# Using Texture Mapping with Mipmapping to Render a VLSI Layout

Jeff Solomon
Computer Systems Lab
Stanford University
jsolomon@stanford.edu

Mark Horowitz
Computer Systems Lab
Stanford University
horowitz@ee.stanford.edu

## ABSTRACT

This paper presents a method of using texture mapping with mipmapping to render a VLSI layout. Texture mapping is used to save already rasterized areas of the layout from frame to frame, and to take advantage of any hardware accelerated capabilities of the host platform. Mipmapping is used to select which textures to display so that the amount of information sent to the display is bounded, and the image rendered on the display is filtered correctly. Additionally, two caching schemes are employed. The first, used to bound memory consumption, is a general purpose cache that holds textures spatially close to the user's current viewpoint. The second, used to speed up the rendering process, is a cache of heavily used sub-designs that are precomputed so rasterization on the fly is not necessary.

An experimental implementation shows that real-time navigation can be achieved on arbitrarily large designs. Results also show how this technique ensures that image quality does not degrade as the number of polygons drawn increases, avoiding the aliasing artifacts common in other layout systems.

## Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: [Computer-Aided Design]; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*

## Keywords

texture mapping, mipmapping, VLSI layout editor

## 1. INTRODUCTION

The field of computer graphics has produced two very well known techniques for the display and manipulation of images: *texture mapping* and *mipmapping* [7]. These techniques were developed primarily as a way of efficiently displaying the same image over and over, independent of magnification. Since many graphics intensive applications have a need for such a feature, almost all specialized graphics platforms have dedicated hardware to accelerate this function.



The benefit of using texture mapping and mipmapping becomes clear when considering the size of modern VLSI layouts. The number of rectangles[1] in modern layouts can run into the tens or hundreds of millions. Displaying each of those rectangles at once can take an unacceptably long time if done naively. In addition, when viewed at low magnification, most of the rectangles can be smaller than a single pixel in one or both dimensions; drawing them without proper filtering, therefore, will produce noticable aliasing artifacts.

Thus, the goal of this work is to allow real-time navigation of a VLSI layout independent of the size of the design, and to create an accurate representation of the design at any magnification. Additionally, the resources required to implement a usable solution should be bounded.

The format of the rest of the paper is as follows. Section 2 reviews texture mapping and mipmapping, laying the foundation for the techniques used throughout the paper. Section 3 discusses how an IC layout can be treated like a regular image and how previous techniques have been developed to handle such images. Section 4 introduces the tiled texture pyramid which allows for the efficient representation of an arbitrarily large IC layout as a mipmap. Sections 5 and 6 discuss how the texture tiles are created and managed, and Section 7 explains how heavily used sub designs are chosen and precomputed to further speed up the rendering process. Section 8 discusses how this architecture is amenable to multi-threading, which also speeds the rendering process and improves tool responsiveness. Finally, Section 9 discusses the experimental implementation and gives results showing the effectiveness of this approach.

## 2. TEXTURE MAPPING AND MIPMAPPING

A *texture* is a static image comprised of elements called *texels*. The values contained in each texel can be any type of visual information such as intensity, transparency or, most commonly, red green blue (RGB) triplets. A texel in a texture is distinguished from a pixel on the screen in that a texel can represent more or less area than a pixel depending on the texture's final scaled size on the screen.

*Texture mapping*, in its simplest form, is a way to apply a texture as a decal to a polygon. *Mipmapping* is a way to specify down-sampled views of a texture that are used to represent the texture when it is scaled in the scene. Each down-sampled representation of a texture is $1/4$ the size of the texture it was sampled from. When

[1]The overwhelming majority of polygons in VLSI layouts are rectangles. The rest of the paper will refer to layouts made up solely of rectangles, although the techniques described could be readily adapted to any type of polygon.

abstractly viewed with each mipmap level stacked on top of each other, the whole data structure is known as a *mipmap pyramid* or simply as a *mipmap*.

There are two primary advantages to using a mipmapped representation. First, since the down-sampled textures are precomputed, more care can be taken to produce an accurate representation than if the filtering were done on the fly. Secondly, the down-sampled textures are by definition smaller than the base texture, thus consuming less graphics bandwidth to display. The main disadvantage of mipmapping is the increased memory footprint from storing the levels of the mipmap pyramid.

When rendering a polygon with a mipmapped texture, the final pixel values of the polygon are computed by determining the *level of detail* (LOD) on a per pixel basis. The LOD is the ratio of the pixel area to the area of texture to be drawn in the texture's base units. In the general case, the LOD computation is needed on a per pixel basis in both the $x$ and $y$ dimensions because the polygon to which the texture is mapped could have an arbitrarily oblique orientation in the scene. In the specific case of rendering a 2D image parallel to the screen, as is the case with IC layouts, the same LOD applies to all pixels.

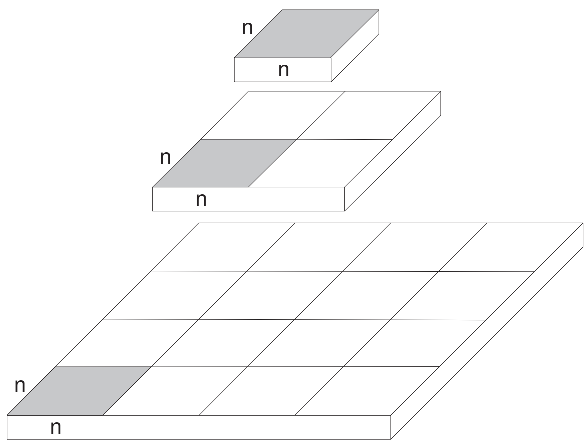



**Figure 1: A Tiled Texture Pyramid.**

# 3. AN IC LAYOUT AS AN IMAGE MIPMAP

The concepts of texture mapping and mipmapping can be applied directly to IC layouts if the IC layout is first converted into an image. This is done in the following way. The width and height of the image is directly computed from the size of the layout and the underlying grid resolution. For example, a $1mm \times 1mm$ layout with a grid resolution of $1\mu m$ would have a base image width and height of 1,000. Once the base image width and height have been determined, the next step is to rasterize the rectangles into the image. This process is straightforward, because the grid resolution was chosen such that all rectangles have integer coordinates. The rectangles in the layout can be rasterized into the base image following any convention of layer stacking order or transparency. Once the base layout has been rasterized, the higher mipmap levels are generated by filtering down-sampled versions of the base image.

To correctly down-sample an image, an ideal low-pass filter (a $sinc$ filter) can be applied that removes the appopriate high frequencies. Usually though, an approximation of a $sinc$ filter, the $box$ filter, is used for such operations because it requires much less computation, yet still provides acceptable quality. The definition of the $box$ filter used in this paper is to average the RGB values in a $2 \times 2$ square of texels to obtain the new down-sampled value.

After the mipmap has been created for an IC layout, it can be viewed as any other image. However, most, if not all, hardware graphics implementations impose limits on the dimensions of the base image used for texture mapping and mipmapping. Currently, these limits range from 256 to 16384 texels on a side, creating a severe restriction on the size of the IC layout that can be viewed using standard mipmap techniques.

Tanner presents the *clipmap* [6] as a solution for viewing arbitrarily large images. He observed that although the mipmap pyramid may be huge, the portion that is currently visible at any one time is bounded and small because of mipmapping and a fixed screen resolution. The example used by Tanner was a 20 million by 20 million texture that represents an image of the Earth at one meter resolution. Using specially modified hardware and optimized disk caching techniques, the clipmap implementation is able to render the texture of the Earth at any magnification in real-time.

The size of a modern microprocessor, when viewed as an image, is comparable to the example used by Tanner. Consider a modern microprocessor that is $20mm \times 20mm$ on a side with a grid resolution of $.01\mu m$. This would lead to a base image of width and height of 2 million texels. If each texel were 16 bits, the size of just the base image would be 64 terabytes. While it would be possible to build a clipmap solution to this problem, the overhead of hardware and disk usage would make it all but impractical for general use. However, the synthetic nature of IC layouts obviates the need to think of layouts solely as images. Notice that the canonical form of an IC layout, such as a *corner-stitched data structure* [4], is typically very small when compared to its size as a fully expanded image. In corner-stitching, and other data structures like it, the total memory cost is strongly related to the number of rectangles in the design. This is in contrast to an image where the memory cost is only related to the dimensions of its bounding box. These memory costs would only be equal when the number of rectangles in the design database were roughly on the order of the number of pixels in the bounding box. This can only occur when the entire bounding box is covered by small or unit dimension rectangles which is a property not observed in any useful VLSI layouts.

Given this substantially reduced memory requirement, the implementation described here is able to keep the entire layout database in memory which eliminates the need for any disk accesses.

The next section explains how only the visible portions of an immense mipmap pyramid are created, giving the user the illusion that the entire image has been expanded.

# 4. TILED TEXTURE PYRAMID

So far, each level of a mipmap pyramid has been thought of as one texture. For large IC layouts, the size of all but the highest levels in the pyramid could easily eclipse not only the hardware limits of the host platform but also the size of main memory.

To circumvent this limit, the concept of a *tiled texture pyramid* is introduced. A tiled texture pyramid is distinguished from a mipmap pyramid in that each pyramid level is an array of texture tiles where the size of each of the tiles, in texels, is fixed. An example tiled texture pyramid is shown in Figure 1.

The important distinction in the levels is that tiles in the upper levels represent more layout area than those in lower levels even though they physically consume the same amount of memory.

Using this tiled representation is important for three reasons. First, the tile size is chosen to meet the hardware limits of the host platform. This circumvents the restrictions that have been discussed earlier. Second, a tiled representation allows for a simple way to create only the portions of a mipmap level that are needed at any one time. Tanner presents a more complex approach that is more efficient in managing massive mipmaps in the general case; in the special case of IC layouts, however, a tiled pyramid approach has no significant disadvantages[2]. Third, a tiled texture representation lends itself very well to a multi-threaded implementation. This will be discussed further in Section 8.

With the concepts of texture mapping, mipmapping, and a tiled texture pyramid in hand, the following rendering strategy for IC layouts emerges:

1. Read in the entire layout database. This step is similar to most IC layout tools.
2. Perform an LOD calculation to determine which pyramid level is most appropriate to view given the current viewpoint.
3. Lazily create only the tiles of the tiled texture pyramid that are visible on the screen.
4. Draw the layout.
5. As the viewpoint changes, go back to step 2.

Given this rendering strategy, the next sections explain how to create and manage the tiles of the pyramid efficiently.

## 5. CREATING TEXTURE TILES

The creation of a texture tile involves computing the values of the texels for that tile. Section 3 showed how the texel values could be created by first rasterizing at the base level, and then down-sampling. While this algorithm produces very accurate texel values, consider the computation and memory cost for using this approach to create a tile high up in the pyramid. In the pathological case of the top-most tile, the memory cost would be equal to the base area of the design, and the computation cost to filter that potentially enormous area down to a single tile would be equally prohibitive. The focus of the subsequent sections will be on how the tiles high up in the pyramid can be computed directly from the design data, obviating the need to rasterize at the base level and down-sample.

### 5.1 Coverage

The simplest rasterization case is shown in Figure 2(a), representing two wires running horizontally, a wire running vertically, and a via. The coordinates of the rectangles coincide with the texel boundaries. This case corresponds to a texture tile at the base level of the pyramid. Here, the grid resolution of the layout and the size of the texels are equal such that all rectangles will fall on integer boundaries. Rasterization is simple: either a rectangle covers a texel completely or not at all. The final color of the texel is either the background color, the color of a single layer, or the blended color of two or more layers depending on the rendering style.

Now consider Figure 2(b). It shows an assortment of wires and vias that fall arbitrarily on the texel grid. This case occurs whenever the scaled coordinates[3] of the rectangles have non-integer values. This only occurs on pyramid levels other than the base level.

[2]The only substantive disadvantage is that the tile borders require special attention, but this is more of an implementation issue than a performance bottleneck.

[3]The scale factor is simply given by $2^{1/level}$ where the lowest level is considered level zero.

In situations like this, the rasterization process is not as simple. The computation of the texel's final color must take into account the fact that the rectangle only partially covers it. To do this, the amount of *coverage* of a given rectangle over a given texel is used. The idea of coverage is the same as the well-known graphics idiom *alpha* ($\alpha$) which represents a color's opacity. One can think of a color that does not completely "fill" a texel as partially covering it (coverage), or completely covering it at some opacity (alpha) less than one, they are the same thing.

### 5.2 Coverage Maps

An important structure that uses coverage information is a *coverage map*. Figure 2(c) shows the coverage map derived from the darkest horizontal wires shown in Figure 2(b). A coverage map holds coverage information for one layer over the equivalent layout area of the corresponding texture. There is a one to one correspondence between a texel in a texture and a coverage map element. The coverage map in this figure is encoded with gray scale values, darker grays indicate more coverage, lighter indicate less.

The construction of a coverage map consists of rasterizing the rectangles from a single layer and adding their coverage values to the map. The assumption is made that rectangles from the same layer do not overlap so the coverage values add, saturating to one.

### 5.3 Computing Texture Texels

The coverage map defined in the previous section does not contain color information, nor can it be used to display information directly on the screen. It can be used, however, to facilitate the computation of the texels for a given texture tile. First, the target texels are zeroed, and a scratch coverage map is allocated. Then, for each layer to be composited, a coverage map is created, and the information in the coverage map is used to blend that layer's color into the final texture.

The speed of tile creation depends on both the number of rectangles contained in a tile and the number of layers to composite. If the number of rectangles to process is small, then the time to create a tile will be dominated by the time to process the layers. Otherwise, the time to rasterize the rectangles into the coverage maps will dominate. Tiles high up in the pyramid tend to fall into the latter case since they contain the most rectangles, while lower pyramid tiles tend to have their computation time dominated by layer processing.

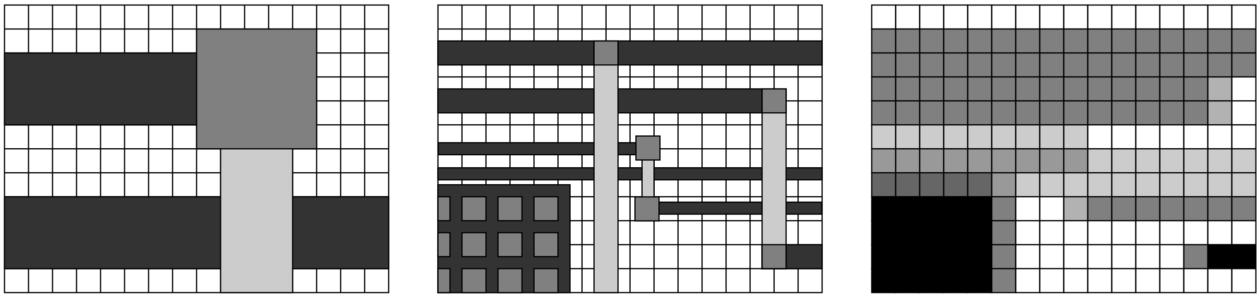
## 6. MANAGING TEXTURE TILES

### 6.1 Using Spatial Locality

As a user changes viewpoints, texture tiles are created and rendered on the display. If left unchecked, the number of texture tiles created and, correspondingly, the amount of memory consumed could grow to the full size of the pyramid. As was noted previously, the memory footprint of a full tiled texture pyramid for a large size IC layout could be on the order of terabytes. Clearly, it is not feasible to allocate new memory blindly every time a new texture tile is created.

As a solution to this problem, a fixed size texture tile cache is created. As texture tiles are computed, they are placed into the texture tile cache. When a tile is created and there is no room left in the cache, a suitable tile is found to replace. Due to the spatial locality of viewing IC layouts, an LRU policy has good performance for this cache.

The size of the cache should be set large enough to avoid capacity conflicts as much as possible and small enough such that the entire cache can fit into the main memory of the host platform. If



(a) Aligned rectangles.

(b) Unaligned rectangles.

(c) A coverage map.

**Figure 2: Figure 2(a) shows the simplest rasterization case. Figure 2(b) shows a more complex case where the rectangles are unaligned with the texel boundaries. Figure 2(c) shows the coverage map generated from the darkest rectangles of Figure 2(b). Darker areas represent more coverage, while lighter areas represent less.**

the cache is so large that the application memory footprint does not fit into the host platform's main memory, then performance is degraded. The time spent recomputing a texture tile is generally less than the time needed to swap a computed texture tile from disk.

## 6.2 Precomputing the Top of the Pyramid

Showing a design at full screen view is a very common operation in layout editors, and should always be fast. However, the very top of the texture pyramid is the most time-consuming to compute. Given this, the fact that the memory cost of these tiles is small, and that it is beneficial to always have these tiles available for display, it is advantageous to precompute a small number of tiles that make up the upper levels of the pyramid.

First, a decision is made on how many of the upper levels to precompute. A good heuristic is to choose the level that coincides with a full screen view when the application window is the same size as the screen. Depending on the screen resolution, this can be anywhere from three to five levels. Next, memory separate from the general texture tile cache is allocated specifically for the tiles in this upper section, guaranteeing that they can never be evicted from the general tile cache. Finally, layer coverage information is stored in uncomposited form so the tiles can be recreated quickly during a global appearance change.

A frequent operation used in layout editors involves hiding layers, changing the order in which layers are displayed, or displaying layers with different colors or transparencies. Changes like this require all texture tiles to be recreated even though the design data has not changed at all. To avoid this problem, coverage map information is kept for the lowest precomputed level, so that the tiles can be recreated quickly if a global appearance change is made.

To create the precomputed portion, coverage maps are made as they were defined in Section 5.2. Next, the bottom-most level is created in the way described in Section 5.3. Lastly, the higher-level pyramid tiles are created by using the standard box filtering technique described in Section 3.

Now when a global change to the view occurs, the computation required to recreate the top portion of the pyramid is limited only to recompositing the coverage maps. This time is constant and not dependent on the size of the design or the number of layout objects contained within.

# 7. USING HIERARCHY

Another unique property of IC layouts is the explicit reuse of sub-blocks in the form of instantiated hierarchy. The majority of the time spent in creating a texture tile comes from visiting each rectangle in that tile. This time can be reduced if the instantiated sub-blocks are pre-rasterized such that iterating over their rectangles is unnecessary.

A hierarchy cache is a block of preallocated memory that is used to store pre-rasterized versions of heavily used sub-designs. Each pre-rasterized sub-design will be a complete mipmap for that sub-design. These mipmaps do not need to be tiled texture pyramids, because their data will never be used directly by the graphics implementation. It will merely be copied into the main design's tiled texture pyramid as needed.

How does one select which sub-designs out of the whole design to pre-rasterize? The cells with the highest number of instantiations are preferred since this maximizes the utility of the cache. In the case that two sub-designs are instantiated the same number of times, the larger sub-design is taken because it will cover more area in the base layout. Once this ranking has been established, a method is needed to select which of the sub-designs to pre-rasterize since the hierarchy cache is of finite size. The steps are:

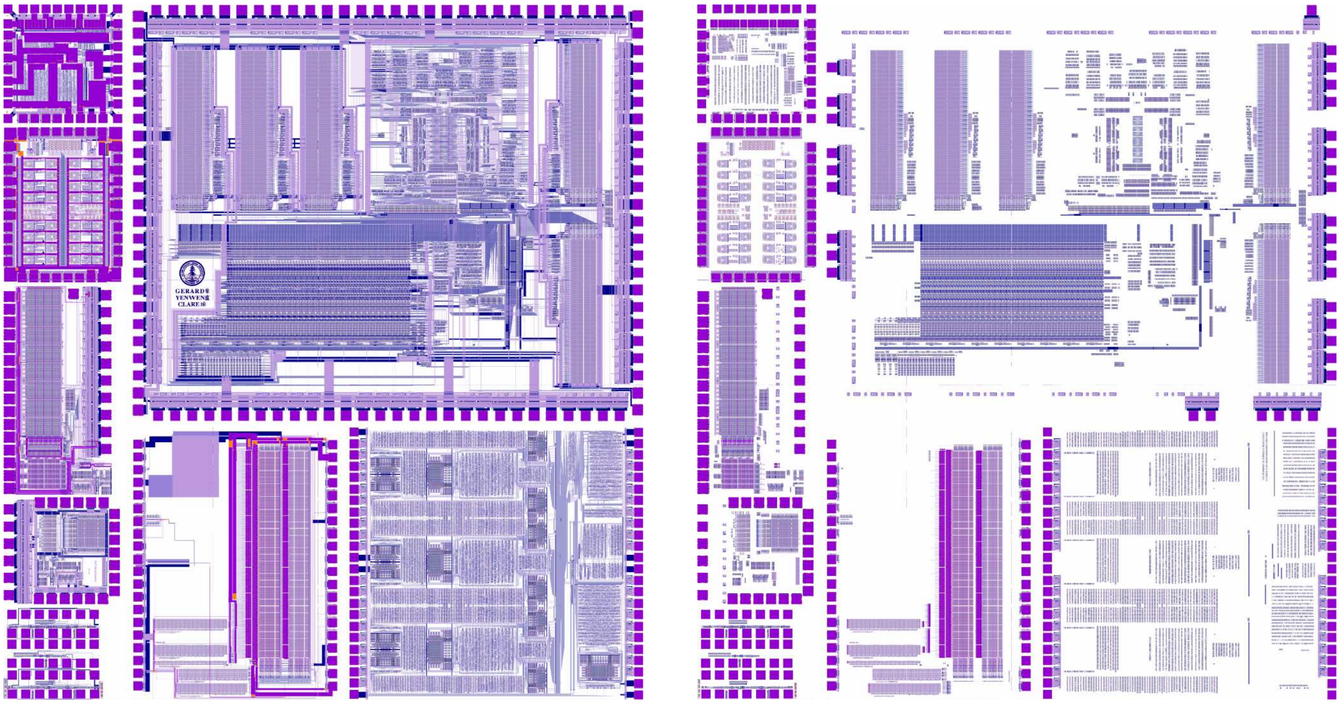
1. Compute the instance count for each sub-design and rank them from highest to lowest.

2. Walk down this ranked list and mark sub-designs as being pre-rasterized, subtracting the memory cost of pre-rasterizing them from the available size of the hierarchy cache. Continue walking down the ranked list until the available hierarchy cache memory is depleted. Note that sub-designs are merely marked for pre-rasterization; no computation is actually done at this point.

3. Visit all the sub-designs that were marked as being pre-rasterized and determine whether all instances of its parent are also going to be pre-rasterized. If so, there is no need to pre-rasterize this design since its parent will be pre-rasterized. In this case, mark the design as not being pre-rasterized and return its memory allocation to the available hierarchy cache pool.

4. Go back to step 2 and continue to select the highest ranked sub-designs until the hierarchy cache is depleted again or terminated when an iteration yields no change.

5. Finally, create mipmaps for all sub-designs that were marked for pre-rasterization.



(a) The SU_Block Design.

(b) SU_Block Design's Hierarchy Data.

**Figure 3: Figure 3(a) shows the SU_Block design in its entirety. Figure 3(b) is also the SU_Block design but only the contents of the hierarchy cache have been rendered. Approximately 52% of the total area is covered by data from the hierarchy cache.**

This process pushes the selection of sub-designs as far up the hierarchy as the size of the hierarchy cache will allow. The case is allowed where the main design can fit into the hierarchy cache. This occurs when the size of the design is small and simply means that the entire layout will be pre-rasterized.

The format of the pre-rasterized hierarchy data is the same as the coverage map information described in Section 5.2. The data needs to be kept as coverage map information so that correct compositing can be done when the texture tiles are created.

# 8. USING MULTI-THREADING

Regardless of how well the algorithms described in the previous sections are implemented, there will still be a finite amount of time to create the necessary texture tiles. This time delay can cause a stutter in the responsiveness of an end application. To mitigate this delay, a multi-threaded approach is taken. One thread renders the texture tiles on the display while one or more threads are tasked with creating the tiles. Since a tiled approach was chosen, and each texture tile is completely independent of any other, $N$ "creator" threads can be used to achieve at most $N$ speedup, if $N$ processors are available on the host platform.

In the case where the drawing thread does not have all of the texture tiles available to it, it can look farther up in the pyramid for another texture tile that covers the same area. A tile found higher up in the pyramid will be a coarser view of the desired area, but it is better to draw a fuzzier view of the layout than nothing at all. Note that since the top part of the pyramid is pre-computed, *some* coarser view of the entire layout will always be available for use.

The result of this multi-threaded approach is that, as the viewpoint changes very quickly, the layout may become fuzzy because the necessary texture tiles have not yet been created. As the viewpoint remains constant, and the necessary tiles are created, the image refines itself.

# 9. IMPLEMENTATION AND RESULTS

The system described was implemented by modifying the Magic Layout System [5]. The OpenGL [3] graphics library was used to render the designs. For the rest of the the paper, this implementation will be called "glLayoutView."

Experiments were run to compare the performance of glLayoutView, the *Magic* Layout System (version 6.5a), and two popular commercial tools, *A* and *B*. Both commercial tools are from major companies in the VLSI design industry. Tool A is primarily used as a layout editor, while tool B is intended for viewing large designs.

The test platform was a Sun Microsystems Ultra 60 workstation with two 450 MHz UltraSparcII processors, 2GBs of main memory, and an Expert3D [1] graphics card. The operating system was Solaris 2.7, and the version of OpenGL was 1.2.1.

Two designs were compared: a design with substantial hierarchy, *SU_Block*, and a design with no hierarchy, *Flash*. Table 1 gives the statistics for the two designs.

The Flash design is a flat layout containing the three metal and two contact layers of the *Flash MAGIC Chip* [2]. SU_Block is a collection of custom designs containing fifteen assorted metal, active, and contact layers with three times the number of total rectangles, but only $1/6$ the number of unique rectangles as the Flash design. The ratio of unique to total rectangles reflects how much hierarchy exists in the design. The last column shows the dimensions of the designs in grid units at the base level.

| | Total Rectangles | Unique Rectangles | Design Size |
|---|---|---|---|
| Flash | 4,893,834 | 4,893,834 | 156K × 156K |
| SU_Block | 14,855,372 | 833,820 | 50K × 54K |

**Table 1: Statistics for the two designs used in performance comparisons. If the designs were treated as regular images, as described in Section 3, the size of the image files (assuming 24 bit color), for only the base level, would be 73GB for Flash and 7.5GB for SU_Block.**

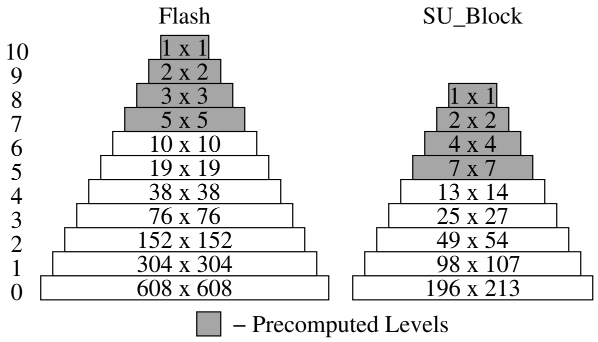



**Figure 4: Texture tile pyramid dimensions for Flash and SU_Block. Four levels of the pyramid were precomputed in each design.**

For all tests, glLayoutView was configured with $256 \times 256$ texel tiles, a 64MB texture tile cache and a 64MB hierarchy cache. Figure 4 shows the tile pyramid dimensions for the two designs, with precomputed levels shown in gray. Notice how the dimensions of level zero, when multiplied by 256, are approximately the values in the column labeled "Design Size" of Table 1.

Both of the commercial tools have the ability to draw a scaled rectangle conditionally based on a user-defined threshold. This feature was turned off in both tools to make an equal comparison, since glLayoutView draws all rectangles regardless of their scaled size.

## 9.1 Comparing Static Rendering Performance

The first performance test compares the tools rendering the entire design once. For each design and tool, redraw time is recorded for a window sized $1280 \times 1024$ pixels. For glLayoutView, this best corresponds to pyramid level five (49 tiles) in SU_Block, and level seven (25 tiles) in the Flash design. The results are shown in Table 2(a).

Because glLayoutView is multi-threaded, the actual redraw time of the screen is fixed and dependent on the graphics capabilities of the host platform. For an Expert3D graphics card, this time is between 0.01 and 0.25 seconds depending on the number of tiles to draw and whether or not they have been loaded into the graphics accelerator. In order to provide an equal comparison, the numbers quoted for glLayoutView are the times to create and initially draw the texture tiles for the appropriate view.

Compare the redraw times in Table 2(a). In all cases except one (Tool A drawing the Flash design), glLayoutView is twice as fast or more. To help answer the question of why Tool A compares so well, refer to Table 2(b) which breaks down glLayoutView's rendering performance into three parts: the time to walk the design database, the time to compute the textures, and the time to drive the graphics display with the texture data. The time to drive the display is neglible because the Expert3D hardware very efficiently

| | Flash | SU_Block |
|---|---|---|
| Magic | 21.5 | 71 |
| Tool B | 18.5 | 76 |
| Tool A | 3.8 | 18 |
| glLayoutView | 9.6 | 6.3 |

(a) Redraw times for a full screen view.

| | Flash | SU_Block |
|---|---|---|
| Database Access | 5.5 | 3.1 |
| Texture Computation | 4.0 | 3.1 |
| Graphics Library | 0.07 | 0.14 |

(b) glLayoutView redraw times broken down.

| | Flash | SU_Block |
|---|---|---|
| No Hierarchy Cache | 9.6 | 34.0 |
| Global Change Time | 1.0 | 3.7 |
| × 2 processors | 5.1 | 3.1 |
| × 4 processors | 2.7 | 1.8 |

(c) glLayoutView under different rendering conditions.

**Table 2: Static rendering performance. All times are in seconds.**

draws textures on the screen. This level of performance is comensurate with leading technology currently available to the average VLSI layout designer. Perhaps suprisingly, the time to compute the textures is comparable to the time of simply accessing all the rectangles. So some of the performance difference between Tool A and glLayoutView may be attributable to the level of optimization in walking the database, but this cannot be verified both because Tool A is proprietary, and also because no attempts were made to enhance Magic's data structures as part of this paper.

Another look at Table 2(a) shows that while glLayoutView is slower than Tool A in drawing Flash, it is 3× faster in drawing SU_Block. Table 2(c) reveals why. The first row shows the effects of the hierarchy cache. The time for Flash is the same since it contains no hierarchy, while the time for SU_Block is 5.5× slower versus when the cache is enabled. If Tool A's time for SU_Block is compared against the 34.0 seconds it takes glLayoutView with the hierarchy cache turned off, Tool A again is faster by a factor of two. It is clear that Tool A is highly optimized.

Referring back to Table 2(c), the second row shows the effects of the optimization of precomputing the top of the pyramid that was described in Section 6.2. In both designs, this recomputation time is faster than the time to draw it from scratch. This time reflects the delay in redrawing the entire screen if a global change were made to the appearance of the design. The faster recomputation time for Flash versus SU_Block is due to the smaller number of layers present.

The last two lines show redraw times when parallelized across two or four processors. In both designs, the speedup to two processors is nearly perfect, and the four processor speedup is a very acceptable 3.6[4]. Only when comparing glLayoutView's parallelized

[4] In the four processor case, glLayoutView was run on a $4 \times$ 400Mhz UltraSparc II processor system. The 3.6 speedup was

performance against the others does glLayoutView win on all accounts.

One might ask if it is fair to compare the parallelized performance of one application against the single-threaded performance of another. In this case, it is fair because even if the other tools were parallelized, their redraw times would not be faster. In fact, they would most likely be *slower*. This is because standard graphics displays do not allow for efficient drawing to a graphics device by multiple threads at the same time. Any attempt to do this would result in the drawing threads competing for control of the framebuffer, resulting in a serial execution. The overhead of the thread switching would cause these implementations to be slower than the single-threaded case. The other tools could certainly change their architecture to allow for more efficient parallelization, but it is argued that any changes would require these tools to more closely resemble the architecture presented here.

### 9.2 Comparing Dynamic Rendering Performance

The previous section does not capture how the texture tiles can be efficiently reused. Although the rendering times of glLayoutView compare favorably with the other tools, the actual user experience is even better because the tile creation time penalty is only paid each time a tile is not present in the texture tile cache. The computational cost of moving the viewpoint a small amount is nearly zero, while the cost for the other tools is the same as drawing it from the original viewpoint. Imagine drawing a full screen view one hundred times. The cost for glLayoutView would simply be the time to draw the screen once, while the cost for the other tools would be the time to draw the screen multiplied by one hundred.

To quantify this behavior, each tool is made to go through a series of viewpoint changes:

1. Full screen view
2. Zoom of $2\times$ on the upper left portion of the design
3. Zoom of $2\times$ on the upper right portion of the design
4. Zoom of $2\times$ on the lower right portion of the design
5. Zoom of $2\times$ on the lower left portion of the design
6. Full screen view

The effects of reusing textures can be measured if the number of intermediate viewpoints to render between the six main viewpoints is varied. Tables 3 shows the results. The first column shows the combined amount of time to render the designs from the six main viewpoints. The remaining columns show the time to render the designs when a different number of intermediate viewpoints are also rendered. As the number of intermediate viewpoints is increased, the movement more closely resembles a smooth animated motion. Now the effects of the texture tile cache are clearly demonstrated because the render times for glLayoutView are constant.

### 9.3 Comparing Image Quality

Figures 5(a), 5(b), 5(c), and 5(d) show screenshots of glLayoutView, Tool A, Tool B, and Magic rendering the Flash design. Each of the tools, except glLayoutView, shows the effects of massive aliasing. Tool A and Magic draw the layers in stacking order so the highest layer is the only one visible. Tool B draws the smallest rectangles last so the final appearance is speckled. Compare these images to the view created by glLayoutView, where the gross features are clearly visible and the wire densities are apparent.

computed by comparing the $1\times$ and $4\times$ tests on this system.

| | Flash | | | |
|---|---|---|---|---|
| | 0(6) | 1(12) | 3(24) | 6(48) |
| Magic | 69 | 139 | 276 | 531 |
| Tool B | 57 | 87 | 151 | 265 |
| Tool A | 15 | 26 | 48 | 86 |
| glLayoutView | 8 | 8 | 8 | 8 |
| | SU_Block | | | |
| | 0(6) | 1(12) | 3(24) | 6(48) |
| Magic | 212 | 444 | 929 | 1816 |
| Tool B | 218 | 433 | 700 | 1293 |
| Tool A | 52 | 97 | 184 | 351 |
| glLayoutView | 9 | 9 | 9 | 9 |

**Table 3: Total redraw times, in seconds, of the two designs from different viewpoints. The column headings represent the number of intermediate viewpoints that were visited. The total number of rendered frames is shown in parentheses. glLayoutView was run with all optimizations turned on, including a multi-threaded factor of two.**

## 10. CONCLUSIONS

The results are promising. By applying common techniques from the graphics field to rendering VLSI layouts, a new level of visualization was attained. The most unexpected result from this research was the observation that the images created by glLayoutView closely resembled a die photo or a high quality chip plot. This type of detail can help IC layout designers a great deal.

Additional work needs to be done to make the techniques presented in the paper useful enough to replace existing systems, but as VLSI layouts continue to grow in size, the need to visualize them quickly and accurately will also grow.

## 11. ACKNOWLEDGEMENTS

The authors would like to thank Matthew Eldridge for his suggestions and insight throughtout the course of this research. This work was supported by DARPA contract MDA904-98-C-A933-P00011 and a gift from IBM Corporation.

## 12. REFERENCES

[1] Expert3D Datasheet and Description *http://www.sun.com/desktop/products/Graphics/expert3d*

[2] J. Kuskin et al., "The Stanford Flash Multiprocessor," in *Proceedings of 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994, pp. 302–313.

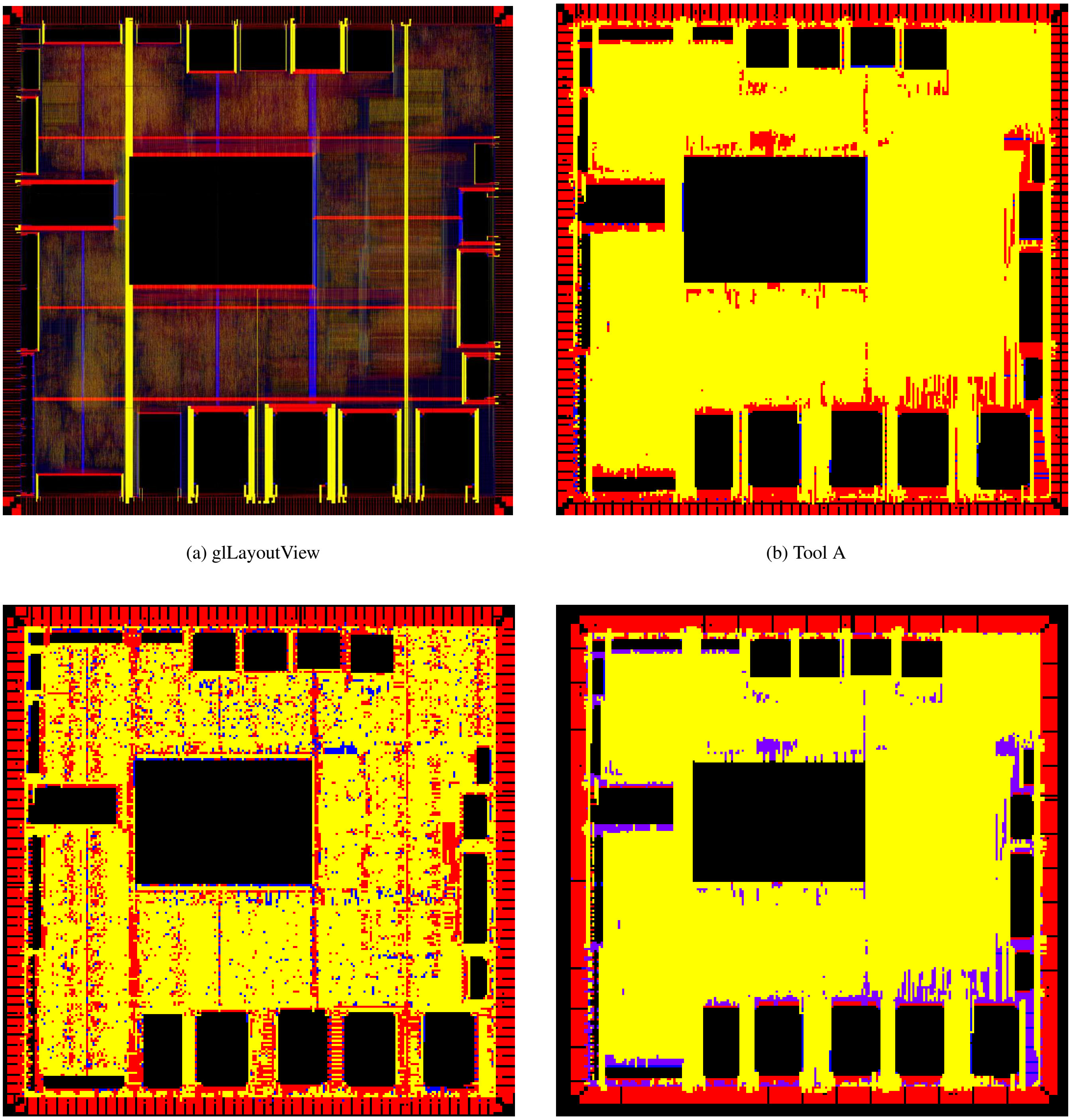
[3] OpenGL Specification *http://www.opengl.org*

[4] J. Ousterhout "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools," *IEEE Transactions on CAD*, Vol. 3, No. 1, 1984, pp. 87–100.

[5] J. Ousterhout et al., "Magic: A VLSI Layout System," *21st Design Automation Conference*, 1984, pp. 152–159.

[6] C. Tanner, C. Migdal, and M. Jones "The Clipmap: A Virtual Mipmap," *SIGGRAPH*, 1998, pp. 151–158.

[7] L. Williams, "Pyramidal Parametrics," *SIGGRAPH*, 1983, pp. 1–11.



(a) glLayoutView

(b) Tool A

(c) Tool B

(d) Magic

**Figure 5: Screenshots of the Flash design. The design database for Flash used in this paper does not contain the data for the internal memories. This is why the screenshots show empty space where the memories would be placed.**