The Clipmap: A Virtual Mipmap

Chris Tanner, Chris Migdal, Michael Jones

Abstract

We describe a new dynamic texture representation, the *clipmap*, which efficiently caches textures of arbitrarily large size in a finite amount of physical memory for rendering at real-time rates. Further, we describe an overall software system for automatic management of this representation that simplifies integration into the most demanding of real-time applications. We show the scale and robustness of the combined hardware/software solution that we have implemented by presenting high-fidelity 60 Hertz examples of the system virtualizing gigabytes of texture. Finally, we suggest ways that rendering systems could exploit the concepts underlying clipmaps to solve similar large scale problems.

A clipmap is an extension of the well-known mipmap texture representation both in structure and usage. However, clipmapping exploits a key aspect inherent in mipmap texture accessing to sharply limit the amount of physical memory required—in typical configurations, a clipmap renders an enormous texture exactly as a mipmap would while using a tiny fraction of the physical memory that a mipmap requires. Further, clipmaps can be dynamically reconfigured to support a trade-off between visual image quality and the time required to update the dynamic representation.

Since clipmaps are dynamically updated on a per-frame basis, the behavior of the overall system in the real-time rendering context is significant. From this perspective, the higher-level clipmap representation behaves like a multilevel look-ahead cache. This caching system employs knowledge of the rendering algorithm, pipelined rendering architecture, and spatial coherence to achieve both robust real-time performance and high-quality rendering using system resources sparingly.

1 Introduction

The problem is a fundamental one. Real-time renderers continue to use higher and higher resolution textures to create more realistic scenes and provide more visual information for users. At the same time, the artificial environments being represented are themselves becoming larger and larger. These systems have reached a level of visual fidelity where it is no longer possible or desirable to improve them with repetitive low resolution texture images. However, the current paradigm for representing and storing texture images does not provide the level of scalability needed to manage textures of the desired magnitude and specificity. For instance, a real problem we address is the representation of the entire earth as a single high resolution texture. Users need to visualize imagery of significant portions of the earth's surface at 1 meter per texel as well as small insets of resolutions up to two centimeters per texel. Representing the earth at 1 meter per texel requires a 40 million by 40 million texture map and an overall mipmap storage of approximately 12 petabytes. By comparison, a texture image representing the southwestern United States at 25 meters per texel and requiring 2.8 GB of mipmap storage is considered small.

Given this context and problem, it is easily possible to define goals for an effective new texture representation. First, the representation must be able to perform effective rendering with a small and finite portion of an otherwise arbitrarily large texture image. Second, it must be able to be efficiently updated in real-time such that updates do not become the fundamental bottleneck of the rendering system. Third, load control must not only be possible but robust and automatic as well. Moreover, fallback under heavy load must provide a high level of visual continuity so as not to be distracting. Fourth, it must be possible for the entire representation to be extended all the way to disk and easily integrated into applications at full performance. Finally, the implementation should cost as little as possible relative to previous texture implementations in terms of complexity, performance, and hardware resources.

We have developed our initial clipmap implementation addresses these goals through a combination of low-level hardware and higher-level system level software. The hardware provides real-time rendering capabilities based on the clipmap representation, while the software provides a complete end-user solution for applications. This paper will describe our clipmap implementation and how well it meets the goals and challenges already outlined: §2 presents other approaches to managing large texture images, §3 describes exactly what a clipmap is and what it achieves, §4 explains how the clipmap is updated using incremental texture downloads and addresses memory bandwidth issues relevant to updating, §5 teaches implementing the clipmap representation as a modification of mipmap rendering, §6 describes how to further generalize clipmaps to be more useful under hardware resource and precision limits, §7 discusses the higher level software used to extend the clipmap representation to manage system load and deal with user data on disk, §8 shows images from representative 60 Hertz examples of the system, and finally, §9 reviews our first implementation and finds that it been successful and leads to promising new research directions.

2 Previous Approaches

Previous real-time solutions for dealing with large textures have had limited success and suffer from inherent limitations. The only common method relies on subdividing the large overall texture image into small tiles so that each tile can be represented as a small texture. This approach provides good paging granularity for the system both from disk to main memory and from main memory to texture memory. It also can be implemented with the basic low-level texture management facilities provided by most hardware renderers. In practice, however, this approach has several very limiting drawbacks. First, geometric primitives must not straddle texture-tile boundaries. Second, texture borders are required for each level of each tile if correct sampling is to be performed at tile boundaries. Third, the level of detail mechanisms that are necessary to address real-time paging issues can only take place at the granularity of the tiles themselves—producing disconcerting visual pops of whole texture/geometry tiles. These limitations mean that this approach not only limits visual effectiveness, but ultimately adds an entire extra level of complexity to geometric modeling, morphing, and LOD definition—all of which must take tile boundaries into account.

A more advanced but more complex approach provides several different textures of different resolutions. Each of these levels roams based on its pre-determined relationship to the eyepoint. Thus polygons slide from one texture to another where the textures closer to the eyepoint are of higher resolution. The drawbacks of such a system are that geometry still needs to be tied directly to one of these textures and thus once again, texture LOD decisions are not made at the pixel level but rather at the polygon level. There also needs to be a complex algorithm to subdivide geometry based on the boundaries between the different textures of different resolutions. Further, predefined texture LOD choices are being made by the algorithm itself based on the range from eyepoint rather than the screen space projection of the texels.

An even more advanced solution is outlined by Cosman [2]. This solution involves a sparse multiresolutional representation of texture in hardware. Although similar to our clipmap representation in some ways, Cosman does not generalize the algorithm to be useful beyond the limited hardware support. Further, the algorithm does not seem to be a look-ahead cache approach, meaning that it does not appear to stage data into texture before it is needed. Cosman also does not address load control or dynamic selection between image quality and texture update time at a fine granularity. Further, although the system takes advantage of spatial coherence, it has a fixed algorithm that determines what texture data must be resident for a given eyepoint. Overall, although this approach solves a large scale problem, it does not appear to be generalizable to address the goals of §1 or one that could easily be integrated into existing applications.

3 Clipmap Structure and Representation

3.1 Observations about Mipmaps

A clipmap is an extended dynamic representation of a partial mipmap. Since it builds on the mipmap foundation, we will summarize the key concepts. A *mipmap* is a collection of correlated images of increasingly reduced resolution arranged, in spirit, as a resolution pyramid[1]. Starting with largest and finest level (level-0), each lower level represents the same image with half the texels in each dimension as the level above. This structure is illustrated in Figure 1.

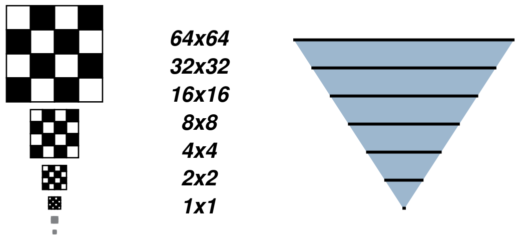

Figure 1: Mipmap Image Pyramid and Side-View Diagram

This relationship is represented (with $0 \le i \le log_2$ TextureSize[0]) using the C-language's bit-shifting notation as:

TextureSize[i] = TextureSize[0] >> i
TexelSize[i] = TexelSize[0] << i

It is important to understand that while each level has fewer texels, the texels in each level represent correspondingly larger regions of the top level image, ending in a single pixel version of the entire image in the bottom 1x1 level.

When rendering with a mipmap, pixels are projected into mipmap-space using texture coordinates, underlying geometric positions, geometric transformations, and texture transformations to define the projection. Each pixel rendered with a particular texture in the course of making an image is derived from one or more texel samples taken from one or more levels of the mipmap hierarchy. In particular, the samples chosen are taken from the immediate neighborhood of the mipmap level where the screen pixel-to-texel mapping is closest to a 1:1 mapping, a level dictated by screen resolution. These samples are then filtered down to a single value for further processing.

Given this simple overview of mipmap processing, we now consider which texels within a mipmap might be accessed during the rendering of an image? Clearly there can be many variations in the factors that control pixel to screen projection, such as differing triangle locations within a large terrain database, so it may seem that all of the texels are potentially used if the entire universe of possible geometry is taken into consideration. Refer to Figure 2 for a simple illustration of the relationship between eyepoint position and texel access.

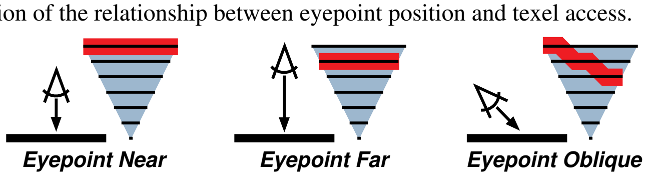

Figure 2: Texel Access within a Mipmap

Interestingly, while typical mipmaps are potentially fully accessed during rendering, this is only due to the fact that typical mipmap sizes are less than twice the size of the maximum screen extent. If the indexing arithmetic for a texture larger than this is analyzed, it becomes clear that there are places in the mipmap pyramid that will not be used in the rendering of a single image no matter what geometry is rendered, so long as the texture projection matrix is unchanged.

That this should be so derives from the fact that screen resolution is finite. For example, when rendering a 32768^2 pixel mipmap on a 1024^2 pixel screen, we know that the mipmap sample-selection logic will choose the texels that are closest to having a 1:1 mapping to pixel area. Given this, it will index at most 1024 texels in a level before accessing a higher or lower level within the resolution hierarchy. The number of potentially accessed texels is twice this in implementations that blend samples of adjoining levels (as in *trilinear mipmapping*). Refer to the center diagram in Figure 2, where texels from level 1 are being used for an example of this. If the eyepoint were just slightly lower, then samples would be blended from both level 0 and level 1, with 1024 texels from level 1 would still accessible. The corresponding texels from level 0 needed to blend with these 1024 level 1 texels are distributed over a 2048-texel extent.

The basis of our system is this realization that screen resolution controls access into the mipmap and that in the case of the very large textures we are concerned with, only a minute fraction of the texels in the mipmap are accessible. This makes a case for building hardware to exploit this fact by rendering from the minimal subset of the mipmap required to be resident for each frame—a structure we term a clipmap.

3.2 The Anatomy of a Clipmap

A clipmap is essentially a mipmap where levels have been “clipped” to a specified maximum size. This parameterization of a clip map defines the shape of the resulting clipmap (as opposed to mipmaps that have an inherent shape) and thus defines the size of the texture memory cache needed to fully represent the texture hierarchy. This shape is defined concisely by two parameters.

3.2.1 Defining Clipmap Size and Structure with ClipSize

ClipSize represents the limit, specified in texels, of texture cache extent for any single level of a clipmap texture. Thus each level of a normal mipmap is defined to be “clipped” to ClipSize if it otherwise would have been larger than ClipSize, as shown in Figure 3. Each level still has the same logical size and thus render-time accessing arithmetic as the corresponding layer in a full mipmap.

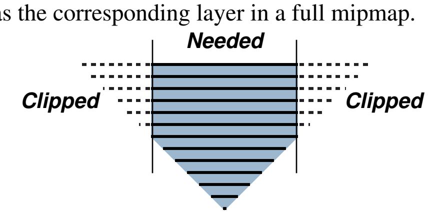

Figure 3: Clipmap Region within Mipmap

Note that even though the same number of texels, ClipSize^2, are cached on each of the clipped layers of a clipmap, each layer (n) obeys the mipmap rule that each texel represents four times the area at half the resolution in each dimension as the next finer level (n-1).

Based on this mipmap subset representation, we define the ***Clipmap Stack*** to be the set of levels that have been clipped from full mipmap-size by the limit imposed by ClipSize. These levels are not fully resident within the clipmap, rather they are cached with only a ClipSize^2 subset present at once; these are the topmost levels in Figure 4. Below the Clipmap Stack is the ***Clipmap Pyramid,*** defined as the set of levels of sizes not clipped by the ClipSize limit. These levels are completely contained in texture memory and are identical to the corresponding portions of the full mipmap.

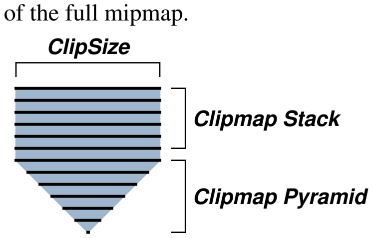

Figure 4: Clipmap Stack and Pyramid Levels

These definitions, Clipmap Stack and Clipmap Pyramid, may be more fully understood if the clipmap structure is considered as a three-dimensional entity. In this form, we can imagine the full mipmap as an inverted pyramid and the Clipmap Stack as a “mine shaft” located at the center of level-0 in the pyramid and extending down toward the apex. (This restriction on the level-0 location of the mine shaft will be lifted shortly.) As the shaft nears the bottom, it intersects with the sides of the mipmap, and this intersection defines the demarcation between the Clipmap Stack and Clipmap Pyramid regions.

3.2.2 Defining Clipmap Contents with ClipCenter

Given the notion of clipping a mipmap to fit in a finite clipmap cache, we define the data present in this cache by specifying a ***ClipCenter*** for each stack level. The ClipCenter is specified for both S and T texture coordinates. By defining ClipSize and ClipCenter for each level, we have precisely defined the texture region being cached by the ClipStack levels of our representation:

Texture Region Size[i] = ClipSize
Texture Region Origin[i].S = ClipCenter[i].S-Texture Region Size[i]/2
Texture Region Origin[i].T = ClipCenter[i].T-Texture Region Size[i]/2

One implementation approach is to specify the ClipCenter for stack level-0 and then derive the ClipCenter of lower levels by shifting the center based on depth. This forces each level to be centered along a line from the level-0 center to the clipmap apex creating the prismatic mine shaft through texture memory suggested previously. This is the center indicated at the top of Figure 4. This centering method loses granularity at coarser resolutions and requires special handling of subtexel motions in the lower layers. Although slightly less than ideal when rendering perspective scenes, this approach simplify application level implementations. In this method:

ClipCenter[i].S = ClipCenter[0].S >> i
ClipCenter[i].T = ClipCenter[0].T >> i

This type of centering yields concentric rings of resolution surrounding the level-0 ClipCenter. The center location may be placed anywhere in full mipmap space based on viewpoint and viewing frustum. The image in Figure 5 shows an orthogonal view of a polygon (using the “Eyepoint Near” viewing position of Figure 2) that has been textured with a clipmap having a very small ClipSize in order to demonstrate the concentric rings of texture resolution.

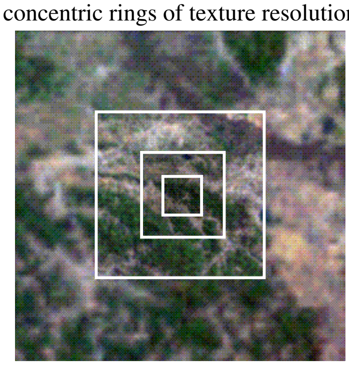

Figure 5: Rings of Texture Resolution

Note it is possible (and perhaps important) to center the clipmap arbitrarily. Despite this, most users center the clipmap either beneath the eyepoint or at a location derived from intersecting the viewing frustum with a representative ground plane. Although centering beneath the eyepoint allows for high-speed rotation invariant of clipmap cache update rates, applications that use far field telescope-like views may need to place the center where they are *looking at* rather than where they are *looking from.*

In normal use the ClipSize would be set to be greater than or equal to screen size, in which case the rings of resolution shown in Figure 5 would be just large enough that, with a properly chosen center, the rings will form a superset of the needed precision. In such a configuration the subset nature of clipmaps will never be a limiting factor when choosing texture samples—every texel accessed will be present in the clipped mipmap and the resulting image will be pixel-exact with one drawn using the full mipmap.

3.2.3 Invalid Border for Parallel Update

In addition to these basic definitions, an additional parameter is needed to support efficient parallel updates and rendering with clipmaps. First, although the previous definition presents each Clipmap Stack level as a cached subset of the complete mipmap level, it does not address real-time paging needs in terms being able to pre-page data asynchronously that will be needed for future frames while concurrently rendering from the clipmap. In order to make clipmaps well suited to this type of paging and for load control, we introduce another parameter called the ***InvalidBorder***, defined as a border of texels existing within each stack level that is InvalidBorder-texels from each edge, as shown in Figure 6.

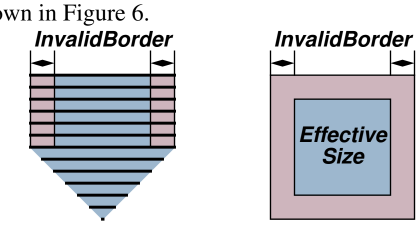

Figure 6: Clipmap with InvalidBorder and EffectiveSize

The active portion of a clip map is simply ClipSize minus twice the InvalidBorder, which defines the ***EffectiveSize*** of each stack level. Thus our definition of the cached region of stack levels is adjusted as follows:

EffectiveSize[i] = ClipSize - InvalidBorder*2
Texture Region Origin[i].S = ClipCenter[i].S - EffectiveSize[i]/2
Texture Region Origin[i].T = ClipCenter[i].T - EffectiveSize[i]/2

Texels within the InvalidBorder are not accessed by the texel-sample gathering hardware when rendering using the clipmap representation. This border provides a destination region where higher level software can safely prefetch data for future frames asynchronously with clipmap access for rendering.

3.2.4 TextureOffset for Incremental Update

The last fundamental parameter of our low-level clipmap representation is the ***TextureOffset***, defined as the starting address used by the texture address generator to start addressing the clipmap EffectiveSize from the ClipSize^2 memory region allocated for each Stack Level. The addressing unit can then use modular addressing to logically roam through a large texture within a static ClipSize^2 memory region. This modular addressing scheme is common in image roaming applications that use a fixed memory region to incrementally roam through a larger 2D image. This offset is specified per-level and affects addressing for the level exactly as a texture matrix translation would when the texture’s clamp mode was set to WRAP. Use of the TextureOffset parameter are discussed in greater detail in §4, where the process of updating and rendering with a clipmap is described.

3.3 Clipmap Storage Efficiency

Let us for the purpose of clarity define a typical small clipmap of 16 levels and thus of size 32768x32768. This texture will be rendered on a screen of size 1024x1024. For our first configuration, the user has requested the texture appear with full fidelity and will tolerate no visual artifacts. Given the screen size, we know that the upper bound on texture usage from a single level is 2048x2048 (and is a rare circumstance indeed), however, to ensure visual fidelity we set the ClipSize to 2048. This dictates that there will be four clipped levels forming our Clipmap Stack and 12 levels forming the Clipmap Pyramid. The texture Storage required for this configuration is 2048x2048 texels * 4 levels + 4/3 * 2048*2048 for the pyramid = 42.7 MB at 2 bytes per texel. Even in this conservative configuration, our texture representation requires only 42.7 MB of the full 2.8 GB present in the complete mipmap. Further a more typical configuration using a 1024 ClipSize will achieve nearly identical results and requires only 1024*1024*5 levels + 1024*1024*4/3 = 12.7 MB of the full 2.8 GB. And finally, a 512 texel stack still yields reasonable results for perspective views and requires a mere 512*512*6 + 512*512*4/3 = 3.7 MB of storage. In general, for a 2^nx2^n clipmap with a ClipSize of 2^m, only $4^m(n - m + 4/3)-1/3$ texels of fast texture storage is required. For full mipmaps, the storage needed is $(4^{n+1} - 1)/3$ texels. Both of the equations must be scaled by the number of bytes per texel to compute physical memory use. From these equations, we can derive the ratio of clipmap to mipmap storage, as a function of the number of levels in a full mipmap, n, also the exponent of the top 2^nx2^n level of the mipmap; and m, the number of pyramid levels in the clipmap and the exponent of the 2^mx2^m clipmap stack and pyramid size.

$$\frac{ClipmapStorage}{MipmapStorage} = \frac{4^m(3n-3m+4)-1}{4^{n+1}-1}$$

Note that total mipmap storage grows exponentially yet clipmap storage grows only linearly, confirming the clipmap system's ability to handle much larger textures than mipmap-based approaches. This advantage is tabulated in Table 1, where KB = 1024 Bytes, and 16-bit texels are used. The final column shows memory use for a 2^{26}x2^{26} image large enough to represent the earth at 1 meter per texel; the 34.7 MB clipmap requirement fits nicely into the texture memory of a modern high-performance graphics workstation, while a mipmap system with 10,923 TB of texture memory is completely impractical—it would require 5.7 billion 16 megabit DRAM chips!

Type	*512^2*	*1024^2*	*4096^2*	*32768^2*	*67108864^2*
Mipmap	682KB	2.7MB	42.7MB	2.7GB	10923 TB
Clipmap 512^2	682KB	1.1MB	2.2MB	3.7MB	9.1 MB
Clipmap 1024^2	682KB	2.7MB	6.7MB	12.7MB	34.7 MB
Clipmap 2048^2	682KB	2.7MB	18.7MB	42.7MB	131.7 MB

Table 1: Clipmap Storage Requirements

3.4 Low-Level Clipmap Summary

As explained in this section, clipmaps define a look-ahead cache mechanism for representing the necessary subset of a mipmap; a representation that can be both rendered from and incrementally updated asynchronously. Multiresolution datasets such as mipmaps are well suited to this type of representation because they always contain all of the relevant data at some resolution in the pyramid levels. This means that for real-time behavior, if the algorithm ever misses the cache by attempting to access a texel not available on a given level (due either to an overlay small ClipSize or a poor choice of ClipCenter), the renderer can use a lower resolution version of the data from a lowe-level of the clipmap stack or pyramid rather than waiting for the cache to be loaded with the desired data. This "clip texture accesses to best available data" nature is a second reason why we chose the name clipmap for this approach. Moreover, 2D images are especially well suited to be cached via a 2D sub-image that roams through a static region of memory of finite size using modular addressing. The cache system consists simply of several levels of 2D image roaming caches which are driven by the specifying the center of resolution for each stack level. Future implementations could add multiple stacks and centers to deal with more complex application requirements at the expense of implementation cost and complexity.

4 Updating Clipmaps

Clipmaps are defined as spatial caches that are, in practice, just large enough to effectively cache the texture used to render one view. Therefore, as on might expect, it is important to refresh the cache often as the view changes. (However, note that even if the clipmap is not updated, the clipped-indexing described in the previous paragraph ensures that the proper texture—though less sharp than desired—will be used for the image.) This update is performed by considering the cached texture regions for each level as described in §3 and applying them to the texture memory. Moreover, to take advantage of the significant frame-to-frame coherence in center locations and cache contents, we must arrange to not flush each level's cache when the cache changes slightly. Thus we define each level's cache as an independent 2D image cache that roams in place through a larger image (the entire region of that level within a complete mipmap) using toroidal addressing. We then update each level incrementally by downloading the new data into the cache and letting the toroidal addressing dictate where it goes. Because we are defining a storage scheme, we must make sure to use the same addressing when reading memory to render the texture. The four steps of this loading/addressing scheme is presented in Figure 7. This process begins with a 2D image centered in the cache as shown in the left diagram. We then specify a new ClipCenter, in this case *d* texels above and to the right of the old center. Examining both the old and new image regions, it is clear that the central area is indeed the same in both cases, so only the border areas marked *T* (top), *C* (corner), and *R* (right) require updates.

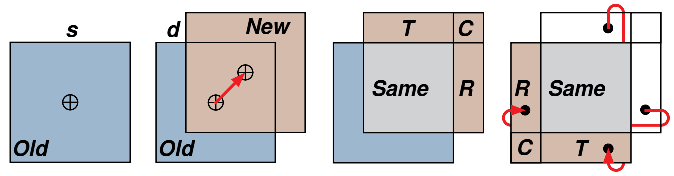

Figure 7: 2D Image Roam using Toroidal Addressing

Implementing toroidal addressing is simple: compute the virtual address of a texel by adding the new ClipCenter and the texel address and then mask any high-order bits greater than the size of the cache. In this way the new data at the top of the image is actually loaded into the bottom, data on the right is loaded into the left, and the upper left corner is loaded into the lower right. Further, the cache start address moves from (s/2, s/2) to (s/2+d, s/2+d). The TextureOffset for this level is set to this new origin address, which communicates to the addressing unit where the cached image starts in the context of modular addressing.

Now, given that each stack level roams independently based on its center and assuming that we are centering all the levels based on the same ClipCenter, we can visualize the complete multi-level update process as shown in Figure 8.

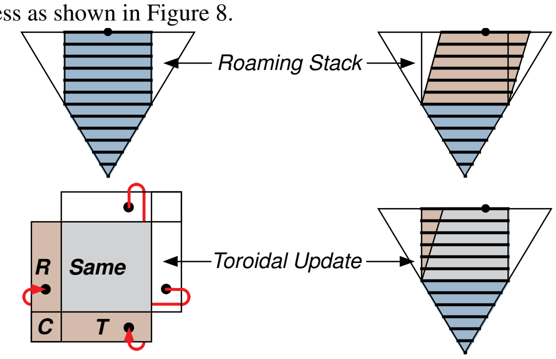

Figure 8: 2D Image Roam of Complete Stack

This side view of the toroidal indexing scheme shows how the entire set of stack levels appears to have moved when in fact all loads are performed in place as described in the previous paragraphs. Also not that the areas of overlap are unchanged and require no update. In practice, the overlap area is often relatively large and only minor paging at the edges is required to update a clipmap stack. Note also that since lower levels are coarser resolutions as in normal mipmapping, the same movement of center point will result in only half as much movement of a lower level than the level above it.

Given this incremental loading scheme and concentric caches for the clipmap stack levels, we note that the Texture Offset parameter behaves exactly like the Center Parameter and is computed as

```
Texture Offset[0] += TextureCenter[0] - OldTextureCenter[0]
Texture Offset[i] = Texture Offset[0] >> i
```

Also keep in mind that if the movement of the TextureCenter is smaller than the InvalidBorder for a given level, then the incremental region can be loaded even while texture is being drawn with the previous center. This is true because InvalidBorder guarantees a border of texels that will not be used where asynchronous loads may take place.

4.1 Hardware Implications

The clipmap cache updating scheme described above directly affects the overall design of the hardware. The hardware must not only be able to handle this type of representation and paging, but it must be robust enough to do so at 60 Hertz. This means that effective bandwidths for real texture downloads must be synchronized with expectations of roam speeds through texture. The upper limit on cache update time is established by how long it takes to completely flush the clipmap cache. In the case of a 32768x32768 virtual clipmap with a 1024 stack, which is what our system is designed around, flushing the cache levels would take 10MBytes of texture paging. Given host-to-graphics texture throughput of 100-400MBytes per second depending on system and usage, such cache flushes should take somewhere between 1/40th and 1/10th of a second. This means that although the cache cannot be completely flushed without affecting frame rate, the cache center can be moved a distance of more than 1000 texels on the highest level map and still meet real-time bandwidth constraints for the overall system. It is important however, to realize the overhead of the download itself—this overhead can make smaller downloads more significant than the required bandwidth alone would suggest.

5 Rendering with Clipmaps

Rendering with a clipmap is essentially the same as rendering with a mipmap. The only modifications to the normal mipmap rendering are slight changes to the level-of-detail selection algorithm and to the texture memory addressing system. However, it is important to note that building clipmapping support into low level hardware is crucial. Although clipmapping requires only minor extra computation compared to mipmapping, this extra computation is per pixel in the heart of the addressing algorithm. Not supporting the low level representation as a core part of a texture implementation would mean throwing away any performance gained with texture hardware when implementing clipmapping.

With this in mind, here are the steps to sample texture for a given pixel using a clipmap representation:

1. Determine the s and t coordinates for the current pixel. This is done exactly as addressing the equivalent mipmap we are virtualizing.
2. Calculate the level-of-detail based on normal mipmapping criteria. This calculation yields a floating point LOD where the fractional part of the number represents blending between two LODs. If this LOD resides completely in the pyramidal portion of the clipmap then we simply proceed with addressing the appropriate LODs as a normal mipmap. If the finer LOD being blended resides in the Clipmap Stack and the coarser LOD resides in the pyramid, then we address the coarser LOD as a mipmap but continue on the clipmap path for the finer LOD addressing.
3. Calculate the finest LOD available given the texture coordinates established in Step 1. This calculation is done by comparing the texture coordinate to all of the Texture Regions described in §3. This calculation needs to be done in case the texels are not present at the LOD requested. There are several hardware optimizations that can be performed to make this Finest LOD for a texture coordinate calculation tenable and efficient in hardware. For cost sensitive solutions, a restriction can be made where all levels must be centered concentrically, such that calculating a simple maximum distance from the TextureCenter to the texture coordinate generated can yield the finest available based on the following:

   ```
   Sdist = roundUp(abs(Scenter - s))
   Tdist = roundUp(abs(Tcenter - t))
   MaxDist = max(Sdist, Tdist)
   FinestLODAvailable = roundUp(log2(MaxDist) - log2(ClipSize))
   ```

   The equation for FinestLODAvailable does not consider the InvalidBorder designed to prevent accessing texels that are in the process of being asynchronously updated. Thus after we have computed this LOD, we make another simple computation to see if we are in the InvalidBorder of the selected FinestLODAvailable. If we are in the InvalidBorder (that is, if MaxDist > ClipSize - InvalidBorder), then we simply add 1 to the FinestLODAvailable. Having calculated the LODmip and LODAvailable, we simply use the coarser one. Once again, it is worth noting that it is possible to configure a system and application such the LODAvailable is never limiting the aforementioned LOD calculation. This is done by selecting a sufficiently large ClipSize and placing the ClipCenter appropriately.
4. Convert the s and t coordinates into level specific and cache specific coordinates. First, we split the calculation into two to account for the blending between two active LODs (fine and coarse).

   ```
   Sf = (s >> LODclip) - 0.5
   Tf = (t >> LODclip) - 0.5
   Sc = (s >> (LODclip + 1)) - 0.5
   Tc = (t >> (LODclip + 1)) - 0.5
   ```

   Then, determine the offsets necessary to address the fine and coarse LOD given the data currently cached inside each level:

   ```
   Sfoff = ClipCenter[LODclip].S - TextureSize[LODclip]/2
   Tfoff = ClipCenter[LODclip].T - TextureSize[LODclip]/2
   Scoff = ClipCenter[LODclip+1].S - TextureSize[LODclip]/2
   Tcoff = ClipCenter[LODclip+1].T - TextureSize[LODclip]/2
   ```

   Finally, determine the actual S,T address within the clipmap Level cache for both the fine and coarse LODs by using the actual texture coordinate, the recently computed center offset, and the user specified TextureOffset. It is necessary to use modular addressing to be consistent with the way that §4 describes how the maps are to be updated based on the TextureOffset.

   ```
   Sclipfine = (Sf - Sfoff - TextureOffset[LODclip].S)% ClipSize
   Tclipfine = (Tf - Tfoff - TextureOffset[LODclip].T)% ClipSize
   Sclipcoarse = (Sc - Scoff - TextureOffset[LODclip+1].S)% ClipSize
   Tclipcoarse = (Tc - Tcoff - TextureOffset[LODclip+1].T)% ClipSize
   ```
5. Use the two sets of texture coordinates (fine and coarse) to generate a filtered texture value for the pixel exactly as for a mipmap texture.

6 Virtual Clipmaps

At this point, we have defined our low-level clipmap representation and described how to update and render it. Now, we need to handle limitations that are encountered during implementation and show how to generalize beyond those limitations.

The main limitations are those right to the heart of all mipmap texture implementations—namely texture memory addressing range and precision. Previous mipmap hardware implementations (and software for that matter) have not had to deal with addressing data of the size and scope of clipmaps. Previously, textures of 4096^2 where considered enormous and users where grateful to store even one such texture into hardware. This means that hardware texture systems were implemented with internal precision that met or possibly slightly exceeded this type of standard addressing. When implementing clipmaps, because a fundamental strength is thinking of the texture as one contiguous texture, it is important to keep texture coordinates at full precision until the actual texture addressing. However, doing so means adding precious bits to the main paths through texture hardware. Adding these bits has an almost linear effect on the gate count and cost of the texture hardware. Thus a decision has to be made—how big of a clipmap is big enough?

Luckily if we take clipmapping for what it is, a robust texture memory virtualization scheme, we can find the insights to trade this question for two others.

Clipmaps work because it is impossible to access all of the data stored in mipmaps when they are even slightly larger than the screen on which they will be rendered. Clipmaps hold the subset of the mipmap that is necessary in cache and use it for rendering. We need to think about virtualizing clipmap hardware in the same way. With slight extensions to the clipmapping algorithm, we can use the finite number of address bits provided in hardware to virtualize larger clipmap memories. This can be done for clipmap memory much as it is done when virtualizing real memory. We can limit the range of our texture address by limiting the number of texture levels that we access. Further, we need only provide a scale and bias at the front end of texture hardware to account for this virtualization. The process is simple—for any polygon, determine the range of texture LOD that will be used, ask the hardware to clipmap using only those levels while providing it with texture coordinates in the appropriate coordinate space. This coordinate space adjustment can be done post transform with a simple scale and bias to continue to keep the address virtualization scheme transparent to the user. Essentially this is be defining a sub-clipmap out of the virtual clipmap that the hardware will then address to texture a given polygon. This allows us address precision accurate to the limit of either user specification or coordinate transformation. With a 32-bit internal implementation, this is sufficient to represent the entire earth at one centimeter-per-texel resolution. Based on this approach the new addressing questions are: how much addressing do we need to satisfy the dynamic range for rendering a single polygon and how does this affect the underlying clipmap representation?

Dynamic range of texture addressing is difficult to gauge when using mipmapping. Normal LOD selection in cases of edge-on extreme perspective gives us large variance of addressing in terms of LODs. We have to draw the line somewhere and this will most likely be determined by the trade-off between expected standard database sizes and the cost targets for the underlying silicon. Our implementation provides 16-level addressing to support a 32768:1 maximum dynamic range on a single polygon.

It is important to remember that as with any good virtualization scheme, it is best not to affect the user's representation of what we are virtualizing. We want the user to simply specify an *n*-level mipmap and a ClipSize to define clipmaps of any size. This means that the hardware must be designed to address a sub-clipmap out of an arbitrary clipmap when the depth exceeds the directly supported precision (16-levels in our implementation). Given the stack and pyramid structure defined in a clipmap, we observe three possible types of addressing that must be supported as illustrated in Figure 9.

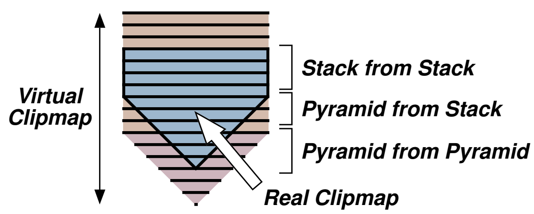

Figure 9: Virtual Addressing in Stack and Pyramid Levels

1. Address a stack level as a stack level. No address modifications necessary other than the upstream scale and bias.
2. Address a stack level as a pyramid level. Need to provide an offset into the level to get to the correct data. This can be done by simply starting from the center texel, and offsetting backwards to find the origin texel of the sub-level based on desired level size.
3. Address a pyramid level as a pyramid level. This can be implemented once again by simply knowing the actual level size, finding the center texel, and offsetting backwards to find the origin texel of the sub-level based on desired level size.

These three simple cases argue for additional addressing hardware to switch between the normal stack address processing (normal clipmap addressing as described in §5), sub-addressing as defined above for cases 2 and 3, and normal mipmap addressing. With these low-level addressing extensions, we can write special low-level software to automatically select sub-clipmaps and provide the scale and bias of coordinates. This enables users to write applications with little or no regard to the underlying address virtualization. They just need to keep in mind that no polygon can ever span more than the low level dynamic range or it will alias or blur. Of course there are still several challenges in managing the underlying representation as well as computing dynamic range per polygon. We initially provide an interface to allow applications to set the dynamic range at a cost similar to activating a texture.

Why add additional complexity to an otherwise simple low-level implementation? Because users truly require this type of virtualization. From a hardware standpoint, we cannot provide enough addressing bits to service the very large texture data that users need to visualize. For customers dealing with databases of this type, going from a typical large 2048^2 (11-level) mipmap to the 32768^2 (16-level) clipmap directly supported in hardware only helps incrementally by improving visual quality and making systems easier to use. Without the additional support to enable virtual clipmaps, the system would not enable the new applications users seek to build. However, with these capabilities, our clipmap implementation is able to scale up to extreme resolutions beyond current data gathering technologies. Further, since the system can represent the largest textures required by applications in this unified manner, then database modelling, geometric LODs, texture insets, and other application tasks can be completed much more easily.

We also mention that extending clipmaps to be three dimensions is possible. 3D clipmaps would be especially well suited for roaming through gigabytes of volumetric data because normally this is done in the context of semi-transparent volumes. Roaming through semi-transparent volumes means that, in general, voxels further from the viewpoint will contribute less to the final image. It seems appropriate that 3D clipmaps could inherently store concentric sub-volumes of different resolutions such that the small contributions by distant voxels would happen at coarser resolution.

7 Dynamic Clipmap Management

At this point we have described the details of everything necessary to build a robust low-level clipmap implementation able to store and render clipmaps in texture hardware at full mipmap performance. For systems of this complexity, defining a clean underlying solution is only half the solution. Application developers and end users want more than a machine capable of drawing clipmapped pictures at 60 Hertz, they want a comprehensive system that robustly does so with optimal performance and little application interaction. Further, they want the system to do this based on whatever resources are available and with minimal effort to input and manage the clipmap. Essentially they would like to think of clipmaps as mipmaps that require far less texture memory, allocate whatever system memory may be required for intermediary caches, and interact with the filesystem to access data stored on disk.

Remember that despite user desires, clipmaps can be so large that we cannot even think of storing them in main memory. Even without virtualization of hardware clipmaps, how many users have systems with 3 GBytes of memory and enough time to load the complete texture into it before rendering? Thus, a complete solution needs to address another level of texture caching. The low level representation uses texture memory to cache a small amount of the clipmap, and it follows, that the high level representation needs to use similar look-ahead caching to manage main memory resources. To talk about the implementation of clipmaps at this level, we need a software context. This context is found in a toolkit level software package [3] that already provides a multi-processed rendering pipeline, process management, and higher level texture primitives. Such a toolkit gives users considerable leverage and can be used to write a large range of applications.

7.1 Main Memory as Second-Level Cache

To use main memory as a real-time 60 Hertz second level cache for clipmap data, several mechanisms must be defined. First, we have to provide a representation that allows information to flow in from disk and efficiently down to texture memory. Thus we must optimize throughput on two different paths—we need to optimize throughput from disk to memory and we need to optimize throughput from memory to the underlying hardware clipmap cache. As for the rest of the software rendering system, if we wish to optimize utilization of system bus resources we need to have separate threads of control moving data and managing it. Thus we need at least one process scheduling data flow, one process moving data from disk to memory, and one process incrementally feeding the graphics subsystem clipmap updates.

In the context of our software toolkit, this means that scheduling is done in the software rendering pipeline in the Cull process, downloading of data to the graphics pipeline is done in the Draw process, and a new lightweight asynchronous disk reading process must be created to direct the disk to write data into certain main memory locations.

To manage our second level cache efficiently and effectively, we must define the page size at which the cache operates. Because the cache can be thought of as a second level image roam, we can think of it has having pagable units which are in fact image tiles. These tiles must be large enough to ensure maximum throughput from disk, but small enough size to keep overall paging latency to a minimum. We use sizes between 128x128 and 1024x1024 depending on disk configurations. In order for this second level cache to operate as a real-time look-ahead cache, we must be able to load into parts of the cache that are not currently needed by the underlying hardware. This implies that we must have each cache level contain at least enough tiles to completely hold the underlying clipmap level that is currently resident while prefetching border tiles. Thus if our underlying ClipSize was 1024 and our tile size was 256x256, then our cache must be configured to be at least 6 tiles by 6 tiles or 1536x1536, as illustrated in Figure 10.

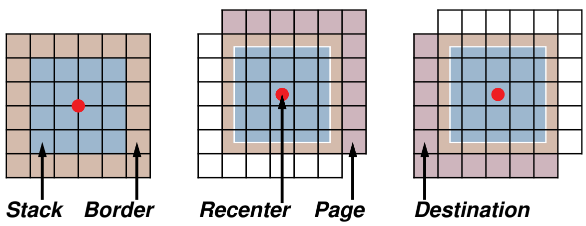

Figure 10: Tiled Main-Memory Image Cache

Due of memory alignment issues, low-level texture updates must be broken up into individual downloads based on tile boundaries. In Figure 10, the left drawing shows a minimal cache for one stack level. This cache has already copied the texels in the ClipSize^2 central area down to the clipmap in texture memory. The cache has also prefetched the minimal single-tile border of image tiles (the border indicated in teal). The center illustration indicates that the center (the red dot) has been moved up and to the right by half of a tile (128 texels, in this case). What actions are implied by this recentering action? First, the clipmap must be updated. This is done by accessing the border tiles above and to the right of the existing stack data. The white square indicates the new clipmap area, and the nine crosshatched rectangles are the texture downloads required to update the hardware clipmap stack level. We do these multiple downloads because it is infeasible to perform real-time memory to memory copies to realign memory when overall texture loading rates are in excess of 100 MB/second and peaks are several times higher. Once the clipmap layer is updated, the tile-cache must be roamed up and to the right, by scheduling disk reads to load the newly uncovered tiles. While the new tiles are logically up and to the right, we store them at the bottom and the left, using the same toroidal addressing in the tile cache that is used in the low-level hardware clipmap—in this case as in that one, unchanged data is never touched.

From these details, our system-level clipmap representation can be seen to look very much like a larger tiled version of the underlying hardware representation. It has a section of static pyramid-levels and a section of cached roaming clipped-levels. These levels can be thought of as independent image caches that roam independently through their level of the full mipmap texture. All levels are performing look-ahead paging by requesting disk reads for image tiles that will be needed in the hardware in the near future. Each level also is responsible for managing the corresponding level of the underlying hardware clipmap implementation. Managing this resource means recentering the underlying hardware level and incrementally downloading the relevant texture data from its tiled cache. To configure this representation, users provide information for each clipmap level. For clipped levels, they provide image cache configurations that describe how to load tiles (such as an *sprintf* format-string to convert tile row and column indices into a full filesystem path name for each level), how big the cache should be, how big the underlying hardware clipmap level is, and other basic information. For pyramid levels, they simply provide the static image. Global clipmap data is also provided, such as the ClipSize for the clipmap, the storage format of the texels, and possibly some parameterization for load control.

7.2 Cache Centering Algorithms

In addition to the low and high level representations we have defined so far, there still have one important issue we have not yet discussed—how to decide what part of the image should be cached. At the low level and even at the toolkit level we allow (or force) the user to set the center. The reason for this is that the optimal place to center of the cache is very application dependent. Some applications might be based on narrow fields of view and may often be looking at data in the distance in which case probes into the database may be required to determine the proper location for the ClipCenter. Others, such as head-tracked ground-based virtual reality applications will want to ensure that quick panning will not cause the clipmap cache to be exceeded, arguing for placing the ClipCenter at the viewer's virtual foot position. From experience thus far, we feel most users will find that a weighted average of intersections of the viewing frustum and database with the projection of the viewpoint onto the ground provides an effective low-effort centering algorithm.

Our clipmap system currently provides utilities to automatically set the clipmap ClipCenter based on either of simple viewpoint projection or simple frustum intersections, and we plan on providing the weighted average approach as the default in the future.

7.3 Automatic Load Management

Earlier in this section we defined the structure of the second-level tile cache system much as we specified the clipmap representation itself. To justify spending time collecting, processing, registering, orthorectifying, and tiling huge texture images on disk, users need to know that applications will display this data at full performance and high fidelity. Since 60 Hertz real-time image generation is such an overriding concern, we need to show how clipmap updates are handled when disks or other parts of the second-level cache system do not keep up with tile loading demands. In practice, graphics applications are made more difficult because they do many tasks in addition to clipmapping, such as loading mipmap textures, drawing geometry, performing image processing on frame-buffer contents, and interacting with video devices. Users wish to do these tasks without needing to manage the system-level issues of clipmap paging load. Thus, being able to make sure that clipmap updates do not become a bottleneck for application performance is a significant design consideration.

When thinking about seamless performance and load control we organize our ideas into several levels of tuning. First, we provide a solid infrastructure and basic representation that can be configured in such a way as to truly allow real time behavior. This paper has thus far provided considerable detail discussing these underlying structures in both hardware and software. Next, we will address one potential system bottleneck at a time, removing as many dependencies between real-time frame rates and the underlying configuration issues as possible. In the case of our current implementation, this has meant starting from the back of the system where users have their data on disks in the filesystem and working our way forward.

With system-to-graphics throughput rates of 200-400 MBytes/second, individual disk bandwidths of 3-9 MBytes per second are comparatively slow. This could mean that clipmaps would be able to roam through the extent of their second-level tile cache at incredibly fast rates, but that they would request tiles from disk faster than could possibly be serviced, resulting in a performance bottleneck when attempting to roam the tile cache. In our initial implementation this meant that users who did not have disk configurations appropriate for clipmapping tasks (arrays of striped disks) had a system that was unusable for clipmapping. The clipmap software would either have to wait for data to be paged in from disk (note the order of magnitude difference in throughput rates described earlier), or to download arbitrary generic data as a placeholder. We find neither solution to be acceptable since we already have the data requested present in the clipmap albeit at coarser resolution than desired. The real solution is for system to disable accesses to a hardware Stack Level if updating that level requires data that is unavailable. This is specified using a parameter known as **MaxTextureLOD**, a feature already present the hardware. The clipmap software system attempts to catch up to outstanding paging requests, and will enable the level (by changing the MaxTextureLOD) as soon as the main memory texture tile cache is up to date, but it does this asynchronously and will not hold up the rest of the rendering system awaiting tardy data. This simple technique of removing a dependency of a fast rendering system on a fundamentally slower disk system makes it possible for clipmap users to see their data at realtime rates no matter what filesystem configuration they use. In such underpowered systems it is perfectly acceptable if the texture becomes lightly blurry when the ClipCenter is moved through the data faster than the tiles can be read. To recover from this overload condition once the data is available, the system downloads the level and tells the hardware to incrementally fade that level back in. It implements this by slowly changing the MaxTextureLOD over an interval of a few frames. This avoids distracting jumps in detail as data becomes available. This simple load control technique makes major strides toward general usability.

As a second optimization for paging-rate limited users, we added another lightweight thread to optimize the disk read queue to reduce latency. As cache levels made requests of the filesystem that the disks could not keep up with, the disk read queue would grow. This queue growth meant that as the system became further behind, the paging latency would grow as well since requests were appended to the end of the disk read queue. This in turn caused the system to get further behind and had a “snowball” effect where the system could bog down waiting on old disk requests to complete. To solve this we added priorities to the disk read requests. The priorities are determined based on the StackLevel issuing the request (with higher priorities for coarser data) and on the estimated time when tile will be needed. These priorities are asynchronously updated by the cache software for each level each frame and the read manager thread continuously resorts the disk read queue based on these changing priorities, removing entries that are no longer relevant. This priority sort limits the read queue size and minimizes latency of read requests for more important tiles, thus ensuring that tiles needed to update coarser levels are loaded and used sooner.

Having now addressed the two major filesystem-related performance risks, we now address paging bottlenecks from main memory to texture hardware. Since bandwidth between the system and the graphics subsystem is at a premium in immediate-mode graphics systems, it is important to understand and scale the use of that resource by any aspect of an application. For instance, it is typical for users to want to use at least 75% of that resource sending geometric information to the graphics subsystem which leaves 25% or less of the frame time for incrementally paging textures, including both mipmaps and updates of clipmaps. Thus we must estimate very precisely the amount of time we spend doing downloads for a given frame. To do this, we first create a table of the measured time for texture downloads of each size that we could potentially perform based on clipmap and system configuration. Then, as we schedule downloads, we start from the lowest (coarsest) cache level and work our way up the cache levels one by one. Each cache, in turn provides the list of loads needed to update its level. We take this list, sum the overall time for each level using the predefined download-time table, and subtract this time from the predefined maximum download time allocation. Once we exhaust this pre-allocated time, we do not update additional (finer) cache levels. This allows us to ensure that the coarsest LODs get updated first, since they are both the most important and require the least bandwidth, and to disable higher levels using the MaxTextureLOD mechanism if the needed tiles have not been read.

We can also use MaxTextureLOD for coarse-grain adjustments to required download bandwidth. Since each higher stack level requires four times the paging as the next lower one, adjustment of maximum LOD has a powerful ability to reduce overall paging time. The visual effect of coarsening the MaxTextureLOD by one level is to remove the innermost of the concentric rings of texture precision. Since this has a visually discernible effect, this must be gradually done over a few tens of frames by fractionally changing the MaxTextureLOD until an entire stack level is no longer in use, after which rendering and paging are disabled for that level. This method is illustrated in the right-most column of Figure 11, where the inner resolution ring has been removed by decreasing the MaxTextureLOD.

For fine-grain control of paging bandwidth, we use the InvalidBorder control as described in §3.2.3 to reduce the area of texels that must be updated in each Stack Level. Our implementation allows the InvalidBorder can be adjusted on a per-frame basis with little overhead. Increasing the InvalidBorder reduces the EffectiveArea, reducing the radius of each concentric band of precision. This is illustrated at the bottom of Figure 11, where rendering with a larger InvalidBorder is seen to scale the Stack Levels closer to the ClipCenter. Note that the Pyramid Levels are unchanged by the InvalidBorder—as indicated by the outer region in the diagram.

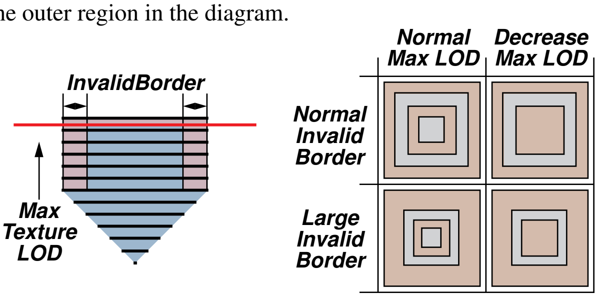

Figure 11: Visual Effect of Load Control

While load control mechanisms even more elaborate than these could be created to modulate the time required to update a clipmap, the simple mechanisms presented here have proven to be effective—they allow existing applications to use clipmaps virtualizing terabytes of filesystem backing store and maintain unwavering 60 Hertz realtime update rates while doing so irrespective of disk speeds and latencies.

7.4 High-Resolution Insets

Insets are regions of extra-high texture resolution imbedded into a larger, lower resolution texture image. They are important since users often have areas within their database that they care more about than others. A typical inset example occurs in commercial flight simulators, where texture resolution at airports must be very high to support taxi, approach, and departure training, while the terrain areas traversed in-flight between airports needs only much more moderate texture resolution.

Because we have defined our clipmaps to be of arbitrary size and because we have integrated back-end load control, support for insets is nearly implicit. We simply allow users to specify sparse tile arrays in the filesystem for each Clip Level. By sparse, we mean clipmaps that are defined to have a certain size, but where some or perhaps nearly all tiles are undefined, resulting in “islands” of resolution in various portions of an otherwise empty tile array (near airports, in the example above). In this case the user defines the giant map, provides only the inset data and the system uses this data it when it can—skipping the empty regions automatically using the TextureMaxLOD load-control mechanism when it discovers that nonexistent tiles are “late” in arriving from the filesystem. (This is one of the reasons why the disk read queue thread discards old invalid requests.) The load control algorithm will naturally fall back on coarser LODs when inset level data is not available.

To allow insets to blend in smoothly, tiles containing the inset texels must be surrounded by a border of at least ClipSize-texels using data magnified from the next lower tile in a recursive manner, a requirement easily met by automatic database construction tools. This restriction exists because the decision to enable or disable a clipmap level is made for the level in its entirety using TextureMaxLOD, and therefore partially valid cache levels—partly in the inset region and partly outside of it—would normally be disabled. By providing this border, users ensure that the load-management algorithm is free to enable the entire level whenever any pixel of the inset is addressed.

8 Implementation Results

We have implemented the clipmap rendering system described here. The implementation consists of low-level support within the SGI InfiniteReality graphics subsystem for virtualized clipmaps; special OpenGL clipmap control and virtualization extensions; and support within the IRIS Performer real-time graphics software toolkit for clipmap virtualization, second-level tile caches, tile paging from the filesystem, and for automatic load-management controls as described above.

In working with the system, we find the results realized from the clipmap approach to be excellent. Real-time graphics applications can now have textures of any desired precision at full speed—examples include whole earth textures from satellite data, visual simulation applications with high-resolution insets, and architectural applications showing murals with minute brush strokes visible at close inspection. Normal mipmaps of 2048^2 or smaller now seem laughably inadequate for serious work, our largest disk array configurations feel confining, and tertiary caching from robotic tape systems is beginning to seem mandatory.

The image in Figure 12 is a frame from this videotape showing an overhead view of a 32768^2 texture image of the Moffet Field Naval Air Station in Mountain View, California rendered onto a single polygon. The colored markers are diagnostic annotations indicating the extent of each clipmap level. The EffectiveArea has been significantly reduced by enlarging the InvalidBorder to make these concentric bands of precision easily visible.

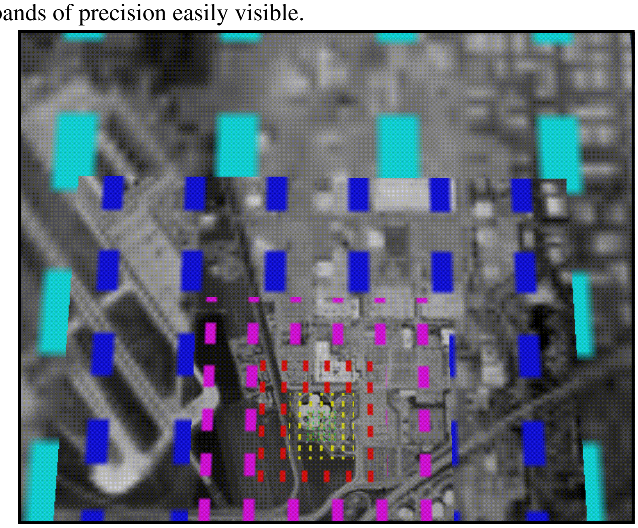

Figure 12: Concentric Resolution Bands

The image in Figure 13 is from a flight sequence on the videotape showing a very small portion of a 25m per texel 32768^2 texture image of California, Nevada, Utah, and Arizona—the area shown here is the southern half of the Yosemite National Park. The prominent rock formation just left of the picture's center is Half Dome, below which the Merced river is clearly visible, winding past El Capitan and extending to the bottom of the image. Mono Lake is in the upper left, just beyond the crossroads city of Lee Vining. That our approach makes this single 2.7 GByte texture displayable at 60 Hertz with a 12.7 MByte clipmap cache is fairly impressive; equally so is the fact that the application being used, *perfly*, does not even know that anything special is happening—all clipmap definition, updating, load-management and default ClipCenter selection is happening automatically within IRIS Performer.

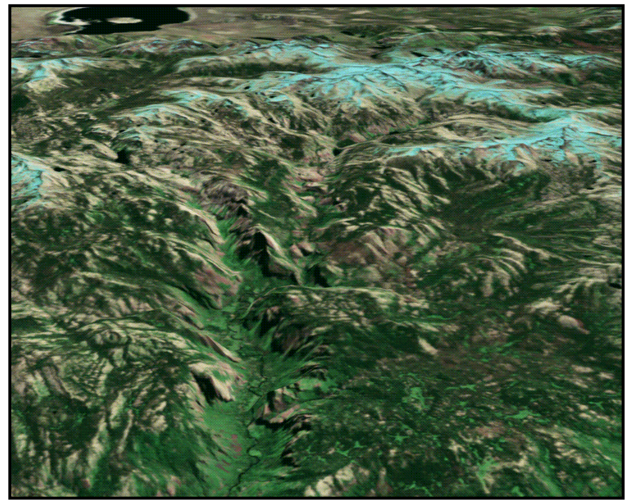

Figure 13: Yosemite Valley and Mono Lake

9 Conclusions

We have designed and built a system based on a new dynamic texture representation—the *clipmap*—a look-ahead cache configured to exploit insights about mipmap rendering, spatial coherence, and object to texture space coordinate mappings. This representation is simply parameterized and can be implemented in hardware via small modifications to existing mipmap rendering designs. The system features clipmap texturing hardware and a fast system-to- graphics bus allowing real-time updates of clipmap caches. Using this clipmapping system, users can render extremely high quality images in real-time while using relatively little texture memory. The hardware can further be generalized with low level software to virtualize even bigger textures than it is able to address directly. The generality of representing textures of arbitrary scale is both useful and necessary. This level of generality allows users greater freedom not only in rendering high quality images of enormous textures at real-time rates; it allows them to develop applications more easily and liberates geometric modeling and level of detail designs from texture placement concerns. These tasks are made significantly easier because texture issues such as which polygon gets mapped to which texture and how to control texture LOD are resolved as easily as for small textures.

In order that users could readily accept clipmaps into their applications, we designed a complete system-level solution to help developers to attain real-time image generation rates with minimal effort. In doing so, we have extended our clipmap representation to include a second level of texture caching in main memory. A software system has been built around this higher-level representation that virtualizes and manage everything about clipmaps—controlling data flow all the way from disk, through system memory, and into texture hardware in real-time. This software uses a pipelined architecture to determine where texture resolution will be needed and pre-stages data both from disk and to texture. In staging the texture data in this way, the software clipmap takes advantage of the fundamental limits of useful rendering resolution to optimize available transfer bandwidths and memory resources. Because this software has detailed knowledge of the underlying hardware, it is able to avoid performance bottlenecks caused by inadequate hardware configurations. Further, it is able to perform effective load control in all aspects of its operation by reverting to coarser resolution data present at lower-levels of the cache hierarchy. Finally, by doing all of these tasks in a comprehensive integrated framework, the software allows users think of clipmaps simply as big textures and to use them accordingly.

Virtualizing texture images in this way has global implications. As this technology migrates to lower cost implementations, it will allow people to utilize orders of magnitude more data than they previously could. In fact, we have begun to see significant third-party development in commercial tools to address the needs of those who create, rectify, register, color-correct, and manipulate large imagery. We also believe that the level of realism and scope in real-time visual simulations will significantly improve. Given a solution to the companion problem of real-time continuous terrain representation (which others have been working on), we can easily see that someday soon the general population of most industrialized nations will be exposed through clipmaps to environments and places that they otherwise could never experience.

In the future, we would like to extend the clipmap hardware to be more general and possibly deal with multiple cached regions. Also, the question of directly supported address precision vs. software virtualization should be evaluated in the context of any new implementation—while virtualization may always be necessary, more hardware precision means less software interaction. We would like to implement higher-order clipmaps with direct support for multiple resolution volumetric datasets of widely disparate resolutions as are typical in the geosciences. Further, we would like to extend the software system to integrate scheduling and real-time paging for large numbers of small mipmap textures with the currently support for huge clipmap textures.

Finally, we point out that the guiding insights about mipmap utilization that made the clipmap solution possible can be applied to related problems in computer graphics. Screen space resolution provides an upper bound to the amount of data that can significantly affect final renderer output. It seems possible to develop a system that stages geometric level-of-detail information for large databases similarly to the way clipmaps stage image data. If an adaptive rendering algorithm were defined to create continuous tessellations from partially specified geometric levels of detail, then the same look ahead cache notions could be used to stage geometry. At a system level this approach has predetermined throughputs and bandwidths that can be established to ensure both high fidelity and robust real-time behavior. In this way, the overall data flow of large systems can be easily sized and tuned for a wide range of applications.

In summary, we add that the notion of providing storage for all the texels that a single rendering requires, but no more than that , seems to us a good approach even if texture memory resources were closer to the required 12 petabytes. It reminds us of the insight about how much leather it takes to cover the earth, as presented in the Panchatantra[4] in prose:

Yes, all prosperities are his,
Whose heart is filled with mirth;
The feet, in leather sandals shod,
Travel a leather Earth.

10 References

[1]Williams, Lance. Pyramidal Parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings).* pages 1-11, 1983.

[2]Cosman, Michael. Global Terrain Texture: Lowering the Cost. *Proceedings of 1994 IMAGE VII Conference*. pages 53-64, 1994.

[3]Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics (SIGGRAPH '94 Proceedings)*. pages 381-394, 1994

[4] Ryder, A. W. translator, *Panchatantra*. Kashmir, 400 B.C.