Clipmapping on the GPU

Roger Crawfis, *Member, IEEE Computer Society*, Eric Noble, Michael Ford, Frederic Kuck, and Eric Wagner

Abstract— Dealing with high-resolution imagery with billions or trillions of samples is an enormous challenge that often overwhelms the graphics subsystem of any computer. Silicon Graphics, Inc. addressed this issue by providing explicit hardware support for offset registers and texture sub-loads in their InfiniteReality™ machine. The clipmap algorithm uses sub-textures and incremental updates based on a toroidal mapping to allow a smaller region to have high-resolution data, while surrounding regions have gradually decreasing resolution. To date, this capability is not supported on most graphics cards. This paper examines several strategies to support clipmapping on commodity GPUs using programmable shaders. We analyze issues associated with clipmaps and provide a taxonomy for clipmap rendering. We have implemented several shaders in our taxonomy and provide performance analysis on current GPUs. Proposals for more efficient hardware to support clipmapping are also presented. Throughout the paper, we provide several extensions to the basic clipmap algorithm, including support for compressed textures, better load balancing, and better support for missing data. Results are presented using our real-time flight simulator applied to large-scale databases.

Index Terms—Terrain visualization, texture mapping, clipmapping, clipmap, mipmap.

1 Introduction

About three years ago, we started examining different solutions for achieving clipmapping using programmable GPUs for our real-time simulators. As the performance of the GPU exponentially increased, our applications became increasingly CPU-bound, due to complex texture management and the severe restrictions imposed by conforming the geometry to texture boundaries. Exacerbating the situation was the customer demand for higher-quality and higher-resolution imagery. Clipmaps allow for a transparent integration of high-resolution textures into an application. Clipmaps [Tanner98] require explicit hardware support for offset registers and texture sub-loads [Montrym97]. To date, replicating this capability on the GPU has not been possible. With programmable shaders and efficient software-based texture updates, we examine several possible solutions to emulate clipmaps on the GPU. Our overall system for real-time visual simulation, described in [Crawfis06], allows for efficient and asynchronous determination of geometry. We combine this with an asynchronous system for controlling and updating the texture data used in our clipmap implementations. This paper focuses on the rendering or shader support needed for clipmapping and the requirements a particular shader imposes on the texture update cycle. Our final system provides the following benefits:

- **Removal of the texture size limit**. By providing clipmaps on consumer GPUs, functionality previously unavailable on a single chip GPU is now possible.
- **The absence of visual artifacts.** Texture seams and blending artifacts prevalent in many texture management schemes are avoided by using a (logical) single texture.
- **Non-square textures:** The overall high-resolution texture can be any size. Power-of-two and square textures are not required.
- **Sparse Textures:** We do not require the entire texture to be present. Any missing data is handled seamlessly. This can lead to a substantial reduction in the database size for large areas having only a few pockets of high-resolution data, such as city data.
- **Low memory and texture usage:** For 2Kx2K texture resolutions, each clip-level requires just 2MB of video memory using DXT1 compressed textures.
- **Asynchronous and amortized clipmap updates**: Since each clip-level is independent, allowing asynchronous updates. We can update a level every n^{th} frame, providing a complete update of all levels amortized over a user-specified duration.
- **Disjoint database development of the texture and geometry models.** The expensive operation of pre-processing a database can be done independently of the texture database construction. Changing the image database does not require recalculating the geometry LOD information.
- **Minimal state changes.** A single shader and set of textures can be used for the entire terrain or all geometric patches for the terrain. A single set of state changes is needed at the beginning of the frame, preventing GPU stall.

2 Related Work

Typical high-resolution texturing systems use a software-based solution comprised of high-resolution textures split into tiles or a quad-tree arrangement [McReynolds05; Hoppe98; Cline98] that can be paged into video memory. This requires either subdividing the geometry at each tile boundary or rearranging the rendering pipeline to loop through each tile, masking out the pixels that do not project onto the texture. The process of clipping or re-tessellating dynamic geometry to match each texture tile introduces additional complexities, as well as run-time performance issues. For terrain rendering, dynamic geometry is generated when switching between levels-of-detail in the underlying elevation data. Hesina et al. [Hesina05] also split the geometry to align with the textures. They use a 2-level texture cache to maintain the video and system management of the textures. We have adopted a similar scheme in our overall system, but do not require the geometry to be split according to texture boundaries.

- *Roger A. Crawfis is with The Ohio State University and DSCI, Inc., E-Mail: crawfis@cse.ohio-state.edu.*
- *Eric Noble, Michael Ford, Frederic Kuck and Eric Wagner are with DSCI, Inc. E-Mail: {enoble|mford|ckuck|ewagner}@dsci.com.*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 2 November 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

Continuous LOD schemes determine the triangulation of the elevation at run-time based on the current viewer position and a desired sub-pixel error [Hwa04; Losasso04; Lindstrom02]. In general, a tight coupling between the texture and the geometric description or tessellation should be avoided, since it either restricts or complicates the tessellation and adds new visual quality issues that the application has to address. Where there is a large disparity between the color or texture sampling rate and the geometric sampling, clipping to artificial texture boundaries severely limits the efficiency of the resulting geometric batching.

Our focus in this paper is applying clipmaps at a per-fragment level. Previous work determined the projection of each triangle, calculating the number of pixels from a triangle's footprint in screen space [Ephanov00]. This was done on the CPU and resulted in poor performance, but aided in migrating users from legacy SGI systems and clipmap databases. The recent advent of geometry shaders would allow for this calculation on the GPU, making this approach much more attractive. For large triangles, an improper texture LOD would still be visible at oblique angles. The further vertex would exhibit aliasing, while the closest vertex would be blurred. Furthermore, if the triangle projected outside of the texture, it was assigned a lower resolution texture. Recently, shader programs have been used to allow a triangle to span up to four textures, alleviating the need to clip the triangles by the tiled texture boundaries [Ephanov06]. A triangle is still assumed to project within a single clip-level.

Geometry clipmaps [Losasso04] render regular nested grids that are viewpoint centered. Each grid is rendered independently and blended with its parent. The texture maps are alpha blended to avoid seams. This was improved by pushing more of the computations onto the GPU by [Asirvatham04]. Clasen and Hege [Clasen06] extended this to planetary size data and spherical domains. Recently, they have examined the update strategy for geometry-clipmaps [Clasen07]. It should be noted, that Geometry Clipmaps are not the same as Tanner's clipmaps. The differences are more than just geometry and imagery. Clipmaps allow for higher-resolution imagery than supported by the hardware for a single geometry patch. Geometry clipmaps are still restricted to the texture size limit of the hardware for both the displacement texture and the imagery texture. Rather than sending eometry batches down from a quad-tree, it sends view-dependent geometry batches based on the geometry clip-level. Our focus in this paper is to support the clipmaps of Tanner for imagery. This is needed to support non-geometry clipmap terrain engines. As will be shown these provide a much greater throughput than those of [Clasen07], as pointed out by Clasen.

Adaptive 4-8 Texture Hierarchies [Hwa04] keep twice as much texture information in an effort to always provide a one-to-one pixel to texel ratio. The geometry and texture are both aligned with a diamond grid, requiring the textures to be rotated and resampled. Döllner et al. [Döllner 2000] use a multi-resolution technique specifically designed for terrain rendering. Wahl et al. [Wahl 2004] accelerate terrain rendering by combining geometric simplification, texture level-of-detail and texture compression with occlusion culling and imposters. Brodersen [Brodersen05] splits the textures rather than the geometry in his geometry mipmap (GeoMipMap) solution [DeBoer00]. The splits contain a grid of GeoMipMaps, which are traversed based on their texture during runtime. MP-Grids [Hüttner98] use a lattice of mipmaps that are updated each frame based on the current view. Hua et al. [Hua04] use a quadtree with a LRU-based cache algorithm to manage their textures.

Other researchers have examined the management of texture atlases for general scenes. Buchhotz and Döllner [Buchhotz05] use a quadtree subdivision containing a hierarchy of a texture atlas for each node. The GoLD [Borgeat05] algorithm requires an expensive pre-processing, providing a hierarchical segmentation of the geometry and the texture into patches. These are optimized into static LODs for efficient run-time execution.

Extent Clipping	Level Determination	
	Implicit	**Explicit**
None	**A**	**B**
Explicit	**Not explored**	**C**
Indexed	**Not explored**	**D**
Implicit	**E**	**Not explored**

Table 1. We have studied five approaches to clipmapping on the GPU as indicated by A, B, C, D, and E. Each of these requires differing sets of resources and exhibit different execution times, as will be analyzed in later sections.

3 CLIPMAP CALCULATIONS

The toroidal texture is ideal for incremental updates to a small roaming sub-image of a much larger image [Losasso04]. Without the toroidal mapping, the entire texture would need to be replaced each time it was updated. For large textures (2048x2048), transferring this amount of data across the PCI-Express bus is too expensive to accomplish within one frame for a real-time application. Double buffering could be used to amortize the updates over several frames, reducing the visible stall. We use a toroidal texture based on small sub-blocks of texels or tiles, restricting updates until an efficient buffer size is reached. This provides a more effective format for streaming from disk and decompression, similar to [Clasen07]. Our tile size varies from 64^2 texels to 256^2 texels. We use a tile size of 256^2 texels for all results in this paper.

3.1 Clipmap Decisions

Texturing large models, such as terrain, boils down to determining the appropriate fragment color for each sample on the terrain. This process needs to consider two key aspects of the texturing:

1. How fast does the texture vary across the sample?
2. Given a discrete representation of the texture, what value should be returned for samples not lying at these impulse samples?

The first issue needs to be addressed to avoid aliasing. The second issue revolves around the need for proper reconstruction and re-sampling. For the purposes of this paper, we will use the hardware based bi-linear interpolation with anisotropic filtering for this issue and focus solely on the first issue above.

In particular, we will examine the process of clipmapping to solve the anti-aliasing problem. The key problem here is the determination of the proper clip-level (or clip-levels for linear interpolation) to use for the fragment. The *ideal* clip-level would be the level corresponding to the texture magnification level, having a one-to-one mapping between the fragments and the texels. We examine two approaches for ensuring that the ideal level or texture is selected during the rendering pass. Using shader programs, the level can be determined using the derivative of the texture coordinates as a function of the screen space. We call this an ***Explicit Level Calculation***. Alternatively, we can construct special mipmaps for each clip-level that provide this automatically in the resulting tri-linear interpolation. We call this approach ***Implicit Level Calculation***. Both of these approaches will be discussed in more detail in Section 3.2.

In a perfect setting, the *ideal* level would be a sufficient condition to support clipmaps. In practice, the clip-textures (or I/O

```
uniform float MaxAnisotropicSampling;
float calculateIdealLevel( void ) {
   vec2 scaleHalf = vec2(0.5,0.5);
   vec2 ddx = scaleHalf *abs(dFdx( gl_TexCoord[0].st ));
   vec2 ddy = scaleHalf *abs(dFdy( gl_TexCoord[0].st ));
   float lddx = ddx.x + ddx.y;
   float lddy = ddy.x + ddy.y;
   float pMax = max(lddx, lddy);
   float pMin = min(lddx, lddy);
   pMin = max(pMin,1.0e-10);
   pMax = max(pMax,1.0e-10);
   float Aspect = min( (pMax/pMin),
                       MaxAnisotropicSampling);
   float fLevel = - log2( pMax/Aspect );
   return fLevel;
}
```

Figure 1. Fragment shader routine to calculate the ideal level in the image pyramid.

system providing the data) are updated asynchronously to the renderer. Using texel information past the edge of a clip-level results in unwanted artifacts, and a mechanism must be employed to restrict an individual clip-level to its extent in world space. Thus, the proper clip-level needs to take into account both anti-aliasing and each clip-level's extent. These are denoted as ***Level Determination*** and ***Extent Clipping*** in Table 1.

We have developed and analyzed three different solutions for restricting or clipping the texels to lie within each clip-level's extent. We also examined the performance in avoiding this step altogether, labeled ***None***. Our first approach is called ***Explicit* Clipping,** in which we use a shader program to perform a bounding box test. Alternatively, we have developed an approach using an indexing structure to aid in the level determination. We call this approach ***Indexed Clipping.*** A fourth approach examines the use of opacity and texture filtering to perform the clipping. This is termed ***Implicit Clipping.***

Combining the anti-aliasing and the clipping provides a total of eight possible techniques to explore. These are summarized in Table 1. We have examined the five most interesting approaches as indicated by the table. Section 3.3 will describe the various clipping approaches in more detail. Section 4 presents an analysis of the shader performance for each approach. In section 5, we examine the software side of the clipmap, in particular, the work required to move the clipmap center. Section 6 presents results within our real-time terrain visualization system. We examine the costs and benefits for each approach in section 7. This analysis points to a final technique for clipmaps based on a wavelet-like decomposition of the database. This approach and its improved performance are discussed in section 8. We conclude the paper with a few suggestions for future hardware that would greatly reduce the resources required for clipmaps.

3.2 Determining the Ideal clip-level

We discuss two approaches to selecting the ideal clip-level. The first is an explicit algorithm using well-known formulas. The second massages the texture samplers to provide the correct level, as a by-product of the tri-linear filtering.

3.2.1 Explicit Level Determination

Mathematically, the texture minification at the current fragment can be approximated by the derivatives of the texel sampling with respect to screen space in both the x and y directions [Ewins98]. We treat level zero[1] as the coarsest mipmap level (1 texel) and compute the magnification of this texel onto the screen space. This is accomplished by taking the derivative of the texture coordinates u and v. It is assumed that these texture coordinates span the entire database. We can compute the desired texture unit as:

$$\lambda = -\log_2\left(\max\left(\left|\frac{du}{dx}\right|,\left|\frac{du}{dy}\right|,\left|\frac{dv}{dx}\right|,\left|\frac{dv}{dy}\right|\right)\right)$$

This level may not be integral, allowing for tri-linear interpolation between levels. It may also exceed the maximum level defined, resulting in texture magnification. We let the shaders that utilize this calculation determine the best action for magnification [Hadwiger 03]. Normally, a simple clamping is applied, but we have also experimented with extrapolation to provide greater contrast enhancement [McReynolds04]. The above calculation finds the maximum squishing of the texture in either the x or y axis of screen space for both of the texture coordinates. This provides a safe solution to avoid aliasing, but results in overly blurred texture filtering when a textured object is viewed obliquely. Anisotropic filtering or multi-sampling was added to the GPU to address this problem. The actual multi-sampling and filtering is done within the hardware texture-sampling unit and is implementation dependent. However, when selecting the texture level to use, we need to account for it in our level determination. To support this, we simply pass in the desired level of multi-sampling, *MaxSampling*, and update our calculations for the desired level based on the formula:

$$\rho_{\max} = \max\left(\frac{1}{2}\left(\left|\frac{du}{dx}\right|+\left|\frac{dv}{dx}\right|\right),\frac{1}{2}\left(\left|\frac{du}{dy}\right|+\left|\frac{dv}{dy}\right|\right)\right)$$

$$\rho_{\min} = \min\left(\frac{1}{2}\left(\left|\frac{du}{dx}\right|+\left|\frac{dv}{dx}\right|\right),\frac{1}{2}\left(\left|\frac{du}{dy}\right|+\left|\frac{dv}{dy}\right|\right)\right)$$

$$Aspect = \min\left(\frac{\rho_{\max}}{\rho_{\min}}, MaxSampling\right)$$

$$\lambda = -\log_2\left(\frac{\rho_{\max}}{Aspect}\right)$$

Here, we have used a simplification to determine the distortion in x and y, to avoid an expensive square-root (See [Segal04] for details). A straightforward implementation in GLSL is provided in Figure 1. This is our ***Explicit Level*** determination algorithm. Wu [Wu98] presents an alternative approach towards calculating the mipmap level using bit-operators to avoid the logarithmic calculation.

3.2.2 Implicit Level Determination

Alpha testing allows for per-fragment clipping. A user-specified threshold is tested against the fragment's alpha value and either accepted or rejected from further processing. Blending provides a rich set of operators for combining two colors with associated weights (typically alpha values). Mipmaps are not needed for the clip-level textures, since by definition the next coarsest level is either the overall mipmap of the next lower-resolution clip-level (i.e. a separate texture). We can use this extra dimension and set-up each clip-texture with a mipmap structure filled with a solid color of black and zero opacity. Now, when the texture fetch is performed, we will get a completely transparent (and black) color, the clip-texture color, or a mix of the two. We set the opacity of the clip-texture to one and use the formula below for calculating the clipmap color:

$$color = \sum_{i=0}^{N} c_i + (1-\alpha_i)color$$

where N is the number of texture units (i.e., number of clip levels plus one), c_i is the color sampled from clip-level i's texture, and α_i

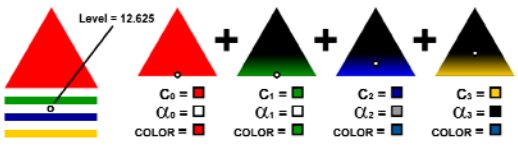

Figure 2. Diagram for implicit level determination

[1] *As opposed to OpenGL, which uses zero for the highest-resolution. Using zero for the lowest resolution allows for an infinite and increasing resolution specification.*

the opacity. This formula is order dependent, starting with the mipmap, and then painting (possibly transparent) colors on top. As an example, consider the following. If the *ideal* level is 12.625 as indicated in Figure 2, with 11 mipmap levels, we first look-up the mipmap texture, resulting in a red color in the diagram. The first clip-level would have texture magnification and select the clip-texture color (green in the diagram, with opacity equal to one). This would then overwrite the current color, resulting in green (color under the green triangle). The second clip level (level 13) would get a mix of black and the clip texture (blue). This is mixed with the current color to provide the correct and final color (turquoise). Since we do not know the proper level, we would continue with the next clip-level. Since this level and all subsequent levels will have texture minification, they will return a black color with zero opacity and not change the final color. Note that no calculation is performed to determine the clip-level. It simply selects the proper color, and hence the proper level, using texture filtering. We call this *Implicit Level* determination. By definition, it requires that all clip-levels be sampled.

Once we have the *ideal* level, we need to determine the desired or proper texture unit for the current projection, by ensuring that the texel for this level is currently loaded. Several approaches for this are discussed in the following section.

3.3 Restricting the Level Based on Extents

Most applications, including Geometric Clipmaps [Losasso04], restrict the geometry to texture boundaries. Geometric clipmaps use a multi-pass solution, drawing a geometry batch restricted to either the texture extent or the ideal level. The seams between levels need to be blended together to avoid artifacts.

3.3.1 Coordinate System Changes

In the most general situation and in our implementations, each clip-level is independently positioned. For tri-linear interpolation, it is desirable for each clip-level to be contained within the region covered by its parent level (i.e., the next coarsest level). The mipmap level contains the entire database. We use two coordinate systems for each clip-level and an additional coordinate system, the world coordinate system, to describe the entire database. The world coordinate system maps the database from geo-specific coordinates to normalized coordinates going from zero to one. Each clip-level can be thought of as a roaming window within the global coordinates. We calculate the roaming window coordinates in the normalized coordinate space to maintain precision. The final coordinate system maps from the roaming window to the toroidal layout of the texture. A common vertex shader is used for all of the shaders in this paper. The user needs only to define the world-space mapping; the clipmapping system automatically generates the mappings for each level. The vertex shaders automatically calculate the texture coordinates for each level using two dot products per level.

3.3.2 No Clipping

Although our application of terrain rendering requires a robust solution, there are applications that may tolerate an inferior image for a short duration, such as web-browsing and image zooming. Other applications may be able to guarantee that the clip-levels are always in synch and centered properly, eliminating the need for clipping altogether. We include the possibility of simply ignoring the clipping extent, both as a baseline for comparison and completeness. For this shader, the fragment color is simply calculated using linear interpolation between the clip-levels indicated from the level determination.

3.3.3 Explicit Clipping

Given texture coordinates in the normalized coordinate system for each clip-level, we need to take into consideration the toroidal mapping within the texture. The texture coordinate, (0,0), may lie in the middle of the roaming window. Each clip-level texture uses a wrapping mode to repeat the texture and allow for the toroidal mapping. The valid roaming region in texture space for a clip-level is (o_x,o_y) to $(1+o_x,1+o_y)$. Thus, we need to pass in the origin vector, (o_x,o_y), for each clip-level into the fragment shader. We use a cascading set of conditionals to first check for the desired level and then test whether the fragment falls within the texture's extent. If it does, the fragment's color is calculated. If not, the desired level is decremented. By ordering the conditionals according to decreasing levels, the proper fragment color is guaranteed. The fragment shader is rather lengthy and is included in the supplemental materials. Only two texture look-ups are performed when the desired level falls between two clip-levels or the mipmap level and the first clip-level. One texture look-up is needed for magnification or if the desired level falls within the mipmap.

```
uniform sampler2D clipLevels[5];
uniform sampler2D levelMap;
uniform float mipLevels;
uniform float bias;
float calcIdealLevel(void);
vec4 calcColor( void ) {
   float fLevel = calcIdealLevel() - mipLevels + bias;
   vec4 c1, c2;
   float maxLevel = texture2D(levelMap,gl_TexCoord[4].st ).a;
   float newLevel = min(maxLevel,fLevel);
   int level1 = max(0,int(newLevel));
   int level0 = max(0,level1-1);
   float w = newLevel - float(level1);
   c1 = texture2D( clipLevels[level1], gl_TexCoord[level1].st );
   c2 = texture2D( clipLevels[level0], gl_TexCoord[level0].st );
   return mix(c1, c2, w);
}
```

Figure 3. Pseudo-code for the index-based clipping.

3.3.4 Indexing Structure

We can envision a map that indicates for each point in the world space, whether the corresponding texel is loaded and available for use. Having a mask for each level would allow us to use a simple lookup to determine whether to restrict a level. For the highest level, this is obviously impractical, as it would require a texture map for the look-up the size of the entire high-resolution imagery. By quantizing the movement of the clip textures, we provide a much coarser grid over the world. This allows for a mask to fit into memory for moderately sized datasets. For clip-levels having closure (that is, the coarsest level contains all of the higher-resolution levels), we can compute a single 2D indexing structure which encodes, for each quantized grid cell, the highest valid clip-level covering this area. This index structure is passed into the shader as another texture. We compute the quantized texture coordinates from the world texture coordinates and look-up the maximum valid clip-level. Taking the minimum of the level and the *ideal* level provides the desired clip level. Pseudo-code[2] for this shader is provided in Figure 3. At most, two texture fetches

```
uniform sampler2D clipLevels[5];
vec4 calcColor( void ) {
   vec4 fragColor = texture2D(clipLevels[0], gl_TexCoord[0].st);
   vec4 newColor  = texture2D(clipLevels[1], gl_TexCoord[1].st);
   fragColor = fragColor*(1-newColor.a) + newColor;
   newColor  = texture2D(clipLevels[2], gl_TexCoord[2].st);
   fragColor = fragColor*(1-newColor.a) + newColor;
   newColor  = texture2D(clipLevels[3], gl_TexCoord[3].st);
   fragColor = fragColor*(1-newColor.a) + newColor;
   newColor  = texture2D(clipLevels[4], gl_TexCoord[4].st);
   fragColor = fragColor*(1-newColor.a) + newColor;
   return fragColor;
}
```

Figure 4. Fragment shader for implicit level determination and implicit clipping (four clip-levels).

[2] *The actual shader and why this is pseudo-code is explained in the discussion section.*

are required for the tri-linear interpolation, and one for the indexing.

An additional benefit to this index structure is its use for missing tiles. The original clipmap algorithm supported databases with differing resolutions. If data did not exist for a single texel, the entire clip-level was invalidated. The granularity for missing data was thus at the entire clip-level, or about 2048 texels across. Using the index structure, we can support a much finer granularity for missing data.

3.3.5 Implicit Clipping

Similar to the *Implicit Level* approach for calculating the ideal level, we examined approaches to perform the extent clipping using opacity. The most straight-forward approach for this is to expand the toroidal map and surround it with a border of texels having zero opacity. This would require updating this border as the clip-texture moved. As the toroidal texture is a repeating texture, it fails once we sample past the border. To address this limitation (repeating), we use a frame-buffer-object (FBO or p-buffer) to unwrap the toroidal texture into another texture. This allows us to use a clamp-to-border wrapping. The double buffering of the textures also allows incremental updates to the toroidal texture. Once the textures are unwrapped, the resulting fragment shader is quite simple, as illustrated in Figure 4. From a shader standpoint, there is no difference between the shaders for approaches A (No Clipping) and E (Implicit Clipping). Rather than testing the resulting opacity value, we sample all of the clip-textures and composite them together. It is also possible to implement this technique using register combiners and OpenGL 1.4.

4 Clip-Shader Performance Comparison

Now that we have a firm grasp of the various approaches for rendering using clipmaps, we will examine the performance of these shaders. Using *n*Vidia's ShaderPerf [Shader07], we analyzed each individual shader. Table 2 provides the number of instructions, number of cycles, number of registers and the theoretical throughput for each shader. Shaders were analyzed assuming four clip-levels on nVidia 6800 Ultra (NV40) architectures. The Explicit Level determination with Explicit Clipping, Shader C, requires the most instructions. The shaders based on implicit level determination, A and E, require only 13 instructions and a minimal set of registers. The register usage is important because the clipmap shader may be a small part of a much larger shader incorporating lighting, bump-mapping and fog. Shader F is discussed in section 8.

We also analyzed five separate static views for each shader. These views are shown in Figure 5, and the rendering times for each shader are reported in Table 3. We show the performance on the nVidia 6800 Ultra and 7800 GTX, as well as the ATI X1900. Each view is rendered at full-screen, using a screen resolution of 1280 by 1024 with 4x FSAA. We used a maximum multi-sampling value of 8x. Two views provide an oblique angle of the San Francisco Bay at different locations. We also captured timings looking straight down from these locations. A separate view is taken from our Nevada database. The number of triangles in each view is also reported. We also provide an image superimposing a pseudo-coloring over one view similar to [Cantlay05].

The San Francisco database is comprised of a 2048 by 2048 mipmap and five clip-levels. The Nevada database has a 2048 by 2048 mipmap and four clip-levels. Several interesting (and surprising) numbers are found in this table (so much so, that we re-checked the numbers three times). First, the fastest frame-rate is reported on the 6800 Ultra for view SF1d. This view looks straight down and has a high texture cache coherency. The shader avoids conditional logic and can be implemented in fixed functionality. Going across the rows, there are differences between the shaders

Table 2. The output from NvPerfShader for each shader, assuming four clip-levels, is shown. The target platform is the NV40 architecture. The Pixel throughput assumes one cycle per texture fetch measured in mega-pixels per second (MP/s). Shader F is our Difference Encoder shader discussed in section 8.

	# instructions	R-Registers	# cycles	Pixel Throughput
A	13	3	8	800.0 MP/s
B	81	6	35	182.6 MP/s
C	137	8	44	145.5 MP/s
D	85	4	36	177.8 MP/s
E	13	3	8	800.0 MP/s
F	13	2	6	1,070.0 MP/s

of 2x to 5x. This indicates that even as the number of triangles increases, the rendering time is fragment bound (mainly due to the large image resolution and 4x full-screen anti-aliasing, FSAA). For the 6800 and the x1900, oblique views are far more expensive than downward looking views. This is still the case with the 7800, but less so. Comparing shaders C and D, we see that for the 6800, they are rather competitive. For the x1900, shader C is clearly faster, while for the 7800 Shader D is faster. Shaders E and F are consistently faster across the architectures.

5 Clip-Level Updating

While the shader performance is an important indicator of the overall clipmap system performance, careful design is required to prevent the software based clip-centering and texture updates from swamping the system. Today's GPUs are extremely powerful and, as will be shown, can often process the data faster than the application can feed it. Different shaders require different update strategies and computations, affecting the overall performance. All approaches require that the individual clip-levels be updated. Issues associated with updating the clipmaps are:

- How often do we check for updates?
- How do we center the clipmaps?
- What is the minimum shift required before performing an update?
- How can we amortize the work across several frames?
- What stages of the update are dependent on each other?

As mentioned, for clipmaps to function properly (in a flat plane) without clipping to the clip-level extents, we need to update the clip-levels as the camera moves. The shaders A and B require a very short update cycle. The other shaders are more forgiving. They will not produce wrong results, but too long of an update

		#tri	A	B	C	D	E	F
6800 Ultra	SF1	10.8K	83	28	28	28	83	45
	SF1d	2.5K	348	71	60	65	348	115
	SF2	27.0K	75	21	19	20	75	35
	SF2d	2.5K	261	68	58	60	261	116
	Nevada	210.0K	80	32	25	31	80	36
x1900	SF1	10.8K	269	244	217	50	269	194
	SF1d	2.5K	330	325	263	292	330	236
	SF2	27.0K	93	85	85	34	93	78
	SF2d	2.5K	250	205	205	200	250	170
	Nevada	210.0K	134	100	100	75	134	100
7900 GTX	SF1	10.8K	178	100	66	100	178	177
	SF1d	2.5K	263	252	147	232	263	320
	SF2	27.0K	166	76	54	77	166	165
	SF2d	2.5K	285	230	247	220	285	285
	Nevada	210.0K	145	99	66	91	145	146

Table 3. Shader performance for five different views. SF1d and SF2d indicate a downward looking view.

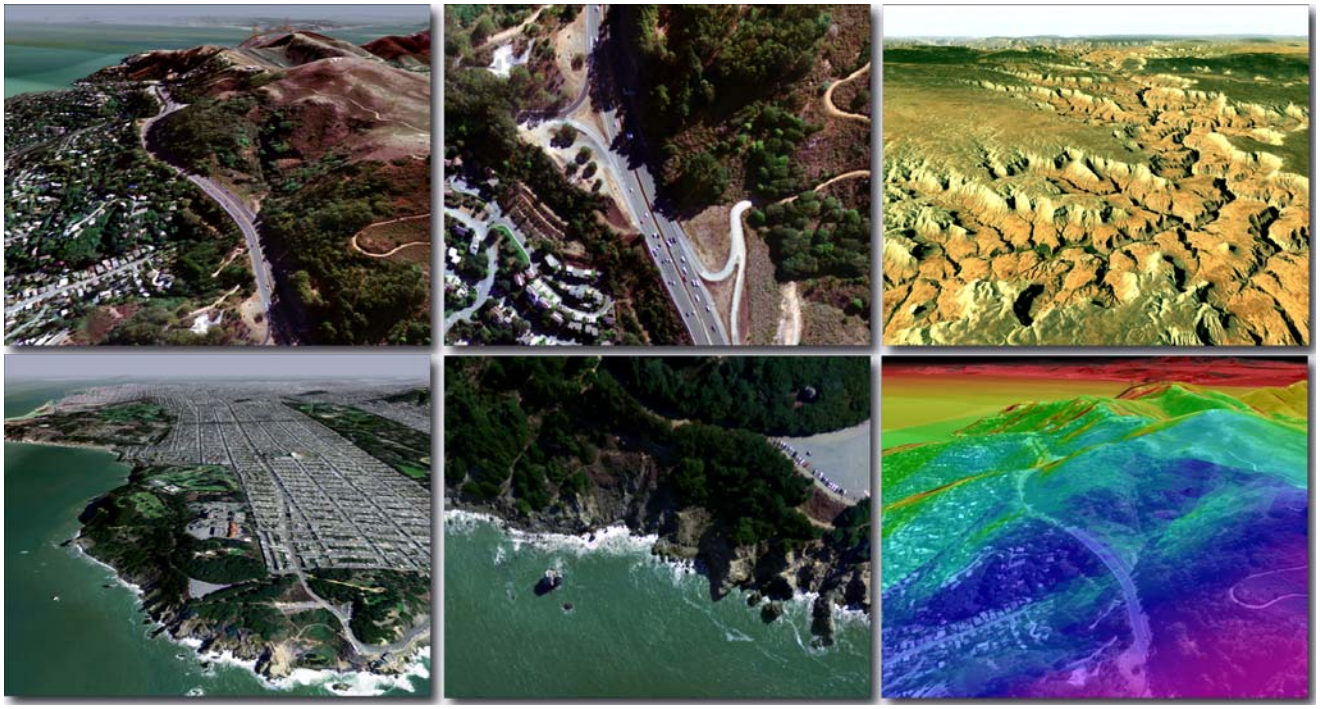

Figure 5. Top-row: two view of San Francisco, on oblique and one looking down, and a view of Nevada. Bottom-row, two more view of San Francisco. The last image has a pseudo-color superimposed over the top-left image to indicate the clip-level the shader selected. There are five clip level plus the mipmap. The mipmap is red to black. The clip-levels are yellow through magenta.

strategy will present noticeable blurring followed by a popping in of the high-resolution data. For all of our tests, we target an average update cycle of two complete updates per second. This has proven to give very smooth and reasonable results. Ultimately, this is dependent on how fast the camera can move. For flight simulators, even with a relatively slow aircraft, rapid camera movements may occur if the user is permitted free rotation of the virtual camera. The clipmap should **not** be centered under the camera, but at a position that covers the viewport. Finding an ideal clipmap center can be quite involved, with no single correct answer. Our centering algorithm for the flight simulator continuously moves the clipmap levels based on the user location, their viewing direction and a measure of the underlying terrain.

For this paper, we will concentrate on the factors influenced by the choice of rendering shader. Our clipmapping system uses a two-stage cache for updating the clip levels. Updating each level sequentially would consist of the following stages:

1. Determine and fetch tiles changed from previous update for level 0.
2. Push changed tiles onto the GPU for level 0.
3. Update any shader variables for level 0.
4. Update the texture coordinate generator or matrix for level 0.
5. Determine and fetch tiles changed from previous update for level 1.
6. …

In addition to these stages, the index-based shader requires two extra stages:

7. Calculate the index grid values (i.e., a level map).
8. Copy the level map to the GPU.

The implicit clipping shader, E, requires that the toroidal-based texture be converted to a non-toroidal format. This adds the following stages for this shader:

9. Unwrap the toroidal texture for level 0.
10. Unwrap the toroidal texture for level 1.
11. …

Any time a texture that is currently used for rendering changes, texture coordinates and uniform variables need to be updated. For shaders A-D, this implies that stages 2-4 operate as an atomic operation. For shader E, stages 3, 4, and 9 must operate atomically. A similar constraint applies to the additional clip-levels. The index-based shader requires a more stringent update. Since the clipping is predicated upon the index structure, whenever any clip-level changes the index structure needs to change as well. There are several solutions for this, but our choice performs stage 1 for all levels, followed by stage 7, and then stages 2-4 (for each clip-level) and stage 8 are performed atomically.

Note, that during typical usage, many of these stages may not actually perform any changes to the system. In particular, the higher-resolution clip-levels will change far more frequently than the coarser clip-levels during a typically fly-through. Each stage requires a vastly different amount of time, and even the same stage's time will vary substantially based on its work load. In general, zero to fifteen tiles will be updated for a single clip-level. The level map requires a complete texture upload. For four clip-levels and a tile size of 256 by 256 texels, the level map occupies a single channel 256 by 256 sized texture, equivalent to a single tile.

While each stage individually does not constitute a burden on the computing resources, for several levels the total cost can tax the overall system. The goals for our system are a sustained 60 fps with no single frame taking more than $1/20^{th}$ of a second. In the next section, we analyze the total update plus rendering cycle of our flight simulator and point out the differences in the update cycles between the shaders.

6 PERFORMANCE

```
uniform sampler2D clipLevels[4];
uniform sampler2D levelMap;
float calcIdealLevel(void);
vec4 calcColor( void )
{
   float fLevel = calcIdealLevel();
   vec4 color, color2;
   float maxLevel = 16.0*texture2D(levelMap,gl_TexCoord[4].st ).a;
   float newLevel = min(floor(maxLevel),ceil(fLevel));
   int level = int(newLevel);
   float w = fLevel - float(level) + 1.0;
   w = min(w,1.0);
   if( level >= 3 && fLevel >= 4.000000 )
   {
      color = texture2D( clipLevels[3], gl_TexCoord[3].st );
      color2 = color;
   }
   else if( level == 3 )
   {
      color = texture2D( clipLevels[2], gl_TexCoord[2].st );
      color2 = texture2D( clipLevels[3], gl_TexCoord[3].st );
   }
   else if( level == 2 )
   {
      color = texture2D( clipLevels[1], gl_TexCoord[1].st );
      color2 = texture2D( clipLevels[2], gl_TexCoord[2].st );
   }
   else if( level == 1 )
   {
      color = texture2D( clipLevels[0], gl_TexCoord[0].st );
      color2 = texture2D( clipLevels[1], gl_TexCoord[1].st );
   }
   else if( level <= 0 )
   {
      color = texture2D( clipLevels[0], gl_TexCoord[0].st );
      color2 = color;
   }
   color = mix(color, color2, w);
   return color;
}
```

Figure 7. Fragment shader routine used in the level-map shader.

Finally, we analyzed the overall performance of shaders C, D, and E using 10,000 frames from a fly-thru over the San Francisco Bay. This database uses DXT1 compressed imagery at 1/3 meter resolution. A complete update cycle occurred twice per second. The accompanying video takes the viewer along this flight and illustrates the performance using our clipmapping system with shaders C, D, and E. The graph in Figure 6 measures the frame-time for each shader. We have zoomed into a typical 50-second segment to better see the results. All of the methods provide adequate performance, averaging from 100 to over 200 frames per second. The level-map, shader D, clearly shows the updating of the textures. Since all texture updates occur within a single frame, its performance oscillates between the rendering time and the update time. The explicit clipping, shader C, only needs to update the clip-textures and the shader uniform variables. This is easily amortized over many frames and the curves are difficult to differentiate. Shader E is by far the fastest shader, but requires a render-to-frame-buffer operation to rasterize the entire 2048 by 2048 clip-texture to perform the unwrapping of the toroidal layout. We measured the cost associated with just this operation and found that it takes 1.5 milliseconds on the 7800 GTX. This operation is performed for each level only when a level changes.

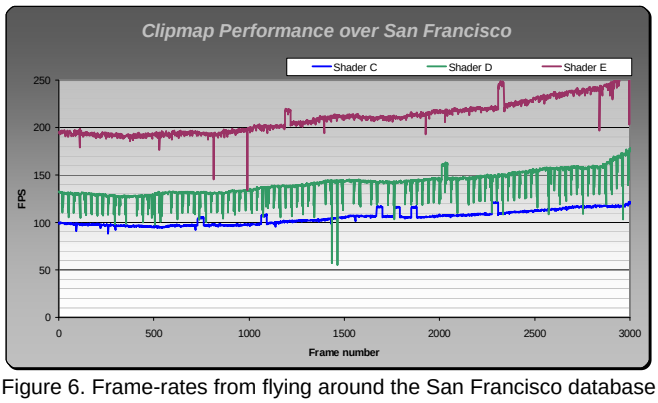

Figure 6. Frame-rates from flying around the San Francisco database for shaders C, D and E with and without updates.

Hence, zero to 2*n* updates may occur per second, where *n* is the number of clip-levels.

7 Discussion

So, having a thorough understanding of the various performance characteristics of each shader, which one is the best? Clearly shader E exhibits the best run-time performance. We can rate each shader based on four characteristics:

- Shader performance
- Maximum single frame-time
- Video memory usage
- Scalability with texture size
- Granularity of missing data supported

The performance measurements for shaders C and D are slower than one would hope or expect. A major reason for this is the lack of support for indexing into the sampler array using a calculated value. While this is legal in the OpenGL GLSL specification, existing hardware or compilers only support constant indexes. This requires us to check every single clip-level and encapsulate the texture fetching inside conditional logic. Figure 7. shows the actual shader used for three clip-levels with shader C. The next generation of hardware provides for a new construct called a Texture Array [MSDN07]. This will allow direct indexing into an array of textures and allow for the shaders depicted earlier. We simulated this for the level map shader, D. With direct indexing, we can reduce the complexity to 30 instructions, 3 registers and 14 cycles for a theoretical throughput of 457 MP/s.

The maximum single frame time is the rendering time plus the maximum clipmap update time. As shown, this is dominated by the update time. Double buffering of the clip-textures would allow for updates (or partial updates) to occur across many frames. This would be particularly helpful for index-based clipping, shader D. This shader requires one additional texture for the level map. This texture only needs to be a single channel, so at most 4MB of additional video memory is required with a level map of size 2048 by 2048. This texture also affects the scalability or maximum virtual clipmap size. With a tile size of 256 and a maximum texture size of 2048, we are limited to a clipmap having at most 512K texels on a side (eight clip-levels). Increasing the tile size and the maximum texture size (DirectX 10 requires a minimum size of 8192) can extend this.

Un-wrapping the toroidal textures for shader E requires an additional texture for each clip-level at the same resolution as the clip-texture. A major drawback to this approach is the lack of compressed FBO formats. As such, our DXT1 compressed textures are expanded into full 32-bit RGBA textures. The mipmap texture can be left in a compressed format. For four clip-levels, this increases the video resources from 10.6MB (mipmap plus 4 clip levels all DXT1 compressed) to 96MB (same as before plus an additional 4 clip-levels uncompressed and mipmapped). While video memory has increased substantially, a near ten-fold increase in usage makes it more difficult to choose this as the optimal shader.

For scalability, shaders C and D's performance does not scale very effectively. This is primarily due to the lack of indexing support noted above. These shaders only require two or three texture look-ups, so they should exhibit better performance once this problem is corrected. The use of texture sets will also allow for a greater number of textures. Currently, only sixteen texture units are allowed. Using eight or ten of these for clipmapping restricts the number of textures that can be used for other routines in the shader (e.g., lighting). Our next set of experiments will examine the DirectX 10 hardware.

The level map shader was designed to allow for partially loaded clip-levels. Clipmaps and shaders B and C must invalidate

the entire clip-level if a tile's information was not pre-fetched properly, or does not exist in the database. This has an impact more on the border of areas scanned at different resolutions. For shaders A and E, we can add in transparent tiles, and allow for missing data at the tile granularity.

8 Difference Encoded Clipmaps

As a final experiment, we decided to extend the implicit clipping with implicit level determination shader, shader E, using compressed textures based on a wavelet-like decomposition. Since every texture is used in this shader, there is no additional penalty for reconstructing the texture from the difference coefficients. This approach also allows us to incorporate the missing tile support of the level map.

The challenge here was incorporating the DXT1 compression into the solution. DXT1 compressed images are one sixth the size of an uncompressed RGB texture. This reduces the bandwidth on the I/O system and the texture updates into the GPU. Our solution is an iterative difference encoding / reconstruction / compression process. We start with an uncompressed version of our database, where the high-resolution imagery has been filtered and down-sampled for each lower-resolution level. Additionally, we break each level into tiles according to the user specified tile size (we found that 256 works well). The lowest levels are coalesced into the mipmap image for the database. We apply a compression algorithm [Brown07] on the mipmap to generate the DXT1 compressed version of the mipmap. We then take the DXT1 version, upsample it by two and subtract it from the uncompressed imagery for the next level. This signed distance is scaled and biased to the range zero to one. We then apply DXT1 compression on this difference image. The remaining steps repeat this process except that the reconstruction combines all of the previous levels by up-sampling the mipmap by the appropriate power of two, inverting the scale and bias on the difference image, up-sampling and adding to the up-sampled mipmap.

We remove any tiles that do not contribute to the final reconstruction (constant grey), providing some additional database compression. These grey tiles are added back in during run-time by the system, as the textures require contiguous data. See [Kraus04] for an alternative towards this. The shader reconstructs the fragment color using the formula:

$$color = mipMap + \sum_{i=0}^{N} (c_i - \tfrac{1}{2})$$

This equation does not rely on the opacity value for the clipping. Instead, we set the border color and lower resolution mipmap levels to a constant grey value of ½ for each clip-level texture. This provides the same affect as shader E, but only requires 24-bit RGB textures for the unwrapping. Fewer DXT1 compression artifacts are visible on magnification, as the difference images are approximated better by the 16-bit color quantization and linear color table used in the compression.

In summary, this shader looks the most promising. It has the small tile-size support for missing data or databases with different levels of resolution. It has an extremely fast and compact fragment shader, and it produces the best quality images. The major drawback is the extra video memory required for unwrapping the textures. In the following section, we propose some simple extensions to existing hardware to improve the performance of the various techniques.

9 Proposed Hardware Extensions

The most pressing extension needed for shaders C and D has already been addressed by the introduction of Texture Sets, as indicated above. For the difference encoded shader and for shaders A, B, and E, a new clamping option is needed. Clamping to border provides the desired behavior if the roaming textures are not toroidally mapped. An extension that would allow the specification of an alternate border location in which the texture coordinate is first clamped to the toroidal border and then wrapped would be ideal. For shader E and the difference encoded shader, this would eliminate the need to unwrap the toroidal textures. Alternatively, a fast texture shift to unwrap the toroidal textures that did not require rendering to a frame buffer object would allow compressed textures in both the wrapped and unwrapped versions, providing a significant savings in video memory.

10 Contributions and Future Work

In this paper we have presented, to our knowledge, the first implementation of clipmapping on the GPU. We illustrate several possible approaches, providing a taxonomy in terms of the two key functions of clipmaps, and we report performance on current hardware. Several efficient and practical implementations are illustrated and shown in real applications. Finally, we offer suggestions for future hardware that will make clipmapping on the GPU even more efficient.

As hardware advances, the possible approaches presented here undoubtedly will change, and different implementations will present themselves. In particular, we are beginning to investigate the remainder of the shaders from Table 1. At first, we did not think these would be interesting choices, but preliminary results seem to indicate that at least one might be a viable alternative. We intend to repeat our experiments on the new DirectX 10 compatible hardware, as well as other architectures such as the Xbox 360. Clipmaps are usually presented as an extension to mipmaps, but this is not a necessary condition. Our current work is exploring the separation of the larger multi-resolution texture space into layers that operate autonomously.

We also plan to investigate similar approaches presented here to address the blending problems with Geometric Clipmaps and other GPU-based terrain meshing algorithms. Variants of the implicit techniques may be useful in the vertex shaders to control the meshing without introducing seams. A final area of investigation entails extending the tile updating to include synthesized texture based Wang Tiles [Cohen03] [Wei04] [Lagae06], or texture splats [Lefebvre05]. We also plan to investigate the use of Adaptive Texture mapping [Kraus04] in this synthesis process.

Acknowledgements

The authors wish to thank Kelley Rice for suggestions on improving the paper, Luke Molnar for assistance in creating the databases, and William Flanigan for help in producing the video segments and figure 2.

References

[Asirvatham05] Asirvatham, A., H. Hoppe. *Terrain Rendering Using GPU-Based Geometry Clipmaps,* in **GPU Gems 2**, Addison-Wesley, 2005, pp. 27-46.

[Blow00] Blow, Jonathon, *Terrain Rendering at High Levels of Detail, in* **Game Developers Conference 2000**, (March 20-24), San Jose, CA, Proceedings, available as: http://number-none.com/blow/papers/terrain_rendering.pdf.

[Borgeat05] Borgeat, L., Godin, G., Blais, F., Massicotte, P., and Lahanier, C. *GoLD: interactive display of huge colored and textured models.* **ACM Transactions on Graphics** 24, 3, **SIGGRAPH 2005**, (Jul. 2005), pp. 869-877.

[Brodersen05] Brodersen, A. *Real-time visualization of large textured terrains.* In Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia, Dunedin,

New Zealand, **GRAPHITE '05**. ACM Press, New York, NY, pp. 439-442.

[Brown07] Brown, S., *Squish – DXT Compression Library*. (http://www.sjbrown.co.uk/?code=squish).

[Buchholz05] Buchholz, H., and J. Döllner. *View-Dependent Rendering of Multiresolution Texture-Atlases.* **IEEE Visualization 2005**, (Minneapolis, MN), pp. 215-222.

[Cantlay05] Cantlay, I. 2005. Mipmap-level measurement. GPU Gems II, 437-449.

[Cignoni03] Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. *Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM).* **IEEE Visualization 2003.** IEEE Computer Society, Washington, DC, pp. 147-154.

[Clasen06] Clasen, M. H.-C. Hege: *Terrain Rendering using Spherical Clipmaps*, EuroVis 2006 – Proc. Eurographics / IEEE VGTC Symposium on Visualization, pp. 91-98, 2006.

[Clasen07] Clasen, M., H.-C. Hege, *Clipmap-based Terrain Data Synthesis*, Simulation und Visualisierung 2007, T. Schulze, B. Preim, H. Schumann (eds.), SDS Publishing House, San Diego, 2007, pp. 385-398

[Cline98] Cline, D. and Egbert, P. K. *Interactive display of very large textures.* In **Proceedings of the Conference on Visualization '98** (October 1998). IEEE Computer Society Press, Los Alamitos, CA, pp. 343-350.

[Cohen03] Cohen, M. F., Shade, J., Hiller, S., and Deussen, O. *Wang Tiles for image and texture generation.* **ACM Transactions on Graphics** 22, 3 (Jul. 2003), pp. 287-294.

[Crawfis06] Crawfis, R., F. Kuck, E. Noble, E. Wagner, *Asynchronous Continuous Level-Of-Detail and Texture-Mapping for Large-Scale Terrain Rendering Systems*, Proceedings IMAGE 2006.

[DeBoer00] De Boer, W. H., 2000. *Fast terrain rendering using geometrical mipmapping.* Available on-line at: (http://www.flipcode.com/articles/article_geomipmaps.pdf).

[Döllner00] Döllner, J., Baumann, K., and Hinrichs, K. Texturing techniques for terrain visualization. In *Proc. of the 11th Ann. IEEE Visualization Conference (Vis) 2000*, pp. 227--234.

[Duchaineau97] Duchaineau, M. A., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. 1997. *ROAMing terrain: Real-time optimally adapting meshes.* In **IEEE Visualization '97**, IEEE, pp. 81-88.

[Ephanov00] Ephanov, A., System and method for simulating clip texturing. U.S. Patent 6924814 (August 2005).

[Ephanov06] Ephanov, A., C. Coleman, *Virtual Texture: A Large Area Raster Resource for the GPU.* In Proceedings **Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2006**, (December 2006), pp. 645-656.

[Ewins98] Ewins, J. P., Waller, M. D., White, M., and Lister, P. F. *Mipmap Level Selection for Texture Mapping.* **IEEE Transactions on Visualization and Computer Graphics** 4, 4 (Oct. 1998), pp. 317-329.

[Hadwiger03] Hadwiger, M., Thomas Theußl, Helwig Hauser, Eduard Gröller. **Mipmapping With Procedural and Texture-Based Magnification,** SIGGRAPH 2003 Technical Sketch.

[Hesina05] Hesina, G., S. Maierhofer, and R. Tobler, *Texture Management for high-quality City Walk-throughs.* **9th International Symposium on Planning and IT**, (2004) ISBN 3-901673-11-2, pp. 305-308.

[Hoppe98] Hoppe, H. 1998. *Smooth view-dependent level-of-detail control and its application to terrain rendering*. In **VIS '98: Proceedings of the conference on Visualization '98**, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 35-42.

[Hua04] Hua, W., Zhang, H., Lu, Y., Bao, H., and Peng, Q. 2004. Huge texture mapping for real-time visualization of large-scale terrain. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, New York, NY, USA, pp. 154-157.

[Hüttner98] Hüttner, Tobias, *High Resolution Textures*, Late Breaking Hot Topics, IEEE Visualization 98 CD-ROM Proc., IEEE Computer Society, Los Alamitos, Calif., Oct. 1998.

[Hwa04] Hwa, L. M., Duchaineau, M. A., and Joy, K. I. 2004. *Adaptive 4-8 Texture Hierarchies.* In **Proceedings of the Conference on Visualization '04** (October 2004). IEEE Visualization. IEEE Computer Society, Washington, DC, pp. 219-226.

[Kraus02] Kraus, M. and Ertl, T. 2002. Adaptive texture maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrucken, Germany, September 01 - 02, 2002).

[Lagae06] Lagae, A. and Dutré, P. 2006. *An alternative for Wang tiles: colored edges versus colored corners.* **ACM Trans. Graph.** 25, 4 (Oct. 2006), 1442-1459.

[Lefebvre05] Lefebvre, S., Hornus, S., and Neyret, F. Texture sprites: texture elements splatted on surfaces. In *Proceedings of the 2005 Symposium on interactive 3D Graphics and Games* (Washington, District of Columbia, April 03 - 06, 2005). SI3D '05. ACM Press, New York, NY, pp. 163-170.

[Levenberg02] Levenberg, J. *Fast view-dependent level-of-detail rendering using cached geometry*. In **VIS '02: Proceedings of the conference on Visualization '02**, IEEE Computer Society, Washington, DC, USA.

[Lindstrom02] Lindstrom, P., and Pascucci, V. *Terrain simplification simplified: A general framework for view-dependent out-of-core visualization.* **IEEE Transactions on Visualization and Computer Graphics** *8*, 3, pp. 239-254.

[Losasso04] Losasso, F., and Hoppe, H. *Geometry clipmaps: terrain rendering using nested regular grids.* **ACM Transactions on Graphics** 23, 3 (Aug.), pp. 769-776.

[McReynolds05] McReynolds, Tom, and David Blythe **Advanced Graphics Programming Using OpenGL,** Morgan Kaufmann (2005), ISBN 1558606599.

[MSDN07] Microsoft DirectX Developer Center. *http://msdn.microsoft.com/directx*.

[Montrym97] Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. *InfiniteReality: a real-time graphics system.* **International Conference on Computer Graphics and Interactive Techniques** (SIGGRAPH 97). ACM Press/Addison-Wesley Publishing Co., New York, NY, pp. 293-302.

[Segal04] Segal, M, K. Akeley, The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004), Silicon Graphics, Inc.

[Shader07] ShaderPerf (2007), nVidia Developer Tools (http://developer.nvidia.com/object/nvshaderperf_home.html)

[Tanner98] Tanner, C. C., Migdal, C. J., and Jones, M. T. *The clipmap: A virtual mipmap.* In **SIGGRAPH 98 Conference Proceedings**, Addison Wesley, ACM SIGGRAPH, 151-158.

[Wahl04] Wahl, R., M. Massing, P. Degener, M. Guthe, R. Klein, *Scalable compression and rendering of textured terrain data*, in **Journal of WSCG**, volume 12, 2004.

[Wei 04] Wei, L. *Tile-based texture mapping on graphics hardware.* In **Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hardware,** HWWS '04. ACM Press, New York, NY, pp. 55-63.

[Williams83] Williams, L. *Pyramidal parametrics*. In Proceedings of **SIGGRAPH '83**. ACM Press, New York, NY, 1-11.

[Wu98] Wu, K. *Direct Calculation of Mipmap Level for Faster Texture Mapping*, Hewlett-Packard Labs Technical Report, HPL-98-112 (June 1998).