The Clipmap: A Virtual Mipmap

Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones
Silicon Graphics Computer Systems

ABSTRACT

We describe the *clipmap*, a dynamic texture representation that efficiently caches textures of arbitrarily large size in a finite amount of physical memory for rendering at real-time rates. Further, we describe a software system for managing clipmaps that supports integration into demanding real-time applications. We show the scale and robustness of this integrated hardware/software architecture by reviewing an application virtualizing a 170 gigabyte texture at 60 Hertz. Finally, we suggest ways that other rendering systems may exploit the concepts underlying clipmaps to solve related problems.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; I.3.4 [Computer Graphics]: Graphics Utilities—Graphics Packages; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture.

Additional Keywords: clipmap, mipmap, texture, image exploitation, terrain visualization, load management, visual simulation.

1 INTRODUCTION

Textures add important visual information to graphical applications. Their scope, fidelity, and presentation directly affects the level of realism and immersion achievable for a given object or environment. From being able to discern each brush stroke in every mural in a virtual Vatican to counting tire tracks in the sand halfway around a simulated globe, the amount of visual information applications require is growing without bound. It is this enormous visual dynamic range that illuminates the limitations of current texture representations and defines our problem space.

Specifically, our representation addresses all of the issues relevant to the real-time rendering of the earth's surface as a single high resolution texture. Representing the earth with one meter texels requires a 40 million by 20 million texel texture and an overall square, power of two mipmap size of approximately 11 petabytes.

We identified six goals for an effective solution to this problem. First, the new texture system must support full speed rendering using a small subset of an arbitrarily large texture. Second, it must be possible to rapidly update this subset simultaneously with real-time rendering. Third, the texture technique must not force subdivision or other constraints onto geometric scene components. Fourth, load control must be robust and automatic to avoid distracting visual discontinuities under overload. Fifth, it must be possible for the representation to be seamlessly integrated into existing applications. Finally, the incremental implementation cost should be small relative to existing hardware.

Author Contacts: {cct|migdal|mtj}@sgi.com

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 manages the representation and interfaces with applications. This paper describes our clipmap implementation and reviews how well it meets the goals and challenges already outlined: §2 presents past approaches to managing large texture images, §3 describes exactly what a clipmap is and what it achieves, §4 explains how the clipmap is updated and addresses memory bandwidth issues, §5 shows how the clipmap representation is a modification of mipmap rendering, §6 describes how clipmaps are generalized to overcome hardware resource and precision limits, §7 discusses the higher-level software used to update clipmaps, manage system load, and deal with data on disk, §8 examines several applications of the system, and finally, §9 considers the success of our first implementation and suggests directions for further research.

2 PREVIOUS APPROACHES

The common method for dealing with large textures requires subdividing a huge texture image into small tiles of sizes directly supportable by typical graphics hardware. This approach provides good paging granularity for the system both from disk to main memory and from main memory to texture memory. In practice, however, this approach has several drawbacks. First, geometric primitives must not straddle texture-tile boundaries, forcing unwanted geometric subdivision. Second, texture border support is required for each level of each tile if correct sampling is to be performed at tile boundaries. Lastly, the level of detail mechanisms take place at the granularity of the tiles themselves—producing disconcerting visual pops of whole texture/geometry tiles. These limitations limit visual effectiveness and add an extra level of complexity to geometric modeling, morphing, and LOD definition—all of which must take the texture tile boundaries into account.

The higher-quality system of Sanz-Pastor and Barcena [4] blends multiple active textures of differing resolutions. Each of these levels roams based on its dynamic relationship to the eyepoint; polygons slide from one texture to another with textures closer to the eyepoint at higher-resolution. The drawback of such a system is that geometry is still tied directly to one of these textures so texture LOD decisions are made at the per-polygon rather than per-pixel. Developers must also obey a complex algorithm to subdivide geometry based on the boundaries between the different textures of different resolutions. Further, texture LOD choices made by the algorithm are based on coarse eyepoint range rather than the fine display space projection of texel area.

An advanced solution outlined by Cosman [1] offers per-pixel LOD selection in a static multiresolution environment. Although similar to our clipmap approach in some ways, Cosman does not address look-ahead caching, load control, or the virtualization of the algorithm beyond hardware limits.

Although these approaches have solved large scale problems, they do not appear generalizable to fully address all six goals of §1.

3 THE CLIPMAP REPRESENTATION

3.1 Observations about Mipmaps

The following review summarizes the key mipmap concepts on

which clipmaps are based. A *mipmap* as defined by Williams [6] is a collection of correlated images of increasingly reduced resolution arranged, in spirit, as a resolution pyramid. Starting with level 0, the largest and finest level, each lower level represents the image using half as many texels in each dimension: 1D = 1/2, 2D = 1/4, and 3D = 1/8 the texels. The 2D case is illustrated in Figure 1.

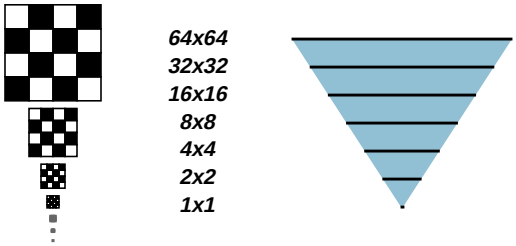

Figure 1: Mipmap Image Pyramid and Side-View Diagram

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 rendered pixel 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 display's pixel-to-texel mapping is closest to a 1:1 mapping, a level dictated in part by display resolution. The texels are then filtered to produce a single value for further processing.

Given this simple overview of mipmap processing, we now ask 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 display projection, such as differing triangle locations within a large database, so it may seem that all of the texels are potentially used if the universe of possible geometry is taken into consideration. Refer to Figure 2 for an illustration of the relationship between eyepoint position and texel access.

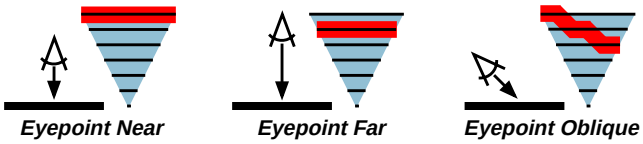

Figure 2: Texel Access within a Mipmap

Mipmaps are potentially fully accessed during rendering when their size in texels is less than twice the size of the maximum display extent in pixels. This observation derives from interactions between mipmap sample selection logic and finite display resolution. For example, when rendering a 32768^2 texel mipmap to a 1024^2 pixel display, the mipmap sample-selection logic will choose the texels that are closest to having a 1:1 mapping to pixel area. Thus it will use at most 1024^2 texels from a level before accessing an adjacent level. Implementations that blend samples from adjacent levels (as in *trilinear filtering*) potentially access twice as many texels in each dimension. For an example of this refer to the center diagram in Figure 2, where texels from level 1 are being fully used. If the eyepoint were just slightly closer, then samples would be blended from both level 0 and level 1, but 1024^2 texels from level 1 would still be accessible. The corresponding texels from level 0 needed to blend with these 1024^2 level 1 texels are distributed over a 2048^2 texel extent in level 0.

When the indexing arithmetic for very large mimpaps is analyzed, it becomes clear that the majority of the mipmap pyramid will not be used in the rendering of a single image no matter what geometry is rendered. The basis of our system is this realization that eyepoint and display resolution control 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. We can build hardware to exploit this fact by rendering from the minimal subset of the mipmap needed for each frame—a structure we term a clipmap.

3.2 The Anatomy of a Clipmap

A *clipmap* is an updatable representation of a partial mipmap, in which each level has been clipped to a specified maximum size. This parameterization results in an obelisk shape for clipmaps as opposed to the pyramid of mipmaps. It also defines the size of the texture memory cache needed to fully represent the texture hierarchy.

3.2.1 Defining Clipmap Region with ClipSize

ClipSize represents the limit, specified in texels, of texture cache extent for any single level of a clipmap texture. Each level of a normal mipmap is clipped to ClipSize if it would have been larger than ClipSize, as shown in Figure 3. All levels retain the logical size and render-time accessing arithmetic of the corresponding level of a full mipmap.

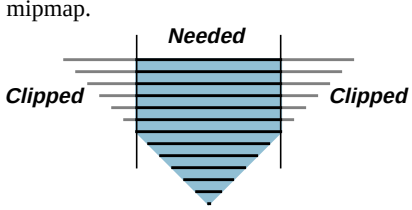

Figure 3: Clipmap Region within a Mipmap

Based on this mipmap subset representation, we further 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; only a ClipSize^2 subset is cached. These levels are the topmost levels in Figure 4. Below the Clipmap Stack is the *Clipmap Pyramid*, defined as the set of levels of sizes not greater than the ClipSize limit. These levels are completely contained in texture memory and are identical to the corresponding portions of a full mipmap.

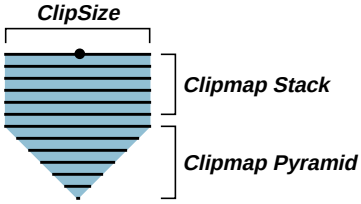

Figure 4: Clipmap Stack and Pyramid Levels

3.2.2 Defining Clipmap Contents with ClipCenter

Given the notion of clipping a mipmap to fit in a subset clipmap cache, we specify the data present in this cache by specifying a ClipCenter for each stack level. A *ClipCenter* is an arbitrary texture space coordinate that defines the center of a cached layer. By defining ClipSize and ClipCenter for each level, we precisely select the texture region being cached by the ClipStack levels of our representation.

One implementation is to specify the ClipCenter for stack level 0 and 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. This center is the dot indicated at the top of Figure 4. 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. The image in Figure 5 shows an orthogonal view of a polygon (viewed

from the “Eyepoint Near” 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.

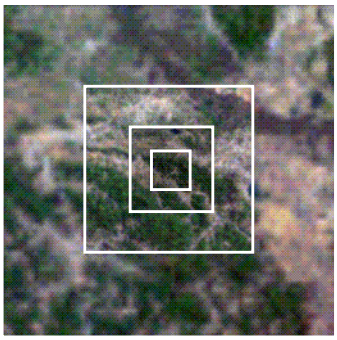

Figure 5: Rings of Texture Resolution

When the cache is insufficient, due either to an overly small ClipSize or a poor choice of ClipCenter, the best available lower resolution data from lower in the clipmap stack or pyramid is used. This “clip texture accesses to best available data” nature is a second reason why we chose the name clipmap for this approach.

In normal use, however, the ClipSize is set to equal or exceed the display size, in which case the rings of resolution shown in Figure 5 are large enough that, with a properly chosen ClipCenter, these rings form a superset of the needed precision. Thus, the subset nature of clipmaps does not limit the texture sample logic—every texel addressed is present in the clipped mipmap and the resulting image will be pixel-exact with one drawn using the full mipmap, meeting the first of the goals outlined for this texture system.

3.2.3 Invalid Border for Parallel Update

The previous subsection presents rendering from a Clipmap Stack level as a static subset of the complete mipmap level; it does not address the need to pre-page data needed for future frames concurrently with rendering. In order to provide storage for paged data and for load control, we introduce another parameter, the *InvalidBorder*, defined as a border of texels existing within each stack level that is InvalidBorder texels wide. Texels within the InvalidBorder are inactive and are not accessed by the texel-sample indexing logic. They provide a destination region into which higher-level software prefetches data. The size of the active portion of a clip map is simply ClipSize minus twice the InvalidBorder, known as the *EffectiveSize* of each stack level. The relationship and position of the InvalidBorder and EffectiveSize are illustrated in Figure 6.

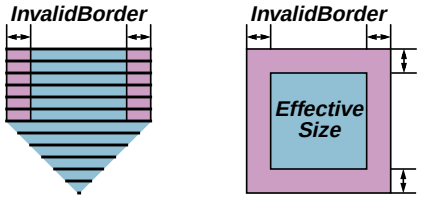

Figure 6: Clipmap with InvalidBorder and EffectiveSize

3.2.4 TextureOffset for Efficient Update

The final fundamental parameter of our low-level clipmap representation is the *TextureOffset*, the starting address used by the texture address generator to locate the logical center of a clipmap’s EffectiveSize within the ClipSize^2 memory region allocated for each Clipmap Stack level. The addressing logic uses the TextureOffset and modular addressing to roam efficiently through the level’s ClipSize^2 memory region as discussed in detail in §4, where the process of clipmap updating and rendering is presented. This offset is specified per level and affects addressing for the level exactly as a texture matrix translation with a wrap-style texture clamp mode.

3.3 Clipmap Storage Efficiency

Consider a 16 level 32768^2 clipmap to be rendered on a 1024^2 display. We begin our analysis with pixel-exact rendering. Given the display size, we know that the upper bound on texture usage from a single level is 2048^2, so we set the ClipSize to 2048. There will be four clipped levels forming our Clipmap Stack and 12 levels in the Clipmap Pyramid. The storage required for this is 2048^2 texels * 4 levels + 4/3 * 2048^2 texels for the pyramid = 42.7 MB at 2 bytes per texel. This perfect clipmap configuration requires only 42.7 MB of the full 2.8 GB present in the complete mipmap. A more typical configuration using a 1024 ClipSize will achieve attractive results using 1024^2*5 stack texels + 1024^2*4/3 pyramid texels = 12.7 MB. Finally, a 512 texel ClipSize yields reasonable results with 512^2*6 stack texels + 512^2*4/3 pyramid texels = 3.7 MB of storage.

In general, a 2^nx2^n clipmap with a ClipSize of 2^m requires only $4^m(n - m + 4/3) - 1/3$ texels of texture storage. 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. Note that total mipmap storage is exponential in n where clipmap storage is linear, underscoring the practical ability of clipmaps to handle much larger textures than mipmaps.

Comparative storage use 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, confirming the ability of clipmaps to address the problem identified in §1.

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

Table 1: Clipmap Storage Requirements

4 UPDATING CLIPMAPS

Clipmap caches are by intent just large enough to effectively cache the texels needed to render one view. It is therefore important to update the cache as the viewpoint changes, typically before each frame is rendered. This update is performed by considering the texture regions needed for each level as described in §3 and loading them into texture memory. Moreover, to take advantage of the significant frame-to-frame coherence in cache contents, we can reuse cached texels in subsequent frames. We define each level as an independently roaming 2D image cache that moves through the entire region of that level within the complete mipmap using toroidal addressing, a commonly used approach in image processing applications [5]. We then update each level incrementally by downloading the new data into the cache and letting this addressing scheme define the destination. The same logic is used when reading memory to render the texture. The four steps of this loading/addressing scheme are presented in Figure 7. The process begins with a 2D image centered in the cache as shown on the left. We then specify a new ClipCenter, in this case d texels above and to the right of the old center. Comparing the old and new image

regions, note that the central texels are unchanged *(SAME)* in each, so only the top *(T)*, corner *(C)*, and right *(R)* border areas are updated, as indicated in the third diagram.

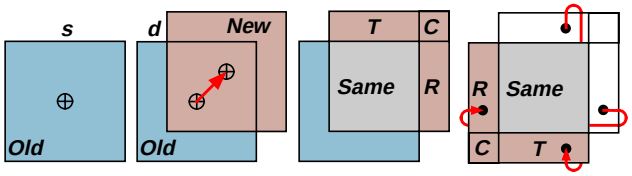

Figure 7: 2D Image Roam using Toroidal Addressing

Using toroidal addressing, new data at the top of the image is loaded at the bottom, data on the right is loaded at the left, and the upper right corner is loaded at the lower left. Implementation is easy: compute the virtual texel address by adding the ClipCenter to the texel address and then use the remainder modulo ClipSize as the physical address. The cache start address moves from ($s/2$, $s/2$) to ($s/2+d$, $s/2+d$), so the TextureOffset for this level is set to this new origin address, identifying the starting address of the cached image to the addressing unit.

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.

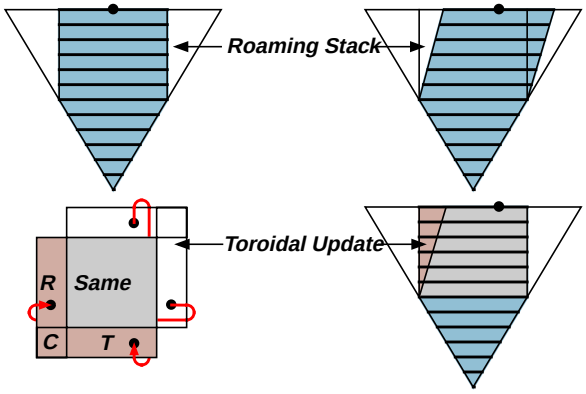

Figure 8: 2D Image Roam of Complete Stack

This side view of the toroidal indexing scheme shows how the entire set of stack levels appears to have moved when in fact all updates are performed in place as described previously. The overlap areas are unchanged and require no update. In practice, this area is relatively large so only minor paging at the edges is required to update a clipmap stack. 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, and less movement means less paging.

When movement of the TextureCenter is smaller than the InvalidBorder for a given level, then the update regions can be loaded while texture is being drawn with the previous center. This is true because the InvalidBorder guarantees those texels will not be used. This also allows large updates to be performed in an incremental manner across multiple frames before the ClipCenter is moved.

4.1 Update Bandwidth Considerations

In addition to rendering, implementations must also offer enough texture download bandwidth to meet the demands implied by Clipmap Stack depth, ClipSize, and the speed of eyepoint travel through texture space as defined by ClipCenter. An upper limit to cache update time is that time needed to replace all clipmap stack levels. In the case of the 32768^2 clipmap with a 1024 StackSize that our implementation is designed to handle, reloading the cache levels requires 10MBytes of texture paging. Given memory-to-graphics rates of 270MBytes per second, a flush requires $1/27^{th}$ second. Fortunately, the cache need not be completely flushed due to the texel reuse and incremental update allowed by toroidal addressing. With this efficiency, the ClipCenter can be moved 1000 texels on the highest level map in this configuration in less than $1/60^{th}$ of a second. This corresponds to an eyepoint speed of 134,160 miles per hour when 1 meter texels are used, a rate that easily exceeds our update goals.

5 RENDERING WITH CLIPMAPS

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

1. Determine the S, T coordinates for the current pixel. This is done exactly as addressing an equivalent mipmap.
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 has 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 LOD 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 is adjusted to consider the InvalidBorder designed to prevent accessing texels that are in the process of being asynchronously updated. Having calculated the LODmip and LODAvailable, we simply use the coarser of the two. We emphasize that it is possible to configure a system and application where the LODAvailable never limits the 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. These addresses are interpreted using modular addressing consistent with the way that §4 describes cache updates based on 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

Having defined the low-level architecture of the clipmap representation, we consider a key implementation issue—the numerical range and precision required throughout the texture system. This issue is central because it controls cost and complexity in an implementation, factors that we seek to minimize as goals of our development. Previous mipmap implementations have supported texture sizes as large as 2^8 to 2^{12} in each dimension, far less than the 2^{26} needed by a 1 meter per texel whole earth texture. Thus, having first solved the storage problem for huge texture images, we must now find an affordable way to address their compact representation.

The hardware environment of our first implementation [3] dictated that directly enlarging the numerical range and precision throughout the system to match the needs of 27-level and larger clipmaps was not practical. The impact would have included bus widths, gate counts, circuit complexity, logic delays, and packaging considerations; full-performance rendering would have been impossible. These issues, along with the practical matter of schedule, encouraged us to solve the 2^{26} clipmap problem within the 2^{15} precision supported in hardware.

Our approach to virtualizing clipmaps is based on the observation that while the polygons comprising a scene extend from the eyepoint to the horizon, each individual polygon only subtends a portion of this range. Thus, while a scene may need the full 27-level clipmap for proper texturing, individual polygons can be properly rendered with the 16-level hardware support so long as two conditions are met: the texture extent across each polygon fits within the hardware limit, and an appropriate adjustment is made as polygons are rendered to select the proper 16 levels from the full 27 level virtual clipmap.

We implement virtual clipmaps by adding two extensions to the clipmap system. First, we provide a fixed offset that is added to all texture addresses. This value is used to virtually locate a physical clipmap stack within a taller virtual clipmap stack, in essence making one of the virtual stack levels appear to be the top level to the hardware. Second, we add a scale and bias step to texture coordinate processing. This operation is equivalent to the memory address offset but modifies texture coordinates rather than texel memory addresses. In concert, these extensions provide 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 1 centimeter per texel resolution, which exceeds our stated goal.

This virtualization requires that the low-level clipmap algorithm be extended to address a sub-clipmap out of an arbitrary clipmap when the depth exceeds the directly supported precision. 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.

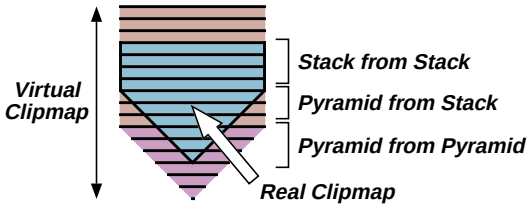

Figure 9: Virtual Addressing in Stack and Pyramid Levels

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 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 by 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 cases define the additional addressing hardware used to switch between the normal stack address processing (as described in §5), sub-addressing as defined above for cases 2 and 3, and normal mipmap addressing. With these addressing extensions, software can select sub-clipmaps and implement texture coordinate scale and bias.

Why add this additional complexity to an otherwise simple low-level implementation? Because there is seemingly no limit to desired texture size. While the size of directly supported clipmaps will likely grow in future implementations, so will the demand for larger textures. Since each extra level directly supported by hardware forces expensive precision-related system costs onto implementations, we believe that virtual clipmap support will always be useful.

7 DYNAMIC CLIPMAP MANAGEMENT

Having described the clipmap representation, the architecture, and the refinements needed to virtualize a clipmap beyond implementation limits, we now consider the remaining issue—keeping the clipmap updated as the eyepoint moves through texture space. This is a significant part of any implementation, since the tremendous efficiency of the clipmap representation reduces only the number of texels resident in texture memory at any one time, not the total number of texels in the mipmap. A series of images rendered interactively may visit the entire extent of the texture image over the course of time; an image which for a one meter per texel earth is an intimidating 10,923 TBytes of data.

Our goals for dynamic clipmap management are these: effective use of system memory as a second-level cache to buffer disk reads, efficient management of multiple disk drives of differing speeds and latencies as the primary source for texture images, automated load management of the clipmap update process to avoid distracting visual artifacts in cases of system bandwidth overload, support for high resolution inset areas (which implies a sparse tile cache on disk), and finally, complete integration of this higher-level clipmap support into software tools in order to provide developers the sense that clipmaps are as complete, automatic, and easy to use as standard mipmaps.

7.1 Main Memory as Second-Level Cache

To use main memory as a second level cache we must optimize throughput on two different paths: we need to optimize throughput from disk devices to memory and we need to optimize throughput from memory to the underlying hardware clipmap cache. In order to optimize utilization of system resources we use separate threads to manage all aspects of data flow. We need at least one process scheduling data flow, one process moving data from disk to memory, and one process incrementally feeding clipmap updates to the graphics subsystem. In the context of our high-level graphics software toolkit, IRIS Performer [2], this means that scheduling is done by 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 is created to manage file system activities.

The second level cache represents each level of the mipmap using an array of tiles. These tiles must be large enough to enable maximum throughput from disk, but small enough to keep overall paging latency low. We use sizes between 128^2 and 1024^2 depending on disk configurations. For this second level cache to operate as a real-time look-ahead cache, we load into parts of the cache that are not currently needed by the underlying hardware. Thus each cache level contains at least enough tiles to completely hold the underlying clipmap level that is currently resident while prefetching border tiles. With a ClipSize of 1024 and a tile size of 256^2, the cache must be configured to be at least 6 by 6 tiles or 1536 by 1536 texels, as shown in Figure 10.

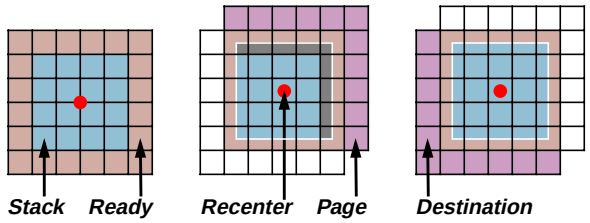

Figure 10: Tiled Main-Memory Image Cache

Due to memory alignment issues, low-level texture updates must be implemented as 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 pre-fetched the minimal single-tile border of image tiles, the "ready" area. The center illustration indicates that the ClipCenter (the 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 for this level. Multiple downloads are performed because it is infeasible to perform real-time memory to memory copies to realign memory when overall texture load rates are 270 MB per second. Once the clipmap layer is updated, the main memory tile-cache must be roamed up and to the right, by scheduling disk reads to retrieve the newly uncovered tiles, indicated as the "page" area in the diagram. 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 used in the clipmap itself—as in that case, unchanged data is never updated.

This main memory clipmap cache is very much a larger tiled version of the underlying hardware representation. It has a section of static pyramid levels and a section of cached roaming stack levels. The biggest difference being that an external tile border is used for paging in main memory and an internal InvalidBorder-sized region serves the same purpose within the clipmap. Each cache roams through its level of the full mipmap texture independently based on the level's ClipCenter. Each cache level performs look-ahead paging, requesting disk reads for image tiles that are likely to be loaded into the clipmap in the near future. Each level is responsible for managing the corresponding level of the underlying hardware clipmap implementation: recentering the underlying hardware level and incrementally downloading the relevant texture data from its tiled cache.

To configure the main memory cache system, developers specify information for each clipmap level. For stack 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 parameterization for load control.

7.2 Cache Centering Algorithms

In addition to the low and high level representations we have defined so far, there is still an important issue we have not yet discussed: how to decide which part of the image should be cached. We encourage developers to set the center directly in both the lower and higher-level software since the optimal place to center the cache is inherently application dependent. Some applications use narrow fields of view and thus look at data in the distance, which is where the ClipCenter should be located. Other applications, such as head-tracked ground-based virtual reality applications, 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. Our higher-level software provides utilities to automatically set the clipmap ClipCenter based on a variety of simple viewpoint projections and viewing frustum related intersection calculations as a convenience in those cases where default processing is appropriate.

7.3 Managing Filesystem Paging

The low-level clipmap representation provides fully deterministic performance because it is updated with whatever texels are required before rendering begins. Unfortunately, this is not true of the second-level cache in main memory due to the vagaries of filesystem access in terms of both bandwidth and latency. Compared to memory-to-graphics throughput of 200-450 MBytes/second, individual disk bandwidths of 3-15 MBytes per second are very slow. This could lead to a situation where speedy eyepoint motion exceeds the pace of second-level tile paging, causing the Effective-Size cache region to "stall" at the edge of a cache level awaiting data from disk before being recentered. We must anticipate this problem since we do not limit eyepoint speed, but we can avoid visual distraction should it occur.

The important realization is that the requested data is always present in the clipmap, albeit at a coarser resolution than desired, since the top-most of the pyramid levels in the clipmap caches the entire image. We use the MaxTextureLOD feature already present in the hardware to disable render access to a hardware stack level whenever updating a level requires unavailable data. The clipmap software system attempts to catch up to outstanding paging requests, and will re-enable a level by changing the MaxTexture-LOD as soon as the main memory texture tile cache is up to date. Thus MaxTextureLOD converts the stack levels into a resolution bellows, causing the clipmap to always use the best available data without ever waiting for tardy data. This makes it possible for clipmaps to allow arbitrary rates of eyepoint motion no matter what

filesystem configuration is used. In underpowered systems it is perfectly acceptable if the texture becomes slightly blurry when the ClipCenter is moved through the data faster than the tiles can be read. Recovery from this overload condition is automatic once the data is available; the system downloads the level and tells the hardware to incrementally fade that level back in by slowly changing the MaxTextureLOD over an interval of several frames.

To control latency in paging-limited cases, we provide a lightweight process to optimize the contents of the disk read queues. Without this the read queue could grow without bound when cache levels make requests that the filesystem cannot service, causing the system to get ever further behind. The queue manager process orders tile read requests by priorities which are determined based on the stack level issuing the request (with higher priorities for coarser data) and on the estimated time until a tile will be needed. The priorities of outstanding requests are asynchronously updated by the cache software managing each level every frame and the read manager thread sorts 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.

7.4 Managing Stack Level Updates

We now address load management issues in paging data from main memory to texture hardware. Since memory-to-graphics bandwidth is at a premium in immediate-mode graphics systems, it is important to understand and control its expenditure. A typical application might devote 75% of the throughput for sending geometric information to the graphics subsystem, leaving 25% of the 16.67ms (60 Hz) frame time for incremental texture paging, including both mipmaps and clipmaps. The time allocated for clipmap updates may be less than the time needed to fully update a clipmap because the center is moved a great distance requiring many texels to be updated or because the time allowed for updates is so little that even the smallest update may not be performed. Since unwavering 60 Hz update rates are a fundamental goal of our system, we must plan for and avoid these situations.

To control clipmap update duration we precisely estimate the time required to perform the downloads for a given frame. A table of the measured time for texture downloads of each size that could potentially be performed based on clipmap and system configuration is generated at system start-up. Then, as we schedule downloads, we start at the lowest (coarsest) cache level and work our way up the cache levels one by one. The total projected time is accumulated and updates are terminated when insufficient time remains to load a complete level. Coarse LODs are updated first, since they are both the most important and require the least bandwidth. Finer levels are disabled using MaxTextureLOD if there is insufficient time for the required updates.

We also expose the MaxTextureLOD parameter for explicit coarse-grain adjustments to required download bandwidth. Since each higher stack level requires four times the paging of 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, it 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 to 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 row of Figure 11, where rendering with a larger InvalidBorder is seen to scale the Stack Levels closer to the ClipCenter. Only the stack levels are modified by the InvalidBorder, as indicated by the outer region in the left diagram.

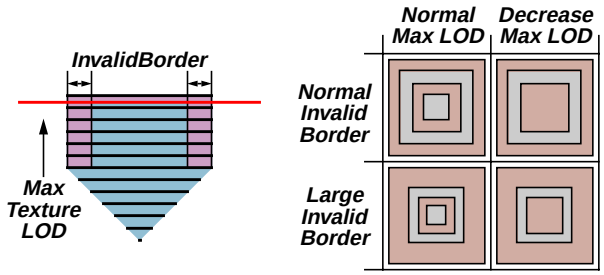

Figure 11: Visual Effect of Load Control

The load control mechanisms presented here have proven to be effective, allowing existing applications to integrate clipmaps virtualizing terabytes of filesystem backing store, maintaining constant 60 Hz real-time rendering rates irrespective of disk speeds and latencies.

7.5 High-Resolution Insets

High-resolution insets are regions of extra-high texture resolution embedded into a larger, lower-resolution texture image. They are important since applications often have areas within their database where higher-fidelity is needed. An 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 require more moderate texture resolution.

Our implementation supports insets as a side effect of the integrated filesystem paging load control that constantly sorts tile read requests. To implement insets, developers specify sparse tile arrays in the filesystem for each clip level. Sparse tile arrays are those where some or nearly all tiles are undefined, resulting in “islands” of resolution in various portions of an otherwise empty tile array (near airports, for the example above). In this situation the higher-level software uses the inset data when it can—skipping the empty regions automatically using the TextureMaxLOD load-control mechanism as it discovers that nonexistent tiles are perpetually late in arriving from the filesystem. The load control algorithm naturally falls 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 at least ClipSize texels wide using data magnified from the next lower level 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. Providing this border ensures that the load-management algorithm will enable the entire level when 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 for real and virtual clipmaps within the InfiniteReality [3] graphics subsystem;

special OpenGL clipmap control and virtualization extensions; and support within the IRIS Performer [2] 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, developers find the results of using the clipmap approach to be excellent. Real-time graphics applications can now use textures of any desired precision at full speed—examples include planet-wide textures from satellite data, country and continent scale visual simulation applications with centimeter scale insets, architectural walkthrough applications showing murals with minute brush strokes visible at close inspection, and numerous advanced uses in government.

The image in Figure 12 shows an overhead view of a 8192^2 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.

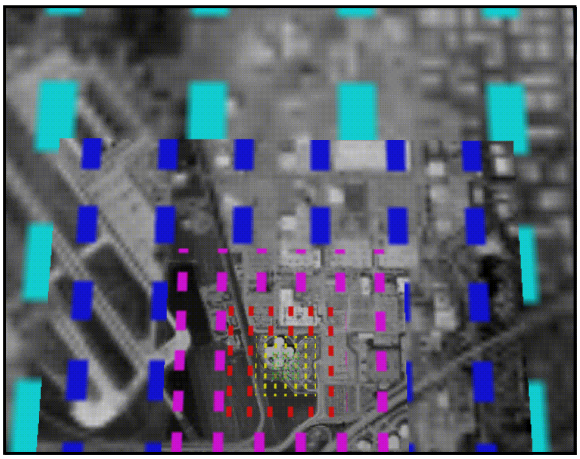

Figure 12: Concentric Resolution Bands

The image in Figure 13 shows a very small portion of a 25m per texel clipmap of the entire United States—the area shown here is the southern half of the Yosemite National Park. That our approach makes this single 170 GByte texture displayable at 60 Hertz with an approximately 16 MByte clipmap cache is impressive; equally so is the fact that the application being used need not know about clipmaps—all clipmap definition, updating, load-management and default ClipCenter selection happens automatically within IRIS Performer.

9 CONCLUSIONS

We have developed a new 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. It has been implemented in a system featuring a fast system-to-graphics interface allowing real-time update of clipmap caches. This system renders high quality images in real-time from very large textures using relatively little texture memory. The hardware supports virtualization for textures larger than it can address directly. This approach is not only important for representing textures of arbitrary scale. Equally important is that it also liberates geometric modeling and level of detail decisions from texture management concerns.

The guiding insights about mipmap utilization that made the clipmap solution possible can be applied to related problems in computer graphics. Display resolution provides an upper bound to the amount of data needed in any rendering. 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.

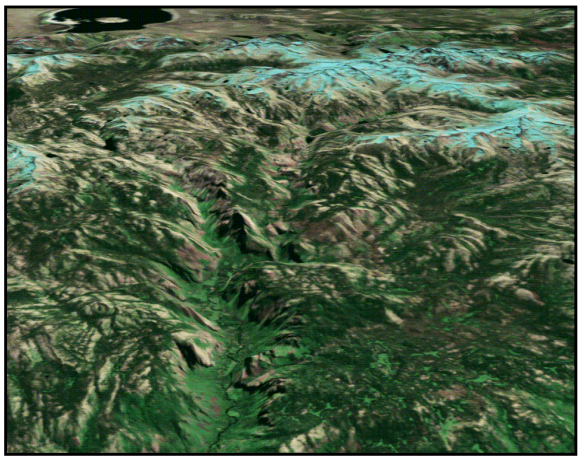

Figure 13: Yosemite Valley and Mono Lake

Since the original development of clipmaps and their hardware and software implementations, we have explored several extensions including 3D clipmaps, texture download optimizations, and new inter-level texture blend modes. This work leads us to believe that there is considerably more to be discovered. We have seen realism in real-time visual simulation revolutionized in the wake of the introduction of clipmaps and we eagerly anticipate new ways in which clipmaps can have this effect in other application areas.

Acknowledgments

We would like to recognize Jim Foran for his original idea, Don Hatch and Tom McReynolds of IRIS Performer, Mark Peercy of OpenGL and the technical marketing team for their contribution.

References

[1] Cosman, Michael. Global Terrain Texture: Lowering the Cost. In Eric G. Monroe, editor, *Proceedings of 1994 IMAGE VII Conference*, pages 53-64. The IMAGE Society, 1994.

[2] Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*. Annual Conference Series, pages 381-394. ACM SIGGRAPH, Addison Wesley, July 1994. ISBN 0-89791-667-0.

[3] Montrym, John S, Daniel R Baum, David L Dignam and Christopher J Migdal. InfiniteReality: A Real-Time Graphics System. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*. Annual Conference Series, pages 293-301. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[4] Sanz-Pastor, Nacho and Luis Barcena. Computer Arts & Development, Madrid, Spain. Private communication.

[5] Walker, Chris, Nancy Cam, Jon Brandt and Phil Keslin. Image Vision Library 3.0 Programming Guide. Silicon Graphics Computer Systems, 1996.

[6] Williams, Lance. Pyramidal Parametrics. In Peter Tanner, editor, *Computer Graphics (SIGGRAPH 83 Conference Proceedings)*, volume 17, pages 1-11. ACM SIGGRAPH, July 1983. ISBN 0-89791-109-1.