large terrain features. Geometry behind a large hill will very quickly drop its geometry count. Furthermore, this method could easily be integrated with occluding factors being rendered on top of the geometry. If they are rendered first, the algorithm will automatically take them into account, as the depth test will fail, and the occlusion query will report fewer pixels.

Results

The test machine used to gather these results was a 700 Mhz AMD Athlon CPU, with 384 MB RAM, and an ATI Radeon 9800 Pro running on a 2X AGP bus. The results in this section do not include the use of *vertex_buffer_objects*, due to limitations of the extension at the time of this writing.

This application rendered the entire globe with a single function call. This call will return almost immediately, but any subsequent OpenGL calls will need to wait until the graphics hardware has completed the operation. This leaves a large amount of CPU idle time that is available for other uses. Thus, measuring the rendering time of this algorithm needs to be broken into two parts. The GPU rendering time and the CPU rendering time are both relevant.

Note that algorithms that require multiple calls (such as rendering the texture annotations), despite being not as parallel as single calls, can still take advantage of a great deal of parallelism. In this case, however, it is more difficult to enable the parallelism, due to the need to occasionally return to the rendering task to start the next operation. Timing this explicitly can be very difficult, so typically threading is used in this case.

Table 1 shows the GPU time and CPU time for various framebuffer resolutions and tessellation levels of the globe. Note that the GPU time is almost completely independent of the level of tessellation. This tells us that almost all the time is being spent in the fragment processor, as the number of fragments does not change with the number of triangles. The dependence on the framebuffer size reflects this as well. To demonstrate this further, we wrote a very simple ‘white’ fragment shader, which does as little computation as possible (sets the color to white), to demonstrate the time spent in the complex fragment shader. Note the very high percentage of CPU idle time. This extra time could be used for preprocessing textures, handling user interaction, fetching data from the internet, or many other tasks, without impacting the interaction rate of the application. If the spiral could be moved to GPU memory via *vertex_buffer_objects*, the idle time would rise significantly.

Table 1:

Resolution SpiralSize	GPU (ms)	CPU (ms)	CPU Idle %	GPU time (ms)white	CPU time (ms)white	CPU Idle (white)
1000x1000 17058 tris	13.313	2.326	82.50%	3.046	2.266	25.60%
800x800 17058 tris	9.623	1.967	79.5%	2.312	1.714	25.80%
600x600 17058 tris	6.3	1.753	72.10%	2.097	1.544	26.30%
1000x1000 4294 tris	13.89	1.721	87%	2.056	1.38	32.60%
800x800 4294 tris	9.543	1.41	85%	1.626	1.145	29.56%
600x600 4294 tris	6.07	1.28	78.80%	1.26	0.94	25.55%
1000x1000 2146 tris	13.7	1.557	88.60%	1.853	1.161	37.30%
800x800 2146 tris	9.37	1.23	86.80%	1.523	1.03	31.80%
600x600 2146 tris	5.884	1.07	81.75%	1.226	0.878	28.40%

Rendering performance for the full shaded globe (left columns) and with a 'white' fragment shader (right columns), for various framebuffer and spiral resolutions

Table 2 also shows the time needed to update the texture. Update time is needed to retrieve results of occlusion queries and transfer more, or less, detailed textures to the GPU. The average update time is quite small, but there are occasional spikes when many nodes in the texture quadtree need to be changed at the same time. We can somewhat cap the bandwidth per frame and smooth out these spikes. The first number in the maximum update time column shows the time without this smoothing. The second number shows the update time when only one transfer is allowed per update.

Resolution and path	GPU time (ms)	CPU time (ms)	CPU idle %	Avg update time (ms)	Max update time (ms)
1000x1000 smooth	3.2	2.07	35%	.665	84.1 / 56.0
800x800 smooth	2.56	1.74	31.70%	.45	28.1 / 16.6
600x600 smooth	2.04	1.488	27.30%	.381	19.7 / 15.1
1000x1000 zoom	2.86	1.96	31.20%	.56	76.6 / 29.3
800x800 zoom	2.18	1.588	27.10%	.37	93.5 / 18.4
600x600 zoom	1.686	1.26	24.70%	.28	11.8 / 14.5

Table 2: Rendering and update times and idle percentages for managing an 8192x8192 texture using the quadtree subdivision.

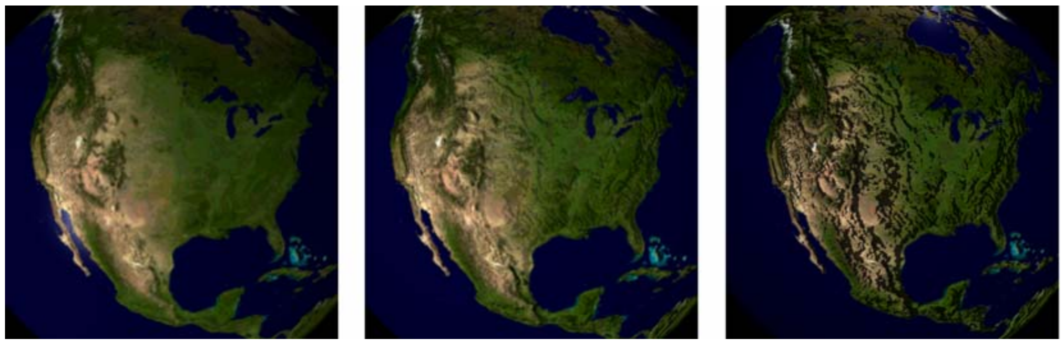

Figure 13: Successively increasing bump-map scaling. The left image (no bumpmapping) looks flat, while the middle image looks good. From this camera viewpoint, bumpmapping is indistinguishable from displacement mapping.

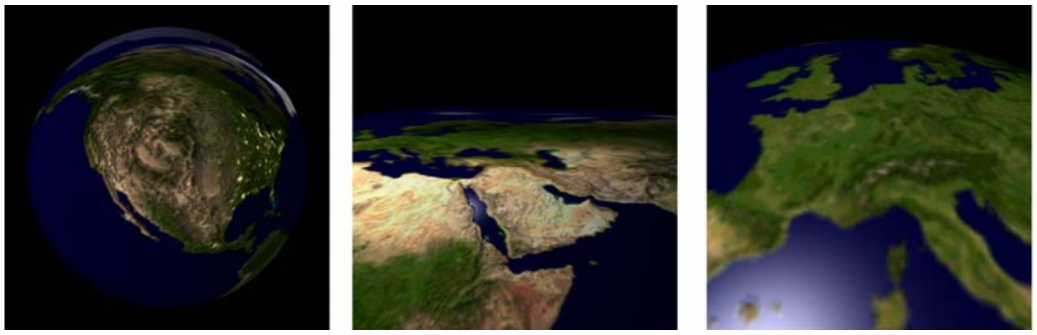

Figure 14: Various images in various resolutions. Clouds were removed from these images so that the effects of bumpmapping are more visible. The image on the left is North America with high bumpmapping in hyperbolic mode. The middle is the Middle East with moderate bumpmapping in flat mode. The right is Europe without any bumpmapping, in dome mode.

Image Quality
Displacement mapping the globe was not an option with the hardware used in this research. Bump-mapping works well for zoomed out images, but is notably poor when up close. Figure 13 shows some bump mapping images with various height exaggeration values. Figure 14 shows some example images from globe, in various projections. Likewise, the texture annotations would be quite poor if the texture detail was not adequately positioned. Figure 15 shows the balance of texture detail in various projections, including the rendered result.

Fragment Shader Optimizations
Table 3 gives an indication of the importance of optimizing the fragment shader. We started with the final version of the globe fragment shader, and began removing various functions. Each one reduces the quality, some more than others. The largest jump comes from removing bumpmapping. This isn't surprising, considering the number of math operations needed to perform bumpmapping. Bumpmapping be sped up a great deal by replacing it with normal mapping. Encoding the normal directly in the texture reduces bumpmapping to the texture lookup, but only works when the surface is known when the normal map is generated. The map no longer specifies bumps, but rather the actual normal of the surface.

Note that the cost of a normalize function is greater than the cost for texture blending. Square roots are particularly expensive, and need to be avoided. This table shows the amount of care that needs to be taken in determining which calculations need to be done per-pixel. If we wished to optimize this program, we would first reduce the cost of bump-mapping by using a normal map. The specular lighting is the next most expensive, so we would reduce the cost by removing the normalize function call and trying to use a lookup texture for the power function.

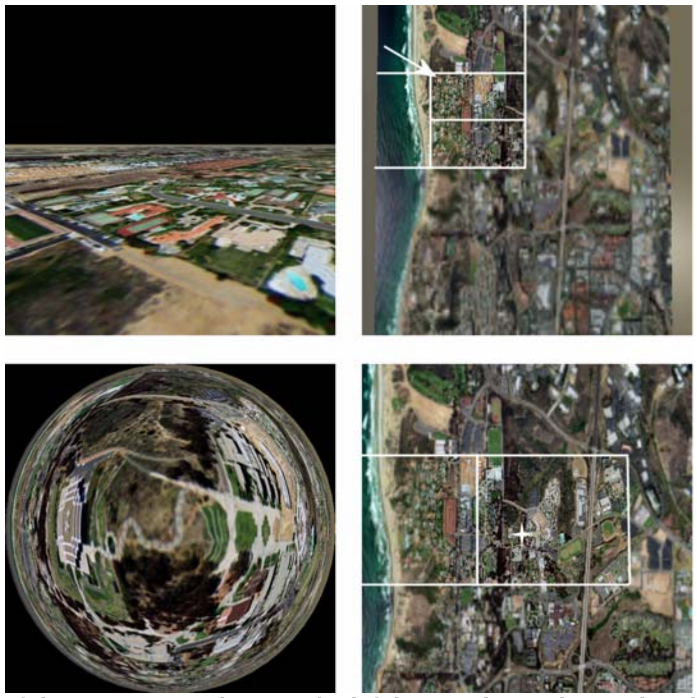
Figure 15: Two images of the texture LOD in use. The left image shows the rendered result. The right image shows the texture balance. The white lines show approximate boundaries between detail levels. The arrow and star show the camera position.

Feature Removed	GPU time (ms)	Code operations removed
None (Complete)	8.326	---
Clouds	8.082	Texture lookup, vector add, vector scale
City Lights	7.820	Texture lookup, clamp, dot, clamp, vector add, vector scale
Tangent normalize	7.457	Normalize
Bump mapping	4.787	Texture lookup, cross, 3 vector scales, 2 vector adds, normalize
Specular lighting	3.535	Dot, normalize, pow, vector add
Diffuse lighting	2.143	Texture lookup, dot, clamp, vector scale, vector add
White	1.974	Normalization of incoming normal.

Table 3: Rendering times for the globe with successive features removed. Each row includes the removal of all the rows above it.

Conclusions

Techniques for rendering annotated, multitextured, multiresolution, interactive terrains efficiently have been demonstrated. The problem of supporting arbitrary projection imposes a number of constraints on the application, especially concerning level-of-detail determinations. With high-resolution imagery, level-of-detail cannot be ignored. Arbitrary projection is supported by using programmable graphics hardware, through the use of the OpenGL Shading Language and OpenGL API. This allows the GPU to perform most of the work, keeping the CPU largely idle for the purposes of preparing annotations and user interaction.

The geometry submitted to the shader is given as an offset from the focus latitude/longitude of the camera. The focus is provided to the shader as well. This gives each shader just enough information to determine the final vertex location. Furthermore, the geometry *never needs to be altered*, which permits the most efficient use of the OpenGL 1.5 feature *vertex_buffer_objects*, which enables storing vertex arrays in graphics memory. An exponentially increasing spiral is used, so that at any level of zoom, the average size of every triangle rendering is roughly constant. The section of the spiral to be rendered is determined based on heuristics or feedback from OpenGL.

Annotative textures (such as detailed satellite imagery) are supported by passing known latitude/longitude coordinates through the programmable pipeline. Level-of-detail on large textures is supported by breaking the texture into a tree of smaller textures, similar to mipmapping. Each quad is rendered, and the number of pixels covered is obtained through use of *occlusion_query*. This feedback tells us how much detail the texture segment should have, and provides a mechanism to determine the level-of-detail. Fast texture paging is handled by preprocessing the large source texture into segments, and loading these segments into GPU memory via memory mapping, avoiding an unnecessary copy into main memory.

The end result is a fast rendering terrain with annotations, running mostly on the GPU, leaving the CPU *85% idle*. We believe that the trend of moving work to the GPU will continue for some time. The spiral level of detail technique might also be applicable to other forms of geometry. The spiral is effectively "wrapped" around the globe or flat data projection, but could be easily wrapped around a loaded model or procedural formula. The difficulty is in providing the GPU the necessary data in such a way that it has access to the required data at the right time. Using such constant geometry can greatly improve performance by using retained mode vertex submission, rather than immediate calls to OpenGL.

The future work in using GPUs for terrain rendering has some fairly obvious direction. Elevation or displacement mapping is required for this technique to really work for traditional heightfield rendering. If it is possible to do this efficiently, such techniques may not be limited to heightfields. If all three coordinates of the vertex could be encoded in a texture, instead of just the height, an arbitrary model could be rendered given parametric coordinates. Normals and texture coordinates could be generated easily from the image. Rendering models this way could save dramatically on graphics memory, and level of detail could become very cheap. This basic approach limits the mesh-dependant level of detail techniques possible, but more complex mechanisms may be invented to merge mesh-dependant level of detail with parametric GPU techniques. This moves even more work off the CPU, and makes better use of the power available on the GPU.

Acknowledgements

This work was supported by the California Institute for Telecommunications and Information Technology (CalIT^2) and the San Diego Supercomputer Center.

References

[1] Mike Bailey, Morgan Gebbie and Matthew Clothier. Realtime 3D Dome Interaction Using the GPU. Submitted for Publication.

[2] James F. Blinn. Simulation of Wrinkled Surfaces. In *Proceedings of ACM SIGGRAPH*, 1978.

[3] William H. de Boer. Fast Terrain Rendering Using Geometrical MipMapping. Article, 2000. http://www.flipcode.com/articles/article_geomipmaps.shtml

[4] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proceedings of the 8th conference of Visualiztion*, pages 81-88, 1997.

[5] Andrew Glassner. Adaptive Precision in Texture Mapping. In *Proceedings of ACM SIGGRAPH*, pages 297-306, 1986.

[6] Hugues Hoppe. Progressive Meshes. In *Proceedings of ACM SIGGRAPH*, pages 99-108, 1996.

[7] Hugues Hoppe. View-dependent Refinement of Progressive Meshes. In *Proceedings of ACM SIGGRAPH*, pages 189-198, 1997.

[8] John Kessenich, Dave Baldwin and Randi Rost. The OpenGL Shading Language. 2004. http://opengl.org/documentation/oglsl.html

[9] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. In *IEEE Visualization 2001 Proceedings*, pages 363-370, 2001.

[10] Frank Losasso and Hugues Hoppe. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. In *Proceedings of ACM SIGGRAPH*, pages 769-776, 2004.

[11] William R. Mark, Steve Glanville and Kurt Akeley. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 2003.

[12] Microsoft DirectX Software Development Kit. http://www.microsoft.com/windows/directx/default.aspx.

[13] NASA. The Blue Marble. http://earthobservatory.nasa.gov/Newsroom/BlueMarble.

[14] OpenGL Architectural Review Board. *occlusion_query*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt, 2003.
[15] OpenGL Architectural Review Board. *texture_cube_map*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_cube_map.txt, 1999.
[16] OpenGL Architectural Review Board. *vertex_buffer_object*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt, 2003.
[17] Bui Tuong Phong. Illumination for Computer Generated Pictures, In *Communications of the ACM, 18*. 1975.
[18] Reuben H. Fleet Science Center. http://rhfleet.org
[19] Randi Rost. OpenGL Shading Language. Addison-Wesley, 2004.
[20] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 2.0). Available at http://opengl.org, 2004.
[21] C. Tanner, C. Midgal, and M. Jones. The Clipmap: A Virtual Mipmap. In *Proceedings of ACM SIGGRAPH*, pages 151-158, 1998.
[22] Thatcher Ulrich. Rendering Massive Terrains Using Chunked Level of Detail Control. Article, 2002. http://tulrich.com/geekstuff/chunklod.html
[23] Virtual Terrain Project. http://vterrain.org/

Appendix A: Fragment Shader Code

```
varying vec3        objtolight;
varying vec3        objtoeye;
varying vec3        normal;
uniform samplerCube basetex;
uniform samplerCube bumptex;
uniform samplerCube cloudtex;
uniform samplerCube nighttex;

void main(void)
{
       vec3 nnormal = normalize(normal.xyz);

       vec4 basecolor = textureCube(basetex, gl_TexCoord[0].stp);
       vec4 bumpcolor = textureCube(bumptex, gl_TexCoord[0].stp);
       vec4 cloudcolor = textureCube(cloudtex, gl_TexCoord[2].stp);
       vec4 nightcolor = textureCube(nighttex, gl_TexCoord[0].stp);

       vec3 tangent = normalize(gl_TexCoord[1].xyz);
       vec3 binormal = cross(nnormal, tangent);

       vec3 bumpnormal = (bumpcolor.r*2.0-1.0) * nnormal
                    + (bumpcolor.g*2.0-1.0) * tangent
                    + (bumpcolor.b*2.0-1.0) * binormal;

       bumpnormal = normalize(bumpnormal);

       float diffdot = clamp(dot(bumpnormal, objtolight), 0.,1.);
       float specdot = dot(bumpnormal,normalize(objtolight+objtoeye));

       float clouddiffdot = clamp(dot(nnormal, objtolight), 0., 1.);

       vec3 outc = basecolor.rgb * diffdot;

       float speccol = pow(specdot, 200.) * texcolor.a;
       outc += vec3(speccol, speccol, speccol);

       outc += cloudcolor.rgb * clouddiffdot;
       clouddiffdot = clamp(clouddiffdot*10., 0., 1.);
       outc += nightcolor.rgb * (1.-clouddiffdot);

       gl_FragColor = vec4(outc, 1.0);
}
```