THE CHIPMAP™: VISUALIZING LARGE VLSI PHYSICAL DESIGN DATASETS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jeff Solomon
December 2002

© Copyright by Jeff Solomon 2003
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William J. Dally

Approved for the University Committee on Graduate Studies:

Abstract

Continuing trends in computer systems have adversely effected the ability of VLSI chip designers to operate graphically on any significant fraction of a layout in an interactive or accurate manner using current methods. While transistor counts of designs have grown at the rate given by Moore's Law, key components contributing to the real-time and accurate display with existing methods (CPU to memory bandwidth, CPU to GPU bandwidth, monitor resolution) have not grown as fast. Consequently, when using today's CAD systems to view a large chip at low zoom, the display can take dozens of seconds or more to refresh and the resulting image can contain visually misleading artifacts which yield no clues about the design's structure.

We present a new visualization infrastructure for VLSI physical design datasets called a "chipmap." First, we show how it can be used to visualize the canonical VLSI database, the layout, in an accurate, interactive, and fluid manner. Visual fidelity is achieved with standard computer graphics anti-aliasing techniques modified to take advantage of the special rectilinear, hierarchical, and layer dependence properties inherent to VLSI datasets. Interactivity is realized by using texture mapping and mipmapping so the information sent to the display is bounded, and the image rendered on the display is filtered correctly.

Our experimental implementation shows that real-time navigation can be achieved on arbitrarily large layouts with a reasonable memory overhead. Results also show an average image error of about 2% (RMS error) between our images and rigorously generated "perfect" images while other layout systems produce errors up to 38% when compared to these images.

Next, we extend the use of a chipmap showing how it can be used to visualize other types of VLSI physical design data. Common operations include selecting feature sets,

back-annotating analysis information and computing device densities. Using these tools, we demonstrate additional accurate and interactive visualizations of floorplan, DRC, critical timing paths, clock skew, and other information.

Acknowledgements

My father said that I would not want to leave graduate school and he was right. The past seven (!) years have been the best of my life and I have many people to thank for it.

Thanks to Charlie Orgish for his amazing support of the machines in our lab. He always allowed me the freedom to configure the machines the way I thought best and he always made sure that we got the best new workstations and equipment to play with.

Thanks to Sun Microsystems for all of their support with my research. Thanks to John McGuigan for loaning me the powerful machine with which I did the majority of my development, and thanks to John, Peter Denyer and Levon Mitchell for allowing me to demonstrate ChipMap in Sun's DAC booth for three years in a row.

I would like to thank my advisor, Mark Horowitz, for giving me the opportunity to be in his group, for having unbelievable patience with my frequent "diversions," and for setting an incredible example of how to perform good research. Thanks, Mark, for always encouraging me to find my own way.

I owe a debt of gratitude to the friends I've made while at Stanford. I will always remember the softball and football games with the Flash team, the colorful lunches with my friends in the graphics lab and the late night frag-fests in the basement with the infamous *Clan 9 from Outer Space*. The best thing about Stanford has been the people, who have each shown me something in themselves that I truly admire. A hearty thanks to Joel Baxter, Robert Bosch, Matthew Eldridge, Mark Heinrich, Ron Ho, Chris Holt, Greg Humphreys, Hema Kapadia, Jeff Kuskin, Dean Liu, Dave Ofelt, John Owens, Matt Pharr and Kekoa Proudfoot.

Thanks to Chris Holt for being one of my first friends at Stanford, for always being there when I needed it, and for finally realizing that what Ahnuld really says is, ”Here, let me give you a haaand!”

I couldn’t have made ChipMap without Matthew Eldridge. Thanks for all the long discussions about texture management, for entertaining me with tirades about ridiculous stories on Slashdot, and for receiving my outlandish rants that happily never made it beyond your inbox.

Thanks also to Matt & Jen Rhodes-Kropf for being great friends all through college and graduate school. Both of you have stuck by me through the good times and the bad and I really appreciate it.

I would like to thank my family whose love, encouragement, and support have always been, and always will be, very important to me. Thanks to my sister and brother-in-law who have made being an uncle the greatest job in the world. Thanks to my father, who educated me on the difference between right and wrong and taught me to expect more of myself, and to my mother, for setting a standard of excellence in a job well done and for showing me how to be a good person. Thanks to you both for loving me unconditionally and for always, always being there for me.

I would like to thank my grandparents David, Florence, Alva and Marie for the sacrifices they made for their children, their grandchildren and future generations. I would not be where I am today if they had not overcome their struggles.

Lastly, I’d like to thank my wife, Stacey. I am eternally grateful that she so expertly edited this document from the occassionally incoherent ramblings that I frequently produce to the (hopefully) readable prose that is now before us. I didn’t think someone like her existed for me in this world and I’m glad I was wrong. She is an everlasting source of love, devotion and happiness to me as well as my best friend. To her I say – without you, I would have foundered.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

1.1 Current Visualization in VLSI Design 2

1.1.1 Current VLSI Layout Visualization 3

1.1.2 Visualizing Other VLSI Datasets 6

1.2 Research Goals & Methods 6

1.3 Thesis Organization 7

2 Computer Graphics Review & Design Examples **9**

2.1 Computer Graphics Review 9

2.1.1 Rasterization 10

2.1.2 Compositing 11

2.1.3 Texture Mapping 12

2.1.4 Mipmapping 13

2.1.5 Clipmapping 16

2.1.6 Graphics Review Summary 18

2.2 Three Example VLSI Designs 18

3 A Chipmap **23**

3.1 A Chipmap Pyramid 24

3.2 Region 1: Geometry 25

3.2.1 Texture vs. Geometry Boundary 26
3.2.2 Texture/Geometry Boundary Grid Resolution Invariance 29
3.3 Region 2: Dynamically Generated Texture Data 30
3.3.1 Tiled Texture Pyramid 30
3.3.2 Rendering Texture Data 33
3.3.3 Panning and Zooming Without Hardware Support 34
3.3.4 A Texture Tile Cache 35
3.3.5 Managing Dedicated Graphics Texture Memory 36
3.4 Region 3: Static Texture Data 37
3.4.1 Static/Dynamic Texture Boundary 37
3.5 Rendering Architecture 38
3.6 Example Chipmaps 39
3.6.1 Viewing Very Large Designs 40
3.7 Summary 41

4 Image Generation **42**
4.1 Anti-aliasing Techniques 43
4.1.1 Level Zero Rasterization & Averaging 44
4.1.2 Point Sampling 45
4.1.3 Area Sampling 47
4.1.4 Rasterization 49
4.2 Compositing 49
4.2.1 Coverage Equals Transparency 50
4.2.2 Different Layer Compositing 50
4.2.3 Same Layer Compositing 51
4.2.4 Color Compositing 52
4.3 Texture Tile Creation Strategies 53
4.3.1 Coverage Map Tile Creation 53
4.3.2 Direct Tile Creation 54
4.3.3 Discussion 55
4.4 Pre-rasterized Hierarchy 57

4.4.1 Hierarchy Data Format 58
4.4.2 Using Hierarchy Data In Tile Creation 58
4.4.3 Sub-design Selection Algorithm 59
4.4.4 Selecting Hierarchy Cache Size 61
4.4.5 Hierarchy Errors 63
4.5 Summary 65

5 Other Visualizations **66**
5.1 Data Selection Visualization 67
5.2 Floorplan Data Visualization 69
5.3 Back-annotated Data Visualization 71
5.3.1 Static Timing Analysis Visualization 71
5.3.2 Clock Skew Visualization 74
5.4 Data Density Visualization 75

6 Results **78**
6.1 Implementation 78
6.2 Image Quality Comparison 80
6.2.1 Formal Image Quality Comparison 80
6.2.2 Ad-hoc Image Quality Comparison 81
6.3 Rendering Speed Comparisons 85
6.3.1 Unaccelerated Rendering Times 85
6.3.2 Hierarchy Cache Rendering Times 86
6.3.3 Parallelized Rendering Times 87
6.3.4 Rendering Speed Comparison Summary 88
6.3.5 Refresh Speed Comparison 89
6.4 Memory Usage Comparison 89
6.5 Summary 90

7 Conclusions **91**

Bibliography **94**

List of Tables

2.1 Example design statistics 22

3.1 Average shorter dimensions 28
3.2 Average shorter dimensions times ten 30

6.1 Formal image comparisons 81
6.2 Unaccelerated rendering comparisons 85
6.3 Hierarchy cache times 86
6.4 Parallelized rendering times 88
6.5 Rendering times summary 88
6.6 Refresh time summary 89
6.7 Memory usage comparison 90

List of Figures

1.1 An example VLSI layout 3

2.1 Rasterization process 10
2.2 Compositing example 12
2.3 Texture mapping 13
2.4 Mipmapping example 14
2.5 A texture's mipmap 14
2.6 Mipmap pyramids 15
2.7 A clipmap 16
2.8 Databuffer design 19
2.9 SU_Block design 20
2.10 Flash design 21

3.1 A chipmap 24
3.2 A tiled texture pyramid 31
3.3 Example chipmaps 40
3.4 Intel's McKinley design as a chipmap 41

4.1 Rectangles at various texel resolutions 43
4.2 Three point sample frequencies 46
4.3 Coverage values 48
4.4 A coverage map example 54
4.5 An updated chipmap 56
4.6 The hierarchy cache visualized 60

4.7 Hierarchy cache size vs. Speed-up 62
4.8 Hierarchy data error 63
4.9 Hierarchy compositing errors 64

5.1 SU_Block error data selection 67
5.2 Flash flip-flop data selection 68
5.3 Flash floorplan 70
5.4 Flash floorplan 70
5.5 Timing analysis visualization 72
5.6 Data encoding comparisons 74
5.7 Clock skew visualization 75
5.8 Clock skew visualization amended 76
5.9 Flash metal density 77

6.1 Formal image comparisons 82
6.2 Formal image comparisons 83
6.3 Ad-hoc image comparisons 84
6.4 Hierarchy effectiveness by cache size 87

Chapter 1

Introduction

Of the many challenging aspects of *Very Large Scale Integrated* (VLSI) chip design, perhaps data management is the most difficult. While the number of transistors on the largest designs continues to double every two years [Intel, 2000a], the number of humans responsible for those designs grows at a much slower rate. Consequently, each designer is faced with the ever increasing task of managing more and more data.

In the face of such enormous challenges, the *Electronic Design Automation* (EDA) industry has created better tools to allow designers to cope with expanding datasets. Designers today can describe designs with high-level languages, then synthesize them automatically into transistors. In a perfect world, these *Computer Automated Design* (CAD) tools would allow higher and higher levels of abstraction, keeping pace with chip growth and hiding the underlying complexity. Unfortunately, the reality of VLSI chip design in 2002 is not that good. Manual intervention is still almost always necessary. Additionally, CAD tools are not so good that a designer only need specify a high level design description and have the photo-lithographic masks used for chip manufacture produced as the output. Designers are still required to guide and massage the process. Decisions that have important consequences later in the design cycle are constantly being made. Even though many designers never face the task of operating on the entire design at once, those who do are burdened most by the enormous size of today's designs.

Enter *visualization*. Visualization is the process of putting a dataset into a form most amenable to communicate a particular idea in a visual manner. Typically, visualizations

are images, but that is not a strict requirement. Anything that takes advantage of a human being's tremendous ability to assimilate information in a two or three dimensional visual context is a viable visualization.

The goal of our research is to show how visualization can be used to facilitate the design of VLSI chips. Specifically, we address how we can aggregate the enormous amounts of data present in the various stages of VLSI design and create images and interactions that will allow a designer to do the job more efficiently and more effectively. We see visualization as a productivity tool. Our techniques will not inherently make VLSI chips run faster, nor make them smaller or consume less power, but visualization can help a designer work faster and smarter.

1.1 Current Visualization in VLSI Design

The previous published work on visualizing large VLSI datasets is scarce. First, our own work [Solomon and Horowitz, 2001] is a shorter and less detailed examination of improving large VLSI layout redraw. A primary goal of this thesis is to expand upon the first paper. Next, Restle [Restle, 2001] showed visualization techniques for small datasets (i.e. the voltage and current for one wire) with animation, and other larger dataset visualizations (congestion, power grid, and clock skew); however, all the visualizations were custom-made for specific datasets and the techniques were never generalized for broad purposes.

Our search of previous work leads us to believe that most of the visualization work in VLSI has taken place in the EDA industry itself. Small scale visualization, such as trying to model a single circuit in exact 3-D detail, is offered by more than one company [Ansoft, 2002] [Silvaco, 2002], but for large scale visualization, most companies offer only one-off or ad-hoc visualization solutions for specific datasets.

Of course, by default, most VLSI design systems do come with perhaps the most important visualization tool of all, a VLSI layout editor or viewer. In the next section, we will explore the current state-of-the-art in VLSI layout viewing.

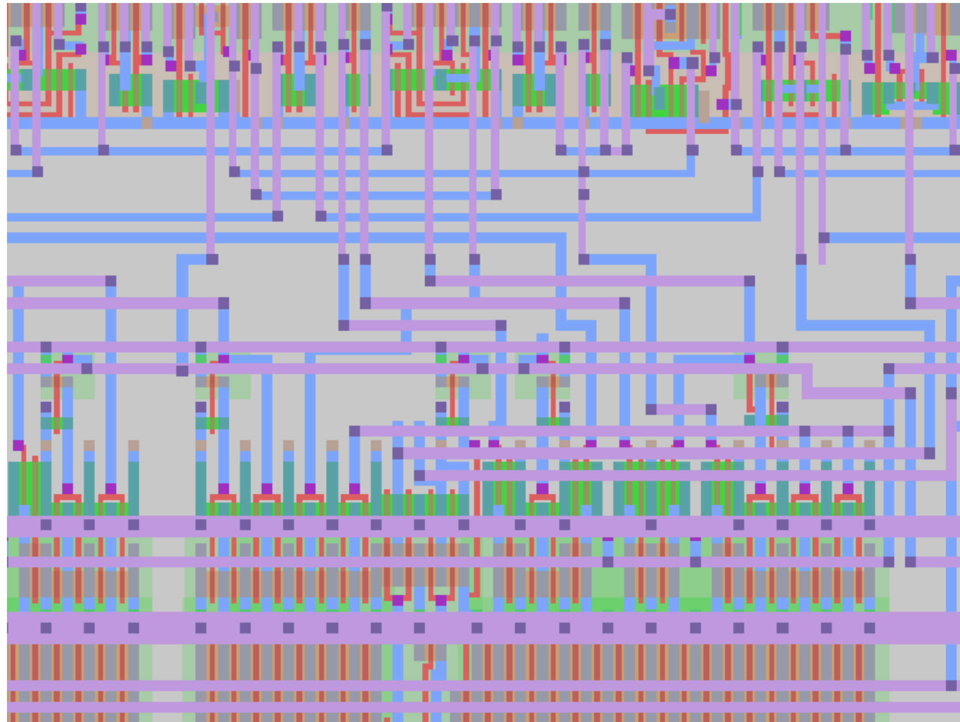


Figure 1.1: An example VLSI layout

1.1.1 Current VLSI Layout Visualization

The VLSI layout is *the* fundamental visualization in VLSI design since it is the information that best represents what ultimately will be manufactured.

Figure 1.1 is an example of a VLSI layout which shows the various layers of the design, represented by different colors and patterns. Polygons[1] of the same color or pattern make up an entire layer, and all of the layers together form the design. The visualization represented in Figure 1.1 shows about 1,000 rectangles. To create the image, each visible element is drawn on the display by means of the *Central Processing Unit* (CPU) sending the object's coordinates to the *Graphics Processing Unit* (GPU), which renders the image on the display. When the display needs to be redrawn, the image is cleared and the process is repeated.

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

This algorithm can create VLSI layout visualizations effectively in real-time when the number of rectangles to draw on the display is below a certain threshold. Problems only arise when trying to view orders of magnitude more layout information on the screen at one time than is shown in Figure 1.1, that is, millions and millions of rectangles. Drawing millions of rectangles on a computer display with existing layout editors has two main problems: the redisplay time is long and accuracy of the image is poor.

A lack of caching causes the inordinately long redraw times. There is always an unavoidable amount of time necessary to draw a display the first time. When performing an incremental viewpoint change, however, existing tools cache nothing; they actually redraw the entire image from scratch.

The images accuracy is poor due to the fact that many of the small rectangles have a scaled size on the display that is less than one pixel in at least one dimension, and the graphics library does not have the ability to handle this appropriately. All layout editors that we know of use the X11 [X Consortium, 1986] Windowing System graphics library, which not only does not have the ability to draw rectangles with sub-pixel resolution, but also guarantees that all rectangles will be drawn a minimum of one pixel in both directions[2]. When the layout features are forced to have one pixel dimensions, gross errors are evident on the display.

For many designers, the problems of slow redraw and image inaccuracy are frustrating. If a designer knows that a certain viewpoint change may cause a delay of dozens of seconds in tool responsive while the screen is refreshed, they may change their interaction patterns to avoid it.

In an attempt to mitigate these problems, EDA companies [Cadence, 2002a] [Synopsys, 2002] [Mentor Graphics, 2002] [Magma, 2002] have employed many techniques to speed the display process. Here are the ones that we were able to uncover[3]:

Draw no rectangles Start with a blank display, possibly showing only the top level bounding box. The user selects an area of the design explicitly to display. Users are aware that redisplay can be lengthy if too much area is selected to display at once.

---

[2] See XDrawRectangle(3X11) for details.

[3] Information about these techniques was obtained by observing various commercial tools run or via private communications with engineers at the cited companies.

Draw only large rectangles Allow a user to specify a minimum dimension threshold; rectangles smaller than the threshold are not drawn.

Draw random rectangles When too many rectangles are present to draw, only draw the largest rectangles and some additional smaller rectangles that are randomly selected. The number of randomly selected rectangles will usually be a small fraction of the total number.

Stop drawing on demand Allow a user to cancel the redraw process while it is in process. Occasionally, the user can signal a redraw interruption by trying to change the viewpoint, but frequently it is necessary to press a mouse button or hit a particular keystroke.

Hide all hierarchy Default to draw all hierarchical detail as hidden and draw bounding boxes instead. Sometimes it is desired to hide hierarchy, but this technique is also used to speed re-display.

Unfortunately, the problems we have described will continue to get worse if current trends in computer systems hold. First, chips are becoming more and more complex, and the number of transistors on them is growing faster than the bandwidth and computation required to display them[4]. In other words, the time for a single redraw is slowly growing over time. Additionally, the resolution of the average display is growing *much* more slowly than any other aspect of computer systems. While bandwidth and computation have been steadily growing at rates approaching 30% to 60% annually, display resolution has been increasing at a paltry 10% [5]. This slow growth of resolution means there are comparatively more rectangles with sub-pixel resolution than in the past, which results in even more inaccuracy.

[4] The time to redisplay a chip is related to all aspects of the system–the memory bandwidth, the CPU speed, the graphics bandwidth, the software efficiency. Given that transistors are growing at a rate predicted by Moore's Law [Moore, 1965], all other system components must grow at a rate equaling to or exceeding this rate to *maintain* current redraw speeds. This has not occurred to date and is unlikely to happen based on existing trends.

[5] This figure was computed by assuming the average display in 2002 was 1920×1200 and the average display in 1984 was 1024×768. This is based on our estimates and Stanford's CS448A [Akeley and Hanrahan, 2001] class notes.

A corollary to these trends is that graphical display was not a problem at the advent (circa 1980) of interactive VLSI design for just the opposite reason that it is a problem now. That is, compared to design size, displays were large enough and processors were fast enough so that redraw and accuracy were not huge problems. At this point, research focused on efficient ways to store geometry in a database. The list of early interactive VLSI layout editors include Caesar [Ousterhout, 1981], Cabbage [Hseuh, 1979], KIC2 [Keller and Newton, 1982], and others [Kedem, 1982] [Bentley and Friedman, 1979]. This period of research culminated with the Magic Layout System [Ousterhout et al., 1984] which has since been the mainstay of academic VLSI layout editors. In fact, our test implementation was built upon Magic since it was best available layout editor with source code available.

In conclusion, the current experience of viewing a large layout with existing layout systems is becoming slower and less accurate as time passes. Not only does the slightest viewpoint change cause the entire redraw process to be repeated, but the accuracy of the image once it's complete is prone to massive errors.

1.1.2 Visualizing Other VLSI Datasets

As VLSI layouts continue to grow at an enormous rate, so do the other datasets related to VLSI design. A large chip is going to have an equally large database that holds the wire connectivity, timing, IR drop, power, clock skew, and floorplan information. Since these datasets can be just as large as the layout, they suffer the same display problems we have previously described for the layout.

1.2 Research Goals & Methods

The goals of our research are two-fold. First, we attempt to solve both problems related to layout display – accuracy and speed. Second, we try to extend the techniques developed in solving the display problem to produce other visualizations for the other types of datasets we have mentioned.

In addition, we want to create visualizations as part of an interactive system for VLSI designers. Our system will be used in a real-time manner, not as a batch job for off-line viewing, so we would like the following constraints to always be met:

Fast display A user should wait a minimal amount of time for a visualization to appear; optimally, no wait at all. In the cases when a delay is unavoidable, the user should be given a visual cue of the progress. In all cases, the display should be updated fast enough so that fluid interaction is possible. This usually is about 5-10 frames per second (fps).

Useful display The visualizations should always show useful information to the designer. In the context of VLSI layout, this means an accurate display of information. Regardless of the amount of data being visualized, it will always be aggregated in a way that does not mislead the user.

Interactive The visualizations should always be interactive. Even though some visualizations will take non-zero time to produce, the user should be able to stop or change the visualization being created with minimal delay.

Reasonable memory overhead A system that meets the previous three goals but requires all of the system memory would not be very useful. We will try to minimize the amount of memory needed for our techniques so that our system can co-exist with other CAD tools running on the same platform.

1.3 Thesis Organization

Many of the visualization techniques used in this work are extensions of techniques used in computer graphics. Chapter 2 first reviews these methods and provides the foundation for the rest of the thesis. The chapter also describes three design examples that will be used to benchmark our implementation throughout the thesis.

The next two chapters give our solution to solving the VLSI layout redraw problem. In Chapter 3, we introduce the primary data structure of this thesis, the *chipmap*, and show how it gives us a way to efficiently manage the layout visualization regardless of viewpoint

or overall layout size. Next, in Chapter 4, we describe the techniques we use to generate the image data that is contained in our chipmap structure.

In Chapter 5, we describe how we can create different types of data to place into the chipmap structure to view alternate types of information in a useful manner. While the previous two chapters focused exclusively on layout display, Chapter 5 shows techniques for selecting data, back-annotating information, and other types of visualization in general.

In Chapter 6 we show the results of trying to solve the layout redraw problem. Specifically, we'll measure whether we were able to produce accurate visualizations of layout in a timely manner and with a reasonable memory overhead.

Finally, in Chapter 7, we speculate on possible future directions for chipmap and VLSI visualization in general, and we summarize our thesis contributions.

Chapter 2

Computer Graphics Review & Design Examples

To meet our research goals of fast display speed coupled with accuracy, we drew upon the techniques and technology currently prevalent in the field of computer graphics. This chapter presents those techniques as a foundation for subsequent chapters. Additionally, we introduce three example designs that represent the full range of designs commonly found today. These designs will serve as our benchmarks throughout the rest of the thesis.

2.1 Computer Graphics Review

A significant component of the field of computer graphics is the study of techniques to accurately and efficiently transform input geometric descriptions into a 2-D array of pixel values that is viewed on a computer display. Since this is basically what we are trying to accomplish given the specific geometric input of VLSI layout databases, it is no surprise that the key techniques we employ come directly from computer graphics.

We now review some of these key techniques to lay a foundation for the discussion in Chapters 3 and 4. The first two techniques will help us with image accuracy, while the last three help with redisplay speed and interactivity.

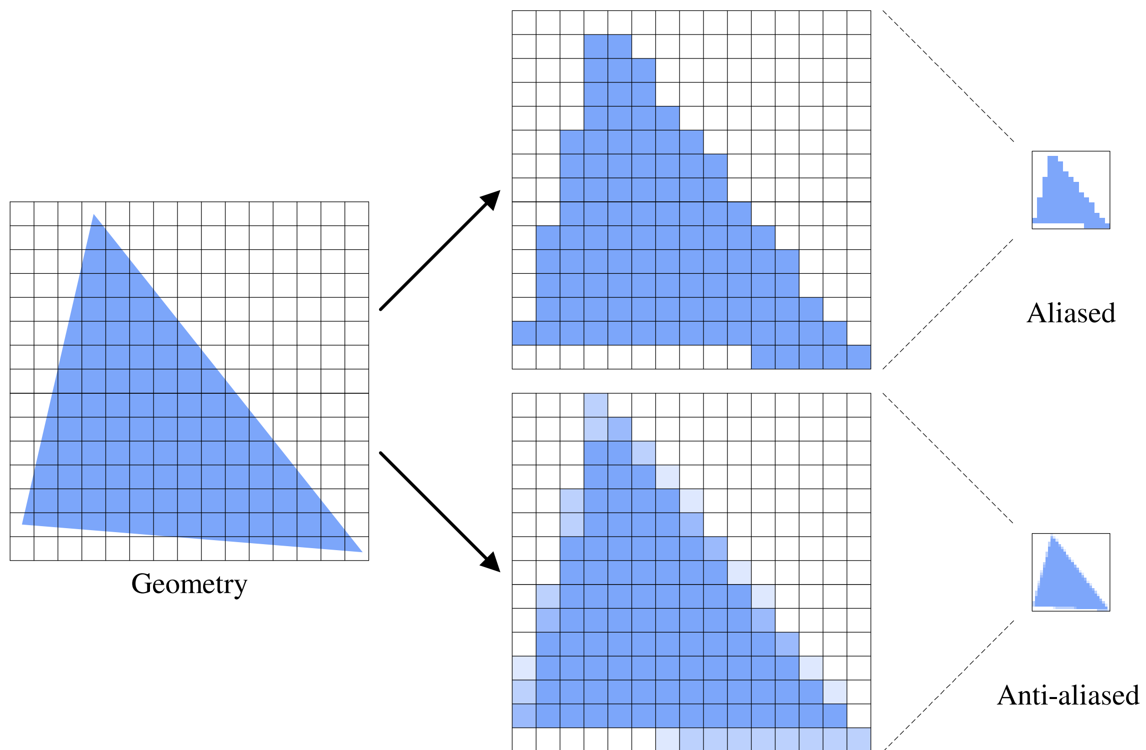


Figure 2.1: The triangle on the left is rasterized into pixel values on the two middle images. On the top, the pixels whose centers are covered by the triangle are colored the same as the triangle, while on the bottom, the pixels values are set to vary by an amount proportional to the percentage they are covered by the triangle. On the far right, the rasterized triangles are shown roughly at natural scale to demonstrate how each rasterization scheme effects the triangle's appearance. The anti-aliased triangle more accurately reflects the shape of the original geometry.

2.1.1 Rasterization

The first important technique for image accuracy is *rasterization* because it sets the fundamental way that we will deal with finite pixel size. Rasterization is the process of converting a geometric polygon into a set of two-dimensional pixel values. Each individual part of the polygon is called a *fragment* after it has been converted into a value which can then be added to a pixel. Given our input specification, proper rasterization is absolutely essential to achieve an accurate representation.

Let us demonstrate this with an example. On the left side of Figure 2.1, a triangle has a pixel grid superimposed on it. In the middle top, the fragments are colored the same as the triangle for each pixel where the triangle covers the pixel center. In the middle bottom, the

fragments' color varies according to the percentage that is covered by the triangle. On the right side of the figure, the two rasterized triangles are shown on a smaller, more realistic scale to demonstrate the effect each rasterization scheme has on the final appearance of the triangle. Most people agree that the upper triangle appears inaccurate and "jaggy," while the lower triangle appears more "correct" and smooth.

We say that the upper rasterized triangle is *aliased*, because the rasterized outcome contains gross errors that make it appear blocky. The lower triangle is *anti-aliased*, which means that more information has been encoded into the fragments, allowing the triangle to be reproduced more accurately. Although anti-aliasing leads to more accurate and pleasing images, it does require more computation which is a consideration for many applications, including ours, although this cost is unavoidable for us since accuracy is one of our goals.

Notice that as the object size shrinks relative to the pixel size, the amount of perceived blockiness grows. Said another way, it is most important to rasterize the *smallest* objects in an anti-aliased fashion if visual accuracy is desired. Since visual accuracy is one of our requirements and since the overwhelming majority of input primitives in VLSI designs will be very small at low zoom in relation to the pixel grid, anti-aliasing (and thus rasterization) is a topic of great importance for us. Chapter 4 will discuss rasterization and anti-aliasing in-depth.

2.1.2 Compositing

The second important technique for image accuracy is *compositing*, which is the process of taking rasterization output (i.e. fragments) and producing final pixel values. This is important because image accuracy is only achievable with techniques that let us model many, many small rectangles that each overlap a single pixel. A simple example of compositing is shown in Figure 2.2.

The figure shows two triangles being added into a final image. When adding the second triangle we have made an arbitrary decision that it should appear on top of the first triangle. For each pixel, we then composite the two triangles' fragments together. For many of the pixels, the compositing is trivial because it contains fragments from only one triangle. In this case, the pixel value is fragment value. It is also simple in the cases where the second

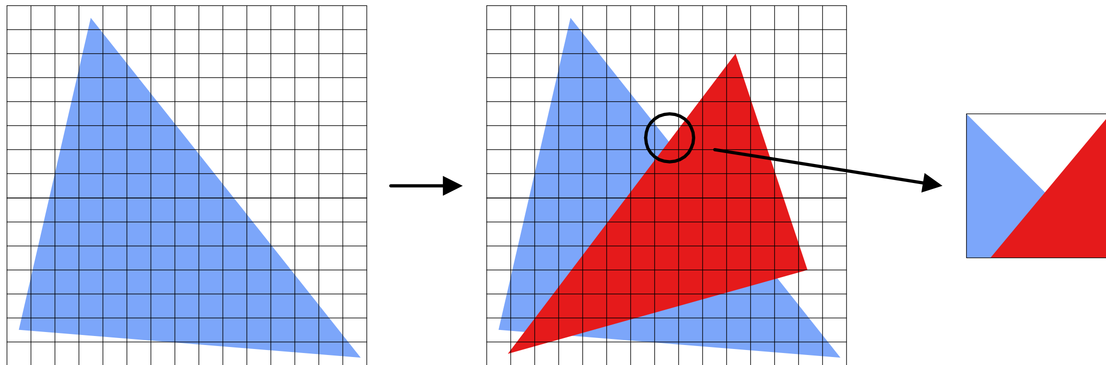


Figure 2.2: Two triangles are each added to the final image to show an example of compositing. The highlighted pixel is an example of a pixel where the compositing is the most complex, that is, the final pixel value should reflect that fact that both triangles partially overlap it, and that the red triangle is on top of the blue triangle.

red triangle completely covers a pixel that is also overlapped by the blue triangle since the blue fragment will not contribute to the final pixel value.

The complex case arises when both triangle's fragments partially contribute to a pixel. This case is highlighted on the right in Figure 2.2. In this situation, the final pixel value is a combination of the two fragments given how much each fragment overlaps the pixel, and how much the fragment on the top overlaps the fragment on the bottom. The most straightforward way to compute the final pixel is to linearly interpolate between the two fragment values. We will go into much further detail on how we composite these types of fragments in Chapter 4.

2.1.3 Texture Mapping

Texture mapping is the prevalent computer graphics technique that helps us achieve fast redisplay. An image of a VLSI layout is transformed into a texture which is then efficiently rendered on the display. Figure 2.3 shows *texture mapping* in its simplest form, applying an image to geometry as a decal.

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

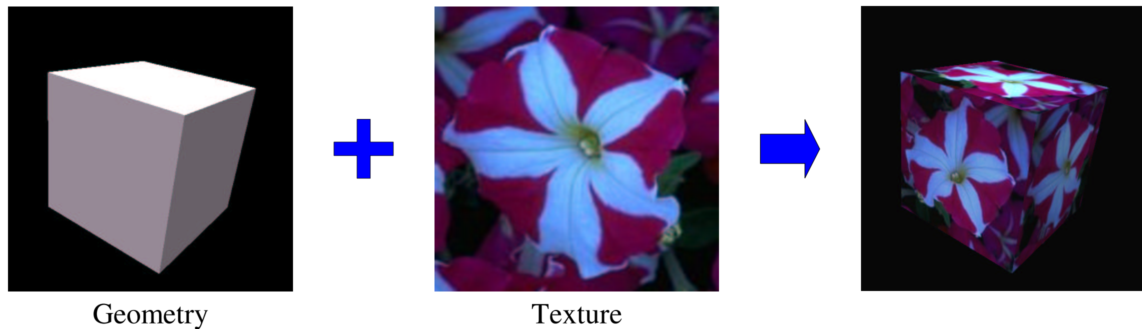


Figure 2.3: The cube on the left is texture mapped with the image of the flower. The result is shown on the right.

on the display in that a texel can represent more or less area than a pixel depending on the texture's final scaled size on the screen.

The main advantage of texture mapping from our point of view is hardware efficiency. When texture mapping is supported in hardware, the process of rendering texture-mapped geometry is very fast, allowing incremental changes in the viewpoint. This feature is essential for real-time display.

2.1.4 Mipmapping

The second important technique for fast redisplay is *mipmapping*. Figure 2.4 shows a 2-D texture on the left with the scene containing it, from the video game Unreal [Epic Games, 2000], on the right. Box 1 shows an area in the final scene where the texture would have about a one-to-one correspondence between texel and pixel. On the the other hand, box 2 shows an area that is much smaller than the original texture, which means that the larger texture area will have to be mapped to the smaller pixel area in some fashion. It would be inefficient to filter down the original texture to the smaller size on-the-fly for two reasons. First, the more texels being mapped to a single pixel, the more bandwidth required to process the conversion. Second, since the final value of the pixel is an aggregation of all the texels, additional computation is needed to perform the down-sampling.

As a solution to these two problems, instead of using the original texture in all situations, we precompute a set of averaged-down images and use texels from these smaller images where appropriate. Figure 2.5 shows this set of images from the carpet texture,

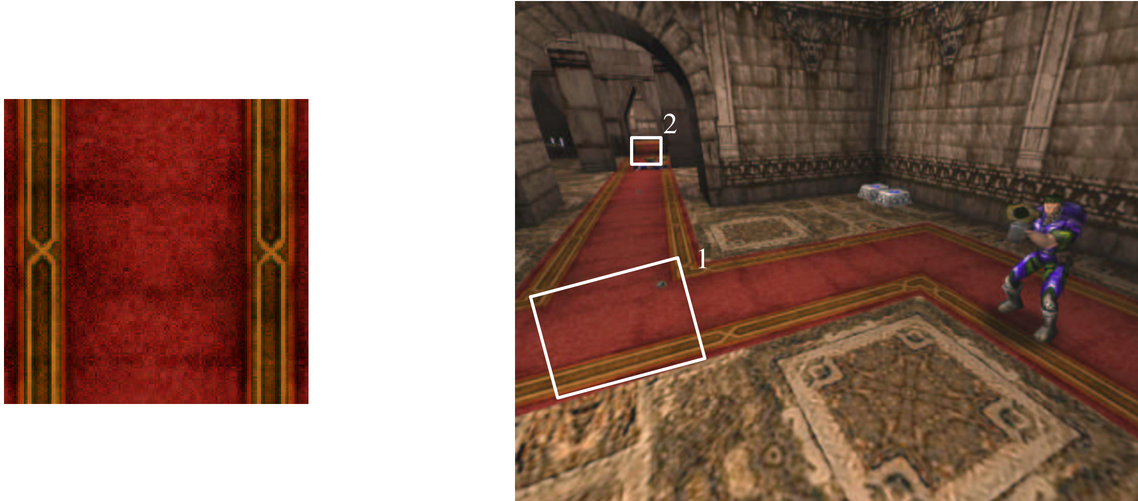


Figure 2.4: The texture of the carpet on the left is used in a scene from a popular game on the right. Box 1 shows an area in the scene where the texels match the pixels approximately one-to-one. For box 2, each pixel maps to many texels in the original texture, taking up more bandwidth and computation resources to compute the final pixel value.

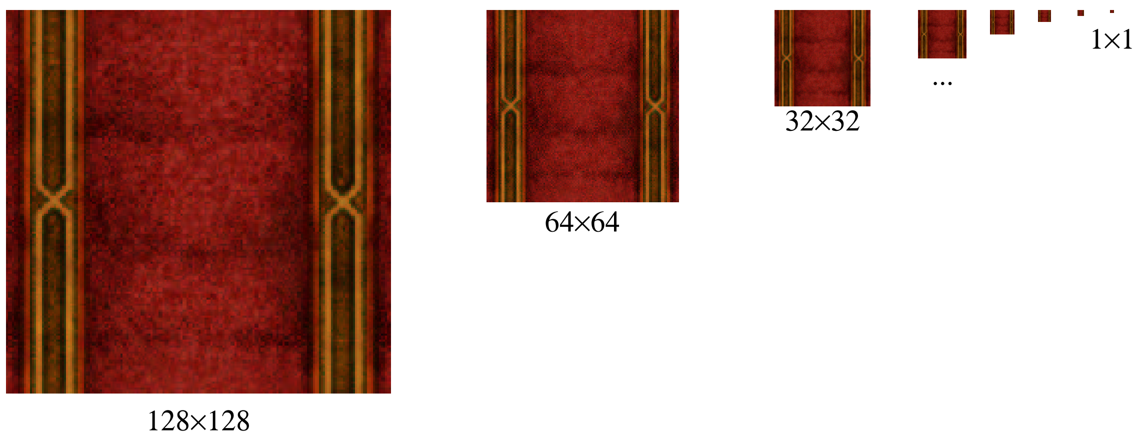
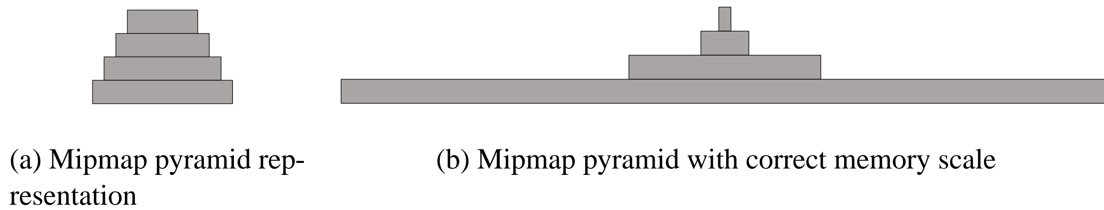


Figure 2.5: The set of mipmap textures for the carpet shown in Figure 2.4. Each level is smaller by a factor of two in both dimensions, all the way to 1×1. While the original 128×128 texture might be the most appropriate for box 1 in Figure 2.4, the 8×8 texture is most appropriate for box 2. Using the smaller mipmap for box 2 saves on both filtering computation and memory bandwidth.



(a) Mipmap pyramid representation

(b) Mipmap pyramid with correct memory scale

Figure 2.6: On the left, the levels of a mipmap have been arranged to resemble a pyramid. On the right, the levels are shown with a correct memory scale.

which is collectively known as a *mipmap* [Williams, 1983]. In a mipmap, each averaged-down image is smaller by a factor of two, all the way down to the smallest mipmap which has a dimension of 1×1 texel. Using this mipmapped representation of a texture allows the averaged-down textures to be computed in any desired fashion, so more care can be taken in producing them than if they were filtered on the fly. Also, we can choose the mipmap level appropriate for a pixel so that only a few texels are needed to compute its value. Because of this, using a mipmapped representation is a constant time operation, which is the primary advantage in using it.

The disadvantage of mipmapping is increased memory cost. Since the levels are geometrically smaller, however, the overhead of the extra levels is only an additional 33% (each level is $1/4$ the size of the previous level). When the mipmap levels are abstractly stacked on top of each other with the smallest resolution levels on top, the structure resembles a pyramid, known as a *mipmap pyramid*, shown in Figure 2.6(a). Figure 2.6(b) shows the same pyramid but in correct memory/resolution scale.

We will use the image on the left throughout this thesis to represent the levels of a mipmap. We will also refer to levels as either "low" or "high" in the pyramid. The "low" and "high" designations refer to the levels' height with respect to other levels as shown in Figure 2.6(a).

When rendering a polygon with a mipmapped texture (like the carpet texture in Figure 2.5), the final pixel values of the polygon are computed by determining the mipmap level on a per pixel basis. The mipmap level is determined by computing the ratio of the

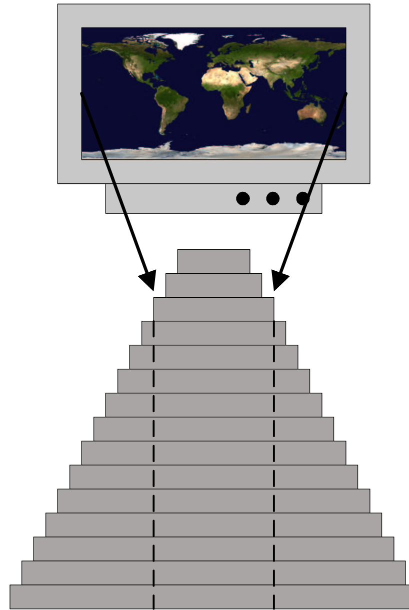


Figure 2.7: A clipmap is a mipmap except that only the required portion of any level is rendered on the display. For extremely large textures, the amount of data "clipped" from the high resolution levels can be enormous. The dotted lines in the figure represent the constant memory cost of displaying a texture, which represents an ever decreasing percentage of the total memory cost of higher resolution levels of the pyramid.

pixel area to the area of texture to be drawn in the texture's base units. Intuitively, selecting the proper mipmap level corresponds to selecting the level that has the closest to a one-to-one ratio of pixel to texel area.

In the general case, a different mipmap level may be appropriate for each pixel 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 VLSI layouts, the same level applies to all pixels.

2.1.5 Clipmapping

Every known platform that supports mipmapping has a limit to the size of the base texture that it can handle, which can range from 256 texels on a side to 16,384 texels on a side. Since VLSI layouts can be many times larger than this, we must consider how to handle a texture that is greater than the hardware limits of any known system.

The clipmap [Tanner et al., 1998] was presented as a solution for viewing arbitrarily large images. A clipmap takes advantage of the fact that although the full mipmap pyramid may be huge, the portion that is currently visible at any one time is bounded and small because of a fixed screen resolution. The example used in the clipmap paper was a 40 million by 20 million, 11 petabyte, texture of the Earth at one meter resolution. This is represented in Figure 2.7 which shows the Earth texture mapped onto a very large mipmap pyramid. While the base texture is 11 petabytes, the smallest levels are only kilobytes or megabytes. When this texture is rendered on a display of 1600×1200 pixels, the max memory cost of three-byte texels to cover the display is around 6 MB. So regardless of the mipmap level necessary, the total amount of texture data needed is very small compared to the immense size of the entire pyramid. The dotted lines in the figure represent the fixed memory cost required regardless of the size of the mipmap level that is appropriate given the viewpoint.

The only downside to using a clipmap is the memory management issues associated with handling an 11 petabyte texture. Real-time interaction requires specially modified hardware and optimized disk-caching techniques.

Since the clipmap has been presented as a way to view arbitrarily large textures, consider using one to view the very large VLSI layout of Intel's McKinley Itanium [Naffziger and Hammond, 2002] processor. The Itanium processor has an area of 421 mm^2 and was designed in a 0.18 μm process on a 0.02 μm grid. This translates into a 2TB image which certainly can be handled by a clipmap. However, we believe that using a clipmap to view such designs is not a feasible solution. Specifically, there are enormous computation and resource costs associated with transforming the design into a 2TB image. The entire design needs to be rasterized and saved to disk. Once this transformation has taken place, the image is now static and not easily changed except by a re-computation of the image. Worse, the size of the image on disk (2TB+) could be many times the size of the design database in its canonical form. This fact will be demonstrated explicitly in the next section.

One of the goals of Chapter 3 is to explain how a clipmap can be modified to exploit the special properties of VLSI layouts so that the transformation cost, in terms of both computation time and disk resources, can be mostly avoided.

2.1.6 Graphics Review Summary

To summarize, the field of computer graphics gives us all of the necessary techniques to achieve our research goals. Using anti-aliasing techniques in rasterization and compositing will result in accurate visualizations. Texture mapping, mipmapping, and clipmapping techniques present a formalized data abstraction that allows real-time interaction given the appropriate graphics hardware. The next two chapters show explicitly how these techniques are used and, in some cases, modified to best suit our specific problems.

2.2 Three Example VLSI Designs

Throughout the rest of this thesis, we will refer to and discuss the three example VLSI designs shown in Figures 2.8, 2.9, and 2.10. These designs were chosen as examples because they represent typical blocks that you might find today. Most modern designs have areas that are non-hierarchical, or flat, containing only standard cells, while other areas are full-custom memories that are dense and hierarchically deep. We have chosen designs that represent both ends of the spectrum. One design is very flat, the other is very hierarchical, and the third is a mix of the two extremes.

Figure 2.10 is the node controller chip from the *Stanford Flash Multiprocessor* [Kuskin et al., 1994], Figure 2.8 is the six-port *Databuffer* memory that was used in the Flash chip, and Figure 2.9 is *SU_Block*, a set of seven student designs that were fabricated together. The Databuffer design is actually part of the Flash design (note the matching empty area in the Flash image), but we will consider them separately because they have very different properties. The other empty areas shown in the Flash design are for proprietary embedded memories, so the layouts are not available.

The Flash design is a 0.50μm standard cell design that was created almost entirely with automated tools. Its register-transfer level description was synthesized into gates and then placed and routed with commercial tools. There was some custom datapath routing and placement done, but the majority was automatic. The Databuffer design, obviously manufactured in the same process, was done entirely by hand with the Magic. SU_Block, manufactured in a 0.25μm process, was done mostly by hand with Magic but also with

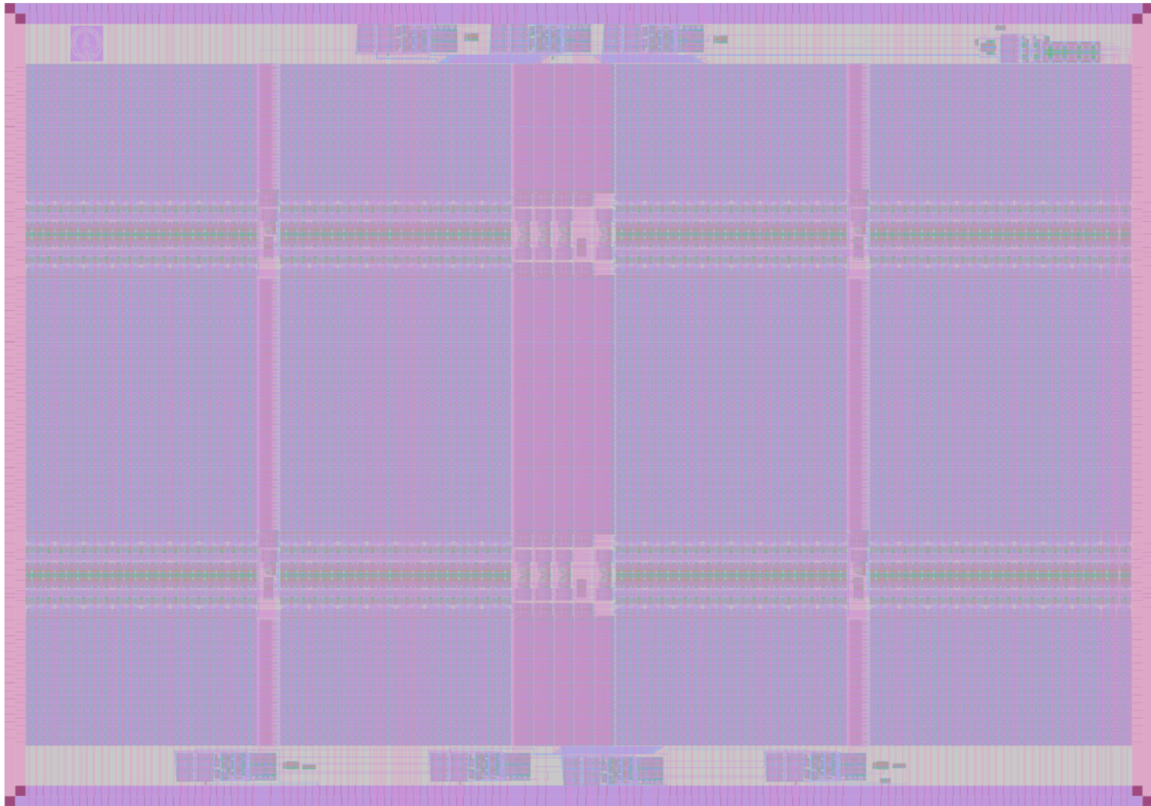


Figure 2.8: The Databuffer design.

some automation for routing. So that all of our test designs could be read by Magic, the Flash design database was converted with a script from the commercial format into the Magic format.

Table 2.1 summarizes the statistics of the three designs, showing that all three have a similar number of total rectangles but widely varying number of unique rectangles. The ratio of total rectangles to unique rectangles tells us how much hierarchy is contained in the design. Flash is a mostly flat design, having only a single layer of hierarchy containing its standard cells. Databuffer is massively hierarchical with a total rectangle to unique rectangle ratio of over 100 to 1. SU_Block is a mix of hierarchical and flat structures with a ratio in between the other two designs.

While the number of metal layers in each design, three, is fairly modest by commercial standards in 2002, the more important data point is the number of Magic *tile types*, shown in the last column. A Magic tile type represents one kind of substance in a design, for

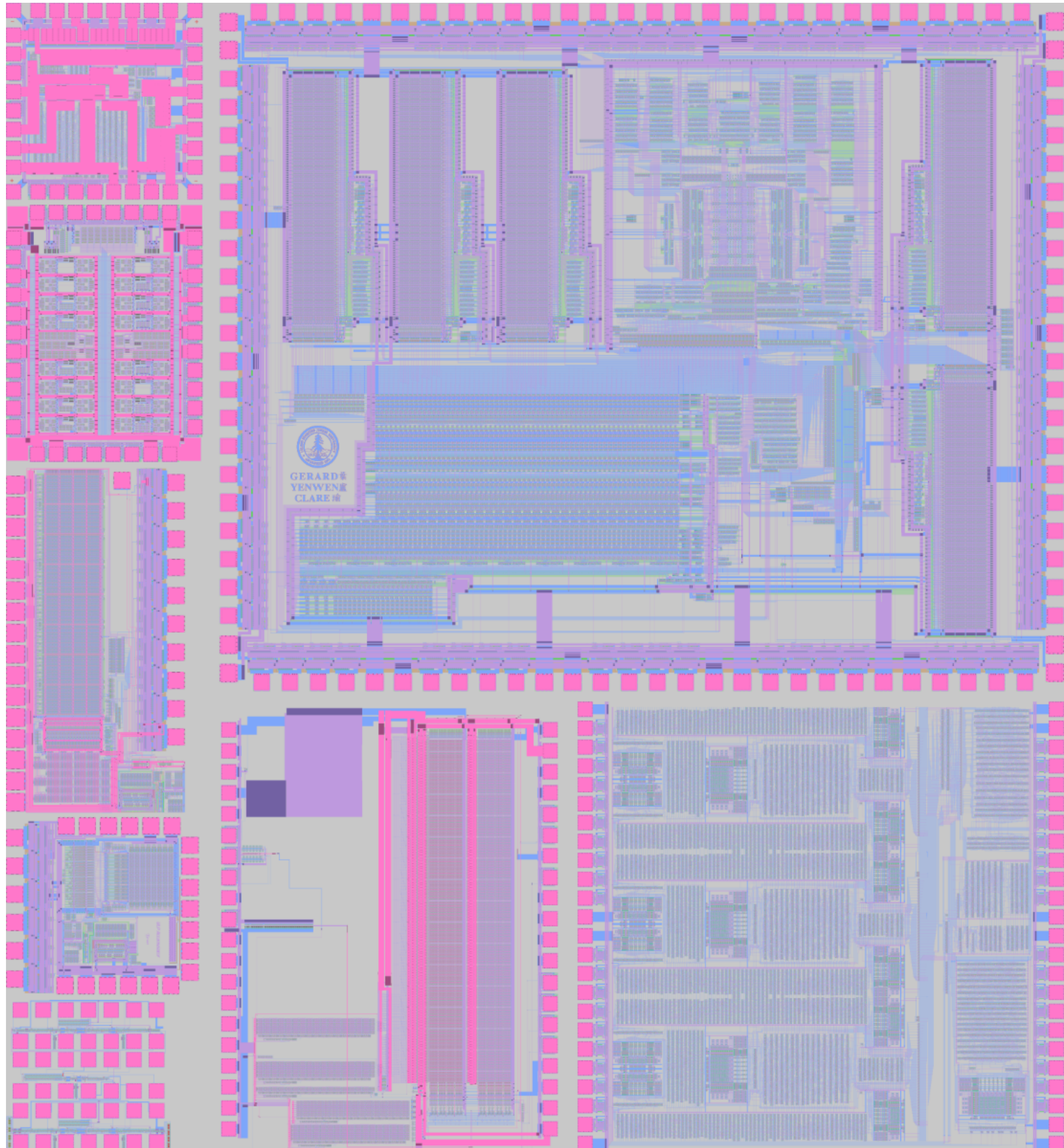


Figure 2.9: The SU_Block design.

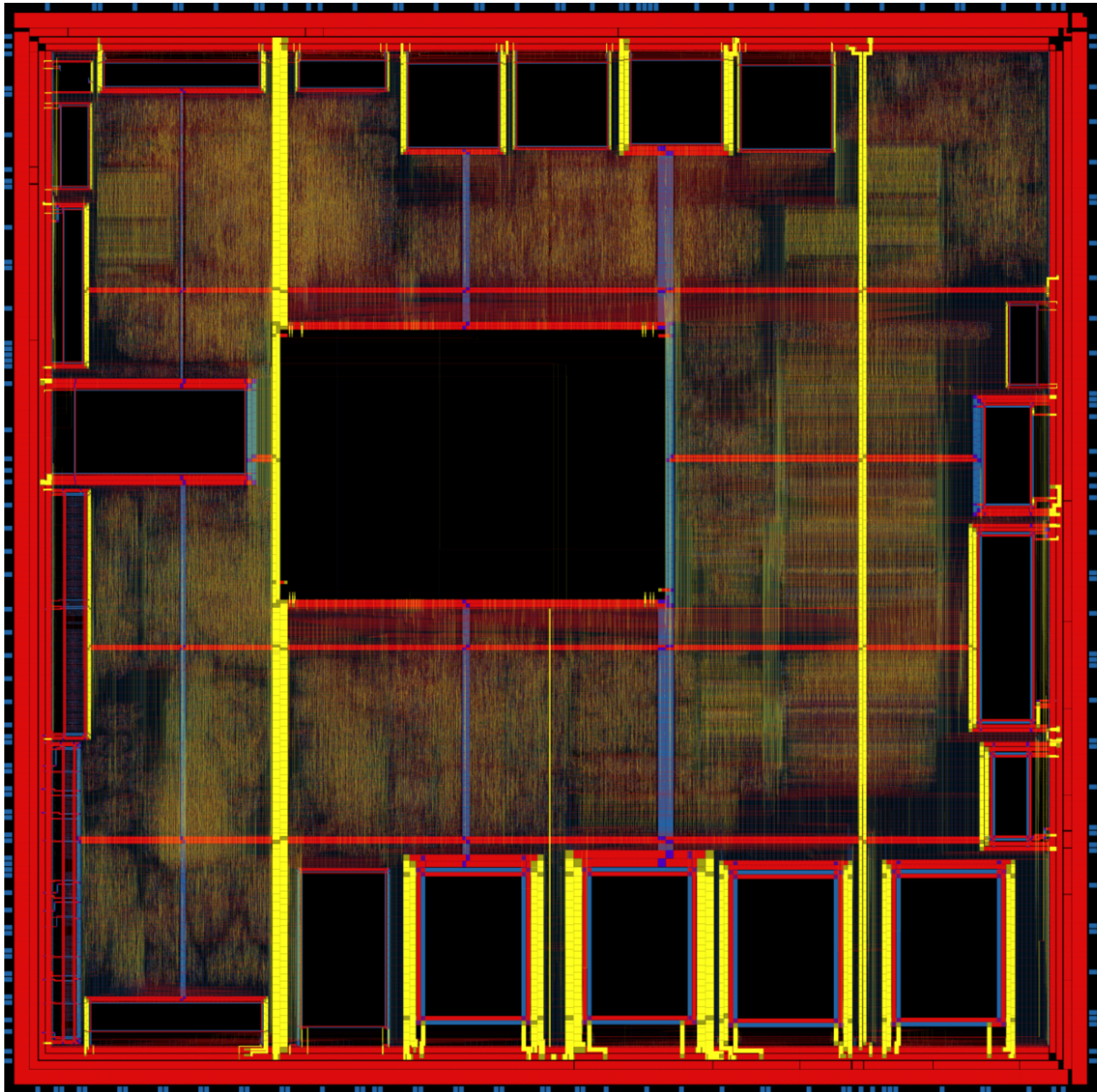


Figure 2.10: The Flash design.

	Databuffer	SU_Block	Flash
Base Dimensions	58 K $\times$ 39 K	100 K $\times$ 109 K	158 K $\times$ 158 K
Total Rectangles	13.0 M	15.6 M	7.0 M
Unique Rectangles	115 K	863 K	5.0 M
Total/Unique Ratio	113	19	1.4
Metal Layers	3	3	3
Tile Types	26	31	6
GDS-II File Size	7 MB	53 MB	319 MB
Clipmap Size	9 GB	44 GB	100 GB

Table 2.1: Statistics for Flash, SU_Block and the Databuffer designs.

example, *polysilicon*, *metal3*, *nwellsubstratecontact*, or *pdiffusion*. Typically, they are each drawn on the display differently, so the more tile types a design has, the more passes required to render on the display.

Finally, compare the GDS-II file size of each design against the size of the image file[1] if each design were converted into a clipmap. The GDS-II file size is roughly three orders of magnitude smaller in each case. As we first mentioned in Section 2.1.5, while it would be possible to view these designs using a pure clipmap solution, the tremendous memory cost makes it unattractive.

The size of the designs that we chose as examples was limited primarily by the memory footprint of the Magic layout system. Magic uses a *Corner-stitched* [Ousterhout, 1984] data structure to hold the layout information which explicitly maintains a record of the empty space between rectangles so that an entire plane of rectangles resembles a quilt made up of what Magic calls "tiles." This data structure is an extremely compute-efficient way to store rectangles for interactive modifications but the explicit empty space tiles incur a 2$\times$ to 3$\times$ overhead in memory cost. We were not able to manipulate truly large designs because large designs would cause Magic to eclipse the 4 GB 32-bit addressing limit. In light of this, we do not claim that these three designs are on the cutting edge for 2002 in any of the metrics lists in Table 2.1, but they are non-trivial designs nonetheless and serve well as benchmarks.

[1]The clipmap size was computed by multiplying the base dimensions by 3 bytes/texel to compute the memory cost of the base image, and then multiplying by $4/3$ to account for the cost of the other levels.

Chapter 3

A Chipmap

Previous chapters discussed existing work in the area of arbitrarily sized image display and manipulation. Most notably, there is a mipmap, which allows economical display of images by precomputing down-filtered views, and a clipmap, which is an improvement upon a mipmap for very large images. We extend this further for the class of images that include VLSI data by introducing the *chipmap*. A chipmap builds on a clipmap in the way that each only requires the texture data that is visible on the display to be computed and in memory at one time. For VLSI designs, this is very important because the designs can be very big. A chipmap is differentiated from a clipmap, however, in a very important way. While a clipmap is used for viewing image data that comes from some source, it does not specify how the data is created. By contrast, a chipmap fundamentally specifies exactly how the image will be generated and displayed at every resolution. As a result, a chipmap not only includes the texture resolution structure of a mipmap and clipmap, it also includes the VLSI database itself, the various caches and the image generation algorithms for the data. Each of these topics will be discussed in detail in this and subsequent chapters.

We will also initially limit our discussion to one particular type of VLSI database, the VLSI layout. As we've stated previously, the canonical VLSI database is the layout itself. We say it's canonical because it is the fundamental database that allows the design to be manufactured. Since the layout is so fundamental and has an actual physical corollary, almost every VLSI design system comes equipped with a tool that views layouts so it is

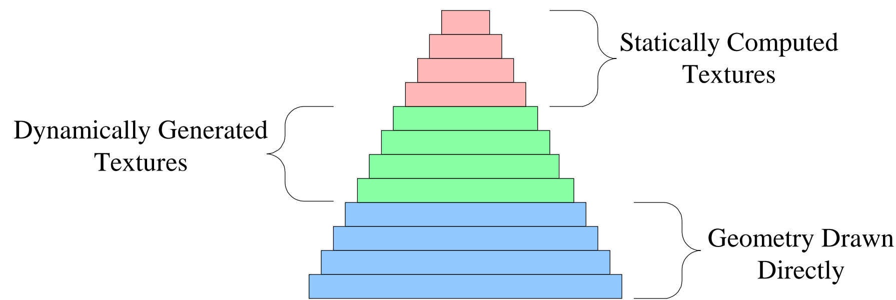


Figure 3.1: A cross-section of a chipmap.

very easy to compare the speed and accuracy of our new methods with pre-existing ones. Chapter 6 shows the results of these comparisons.

For the rest of this chapter and the next, we will concentrate on how a chipmap structure can be used to visualize a VLSI layout. In Chapter 5, we'll discuss how this fundamental structure can be used to visualize other types of VLSI databases.

Lastly, we will discuss a chipmap as if it is being implemented on a platform that supports the OpenGL [Segal and Akeley, 1999] graphics library. Where relevant, it will be noted where implementations that do not use OpenGL (like X Windows) would deviate.

3.1 A Chipmap Pyramid

Figure 3.1 shows a cross-section of a chipmap. This figure builds on Figure 2.7 which shows a similar view of a clipmap. The similarities are that a chipmap will also only load the portions of the mipmap that would be visible at any one time. This is the fundamental difference between a mipmap and a clipmap and it applies to a chipmap as well. However, as we saw in the previous chapter, treating a VLSI layout strictly as an image requires a huge amount of storage space. Figure 3.1 shows that a chipmap is divided into three regions. Each region denotes a level of resolution where the image generation is conducted in a different manner. These three regions are:

1. **Geometry**. Shown in blue. At the lowest levels of the chipmap pyramid, the database can be used directly to draw the layout. The algorithm to draw the layout at these levels is exactly the same as other VLSI layout editors.

2. **Dynamically Generated Textures**. Shown in green. In this region, texture data is dynamically generated and cached as the viewpoint changes. It may need to be re-created again if it is discarded from the cache.

3. **Static Textures**. Shown in red. For this region, the texture data is created in the same manner as the dynamically created data except that it is locked in the cache.

Keep in mind that Figure 3.1 is not drawn to correct resolution scale or memory usage scale; this would require each lower level to be four times as wide as the level above it. We distort the scale to show more levels on the same diagram.

The rest of this chapter is devoted to explaining how the data is managed in each region and how the boundaries between the regions are determined.

3.2 Region 1: Geometry

The first dividing line in Figure 3.1 signifies the boundary between textures and geometry. For levels below the dividing line, the layout is drawn directly to the display from the database; for levels above the line, texture data is used. Basically, as one displays the layout in greater detail, there comes a point when drawing the layout with textures becomes unnecessary for two equally important reasons. First, as one zooms in, the size of the average rectangle is greater than a few pixels in each dimension, so drawing it without anti-aliasing is visually accurate. Second, there are few enough rectangles that it is computationally feasible to draw all of them each frame in a fluid fashion.

Drawing geometry is exactly equivalent to what all current VLSI layout viewers/editors already do. The algorithm is as follows:

1. For each viewpoint, compute the bounding box of the display on the layout

2. Iterate over all the elements visible in that bounding box

3. Using whatever native drawing capabilities exist, draw the elements to the display

4. As the viewpoint changes, go back to step 1

For high detail views, this algorithm is the most preferred because:

1. It requires no extra memory; since the image is created on-the-fly, no extra data structures are required
2. It is the fastest method computationally on both the CPU and GPU; drawing a rectangle that covers many pixels in both x and y as a rectangle is more efficient for both the CPU and GPU than rasterizing it and rendering it as a texture
3. Other details such as stipples and lines can be easily added
4. Since the display is being redrawn from scratch each frame, using this algorithm to implement a layout editor is straightforward

Note that for this algorithm, small designs are equivalent to highly zoomed views of very large designs. The most important thing is the amount of visible data, not the total amount of data.

3.2.1 Texture vs. Geometry Boundary

Determining the level to place the texture/geometry boundary requires that one choose between two factors. The boundary can be placed based either on the number of visible rectangles or on the size of the visible rectangles. While it may seem intuitive to place the boundary based on the number of visible rectangles, we will show that this method is not preferred since it is very dependent on the graphics capabilities of the host platform. The second method, placing the boundary based on the size of the rectangles, is preferred since it is dependent more on human perception, for which it is easier to analyze.

Prioritizing Based on Number of Features

The first way to decide how to divide between geometry and texture is by analyzing the number of layout features that would have to be drawn at any particular resolution. As one zooms out, the number of features grows, taxing both the channel between the CPU and GPU and also the hardware itself as it becomes necessary to draw more and more rectangles.

One must first decide on the threshold number of rectangles that will be the dividing line. Then, any view which contains more than the threshold number of rectangles will be drawn from texture data, while any number that contains fewer will be drawn directly from the layout database. How does one pick this threshold number? Unfortunately, this is extremely system-dependent. A very high-end system might have both a CPU to GPU bandwidth and a rasterization rate which exceeds the capability of a low end system by a factor of 100 or more. This means that trying to have a one-size-fits-all threshold number will be ineffective.

In addition, in VLSI layouts, the number of rectangles in any given area varies with the location of the design. In densely populated areas like memory cells, the number of rectangles could be many factors more than in other areas which contain no rectangles or very few rectangles. Given this, one could either have an adaptive scheme where the dividing line changed over the area of the layout, or one could just choose a number that reflected the highest rectangle density since that would be the most conservative choice given the system's capabilities. Once the threshold number of rectangles has been chosen based on system characteristics, an analysis can run on the target layout to determine at which level would a full-screen image of the layout would never contain more than the threshold number of rectangles.

Finally, we have to consider what would happen if the threshold were high enough such that many of the rectangles appearing on the display were too small to look correct without anti-aliasing measures. In this case, either native anti-aliasing techniques must be used or the threshold must be lowered until it is low enough such that the rectangles drawn directly are large enough not to require anti-aliasing drawing techniques to appear correct. Again, we see that the ability to draw rectangles in an anti-aliased fashion is very system-dependent. On some systems, this would incur no performance penalty, while on others the penalty would be severe (including the lack of ability to draw rectangles in an anti-aliased fashion at all).

Our implementation did not divide texture and geometry based on the number of visible features for the reasons given above. It is too system- and design-dependent, requiring vastly different thresholds given different systems and different locations in the layout.

Level	Average Shorter Dimension
4	1.6 λ
3	3.2 λ
2	6.3 λ
1	12.6 λ
0	25.2 λ

Table 3.1: Average shorter dimensions for a hypothetical VLSI design.

Prioritizing Based on Feature Size

Another way to specify the texture/geometry dividing line is to base the boundary on feature size. Using this criterion eliminates the two problems of using the number of features discussed in the previous section, but it also causes a new problem. If the metric of the boundary is based on feature size, then how does the metric account for the different sizes of the rectangles in an average VLSI layout? To answer this question we say that we will only consider the *shorter* dimension of the rectangles to determine the boundary. While the longer dimension of rectangles in a layout will vary wildly from the minimum allowed in the technology to rectangles that span across the entire layout, the shorter dimension will vary to much less of a degree. Also consider that using the shorter dimension is a requirement for visual fidelity if no anti-aliasing techniques will be used, for it is the shorter dimension that will determine when a rectangle exhibits aliasing artifacts.

So the heuristic used in determining the boundary based on feature size is to average the shorter dimension of every rectangle in the layout and then choose the dividing line so that levels below the line have an average shorter dimension greater than some threshold while levels above it have an average shorter dimension that is less. For example, consider a VLSI design whose rectangles have an average shorter dimension of 25.2 λ. Lambda (λ) represents the underlying units of measurement used in the design. Table 3.1 shows the average shorter dimension for the chipmap of this hypothetical design. The average shorter dimension of the rectangles is smaller by a factor of two for each higher level in the chipmap.

If the boundary threshold were chosen at five pixels, then levels three and above would be rendered as textures while levels two and below would be drawn directly as geometry.

How does one choose a threshold? This decision is based on human perception and assumes that rectangles, when drawn directly, will be rendered in an aliased fashion. At some magnification, the artifacts created by drawing aliased rectangles will be tolerable by the majority of users. The size of rectangles at this magnification should be chosen as the threshold. Aliased rectangles that appear about 100 pixels on the screen might actually be 99 or 101 pixels when drawn in an aliased fashion depending on rounding. This 1% error would not be noticeable to many people, a two pixel rectangle, on the other hand, might actually be one pixels or three pixels which gives a much larger 100% or 50% error, would would be noticeable to most people. We choose a five pixel threshold, which yields about a 25% aliasing error, because it represented the maximum error that we could tolerate.

Finally, consider that even though we have chosen the boundary based on rectangle size, we are still at the mercy of the the host platform's rendering capabilities. We have done nothing to guarantee that any system will be able to render the number of rectangles we will attempt to draw in a real-time fashion. We have only attempted to guarantee that the rectangles, if drawn in an aliased fashion, will appear visually correct. However, in practice this does not turn out to be an issue. Most graphics systems are designed to balance primitive size versus drawing performance. This means that if one covers the display with primitives that are at least a small integer number of pixels in each direction, the resulting performance will be adequate. On lower end systems, the performance might just be a few frames per second, but this is sufficient for VLSI design.

3.2.2 Texture/Geometry Boundary Grid Resolution Invariance

One nice property inherent in a chipmap is that the texture/geometry boundary is invariant to the underlying grid resolution of the design. Consider again the hypothetical design of Table 3.1. Imagine that the technology was redefined to have a grid resolution ten times more fine in each dimension. Now, every rectangle in the layout is ten times larger in relations to the base grid resolution in each dimension although not actually physically larger. Table 3.2 shows the average shorter dimensions for this updated layout.

Now the dividing line between geometry and texture is levels five and six, given the same five pixel threshold. More levels are now drawn as geometry, but nothing else has

Level	Average Smaller Dimension
6	3.9 λ
5	7.9 λ
4	15.8 λ
3	31.5 λ
2	63.0 λ
1	126.0 λ
0	252.0 λ

Table 3.2: Average shorter dimensions for the hypothetical VLSI design of Table 3.1 updated to show the effects of increasing the grid resolution by 10× in each dimension.

changed. The size of the chipmap has grown by three levels but the boundary where textures end and geometry begins will appear to be exactly the same place when rendered on the display.

Note that a transformation like this would cause a 100× increase in memory usage if we were to convert this layout into an image and view it as a clipmap.

3.3 Region 2: Dynamically Generated Texture Data

The second region of Figure 3.1 signifies viewpoints from which the texture data will be dynamically generated directly from the layout database. The rendering strategy is as follows:

1. Given the viewpoint, determine the chipmap pyramid level
2. Lazily create only the texture data that is visible on the display
3. Draw the texture data
4. As the viewpoint changes, go back to step 1

3.3.1 Tiled Texture Pyramid

The second step in the rendering strategy above says that only the visible texture data will be generated. Up to this point, a pyramid level in a mipmap or clipmap has been

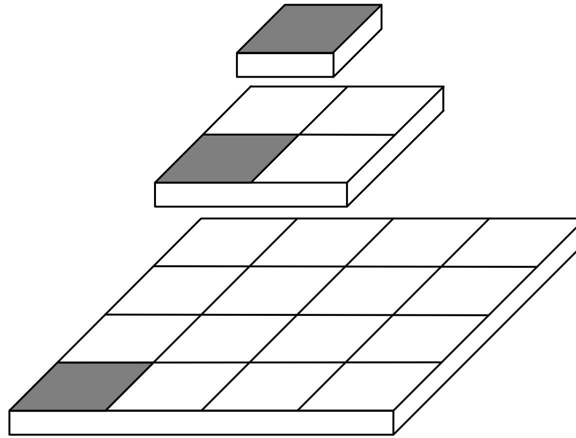


Figure 3.2: The top three levels of a tiled texture pyramid.

thought of as one continuous texture. However, we need a way to think of a pyramid level as something that we can generate in discrete parts. To accomplish this, we use a *tiled texture pyramid* which is shown in Figure 3.2. A tiled texture pyramid is distinguished from a mipmap or clipmap pyramid in that each pyramid level is an array of texture tiles where the size of each of the tiles, in texels, is fixed. Lower levels of the pyramid will simply have more tiles than higher levels, as the tile pyramid shows in Figure 3.2. The texel dimensions of each of the tiles is the same while the total dimensions of each level, in texels, is different. Dividing up the texture pyramid levels in tiles has three important advantages.

First, the tile size is chosen to meet the hardware limits of the host platform. We discussed earlier how all known graphics systems have limits on the size of a single texture that can be used. Using a tiled approach circumvents this issue since any one texture is smaller than the limit. Also, since the tile is the unit of texture that is known to the graphics hardware, there is no requirements that each pyramid level have power of two dimensions. This allows each pyramid level to basically equal the scaled size of the design, modulo to the size of a tile.

Second, a tiled representation is a very simple way of only creating the desired portion of a pyramid level. A clipmap uses a more complex approach (toroidal addressing [Tanner et al., 1998]) that is more efficient in managing massive mipmaps in the general case; in

the special case of VLSI layouts, however, a tiled representation is adequate. The seams between the texture tiles do require special care to ensure visual artifacts between them are not noticeable, but this implementation detail does not detract overall from a tiled approach.

Finally, a tiled representation is a very natural way to quantize texture data generation in a multi-threaded implementation. Later, we will see that texture data generation is the most computationally expensive operation in a chipmap; having multiple processors to accomplish the task can yield linear speedup for N processors over a single processor. A tiled representation is a very natural way to express this parallelism.

Choosing a Tile Size

Choosing the tile size for the texture tiles is the next important step in building a tiled texture pyramid. Larger tile sizes have the advantage of lower overhead per tile, but the disadvantage of possibly having a smaller percentage of overlap with the visible display, causing wasted work. Smaller tile sizes are the exact opposite.

The basic characteristics of the texture tiles are that they will be square and have a dimension that is a power of two. While there is no requirement that they be square, the symmetry of a square tile makes much of the computation easier and there is no advantage to not being square. The tiles should be a power of two in each dimension because most known graphics systems require this to be true.

Additionally, as we have discussed earlier, graphics systems like OpenGL have limits on both the maximum and minimum size of textures. We must, therefore, choose dimensions within that range. All graphics systems that support OpenGL have a minimum dimension of 64 texels on a side while the maximum dimension is system-dependent and can vary between 256 texels on very old systems to 16,384 texels on more modern systems. Implementations built on an OpenGL library will be required to conform to these limits.

The final selection of tile size, in addition to the previous factors, should be based on the maximum display size. The resolution of modern displays can vary from about 800 pixels to about 2000 pixels on a side. A tile size should not approach these values because of the wasted computation done to generate texture data that will not be visible.

We have observed that choosing tile sizes at either extreme can affect the performance of the system negatively. At 64 texels, the extra overhead incurred from processing the

same rectangles that overlap multiple tiles added to the overhead of drawing many, many more textures makes the redraw times slower. At the other extreme, when the tile size is set to approximately the display size, the screen does not refresh as quickly since it only changes as each tile is computed and, on average, a larger total number of texels are computed, many of them not even visible.

In the end, a texture tile size of 256 was chosen as the default size for our implementation. We observed that 256 texels strikes a nice balance between low tile overhead and a low amount of extra computation. However, our implementation allows us to modify the dimension easily should some of the factors that are used to choose it change.

3.3.2 Rendering Texture Data

When rendering the texture data to the display, it is highly improbable that any particular viewpoint will lie exactly at the same resolution as any level of the chipmap pyramid. When a viewpoint lies between levels, texels must either be minified from the level below or magnified from the level above. This is also known as texture *filtering*. Mipmapping theory provides different filtering strategies that trade off image quality and computation time for different situations. In the case of filtering in a chipmap, a different strategy is employed depending on whether the viewpoint is being panned or zoomed.

Filtering Texture Data While Zooming

When the viewpoint zooms, the size of the texels as they appear on the display is incrementally changing every time a frame is drawn. In terms of the chipmap structure, the viewpoint is moving vertically through different levels of resolution, requiring different levels of the pyramid to be rendered. It is desirable to minimize visual artifacts from both the changes in the texel size and the transition from one pyramid level to the next. This is especially important for VLSI layout texture data, which very much tends to be high frequency in nature. Without a careful filtering strategy, the visual popping and shimmering could be distracting.

To minimize visual artifacts to provide the smoothest transitions between frames, our chipmap implementation choose a trilinear filtering strategy for zooming. To accomplish

this, the two surrounding pyramid levels are bilinearly filtered via OpenGL hardware support and then blended together in proportion to their distance from the viewpoint.

Filtering Texture Data While Panning

While zooming requires a computationally intensive filtering strategy to minimize visual artifacts, panning can use bilinear filtering because the size of the texels is not changing from frame to frame, only their location. Bilinear filtering is still required because the texel size will almost never match the pixel display resolution. Additionally, instead of using the nearest pyramid level to the viewpoint, the lower pyramid level is always chosen to give the sharpest image. As with trilinear filtering used with zooming, the hardware is used for bilinear filtering.

3.3.3 Panning and Zooming Without Hardware Support

The previous sections have stated a texture rendering strategy based on the assumption that underlying hardware support exists. However, sometimes it does not and in this case it is still desirable to approximate animation to the best of the host platform's capability. In this case, an implementation could choose to force the viewpoint to lie exactly on a pyramid level and always choose a panning location that had integer coordinates.

Doing these two things would simplify the task of mapping the image data (we now call it image data instead of texture data because we reserve the term ”texture” to mean an image that can be mapped with hardware support). Since no minification or magnification is required, no blending or filtering is required either. Panning is accomplished by mapping the image data directly to the display and zooming changes are forced to jump from one pyramid level to the next.

An implementation could also choose to support arbitrary zoom viewpoints, but then the implementation itself would be responsible for filtering and blending, relying on software to accomplish what is normally done in hardware on today's GPUs, possibly causing a severe degradation in rendering performance.

3.3.4 A Texture Tile Cache

As the viewpoint changes, 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 would grow to the full size of the texture tile pyramid. As was noted previously, the memory footprint of a full tiled texture pyramid for a large size VLSI 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. We defer discussion of possible texture tile replacement policies, including the one that our implementation uses, until we fully explain our rendering architecture in Section 3.5.

Minimum Texture Tile Cache Size

The minimum size of the cache should be set such that the entire display could be covered with texture data using the most aggressive rendering strategy with all the texture data fitting into the cache. Consider a display that has a resolution of 1600×1200 pixels using a trilinear rendering strategy blending two layers of texture data on top of each other. The worst case scenario in terms of texture data quantity would be a viewpoint where the lower pyramid level was being minified nearly $2\times$ in both dimensions, yielding a lower pyramid level texture resolution of 3200×2400. The upper level resolution would be almost one-to-one or 1600×1200. If each texel is three bytes, the minimum size of the texture tile cache given this screen resolution would be about 29 MB.

If the texture tile cache size is not set to at least this minimum, then severe performance degradation will occur if any replacement other than *Most Recently Used* (MRU) is used, since each rendered frame would cause intense thrashing in the cache.

Maximum Texture Tile Cache Size

The maximum tile cache size should be bounded by the amount of main memory on the host platform because it is generally the case that the time spent recomputing a texture tile

is less than the time needed to swap a computed texture tile from disk. This maximum, however, is only a theoretical maximum. Practically, the cache need not be much larger than the minimum because of the way that the chipmap renders the display. As we will describe in Section 3.5, even when the texture data is not in the cache, the display can be updated in a way to give the user a visual cue of the progress.

3.3.5 Managing Dedicated Graphics Texture Memory

When a chipmap implementation is written to take advantage of hardware-accelerated texture mapping, it raises issues with managing the dedicated texture memory that exists on the majority of such hardware. Typically, the amount of dedicated memory in these systems is an order of magnitude less than the amount of main memory. If an implementation chooses to size the texture tile cache larger than the amount of texture memory, it is usually advantageous to decouple the texture tile cache from the dedicated texture memory by having an additional cache that will manage the dedicated texture memory. This cache of "texture objects" gives a chipmap implementation a way to manage the dedicated texture memory explicitly.

On some graphics system platforms, it has been observed that a "texture spill," or what happens when there is a capacity conflict on the graphics hardware itself, has very bad performance ramifications. These spills can be avoided altogether with the texture object cache. Once this cache is in place, the graphics system sees that the same portions of texture memory are being used over and over again with no capacity conflicts.

Compressed Textures

It has been suggested that using *compressed textures* would be a way to increase the utility of the dedicated texture memory. Texture compression is a scheme that compresses the texture data before it is sent to the graphics hardware. Once it is compressed, it requires less CPU to GPU bandwidth to transmit and less dedicated graphics memory to store. It can be a big win for applications that pre-create all of their texture data. However, for a chipmap, where all the texture data is dynamically generated, a $2\times$ to $4\times$ overhead is

incurred in the texture compression stage. This overhead erases any benefit that would be gained in saved bandwidth or storage space.

One type of texture compression does have benefits, however, because the computation cost is so low. While the texture data created on the host platform is nominally three bytes/texel (24-bits), it can be stored on the graphics hardware as only two bytes/texel, also known as “16-bit textures.” Transforming 24-bit textures to 16-bit textures is computationally simple and incurs little overhead at the expense of two or three bits of lost color precision. This loss of precision is generally tolerable and will greatly enhance an implementation’s usability on graphics systems with a scant amount of dedicated texture memory.

3.4 Region 3: Static Texture Data

The last region of the chipmap shown in Figure 3.1 is the static texture data portion that comprises the highest levels of the pyramid. A very common operation for a VLSI layout viewer is showing the entire design. Users typically always start from a full view and then zoom in on an area of interest. A new area of interest will usually be chosen by returning first to a full view of the design. Consequently, viewing the entire design should always be fast.

Full screen views will be optimally fast to redraw when the appropriate texture data already exists by keeping it separate from the general texture tile cache discussed in Section 3.3.4. Then any time a full design view point is requested, the data will always be available and the redraw time will be nearly instantaneous.

3.4.1 Static/Dynamic Texture Boundary

There is no hard and fast rule on how set the boundary between the static and dynamic portions of the chipmap pyramid. Our implementation chose to set the static boundary at the level of the pyramid that would be used if the full design were viewed on a window the size of the display. We also include all levels above this level because they are only an additional 33% memory cost. Section 6.4 on Page 89 shows how the size of this section

can be up to a few dozen megabytes on a 1600×1200 display. We chose this level so that full views would always be fast, but implementations can choose different levels to trade-off memory cost and redisplay time.

At one extreme, an implementation could choose to have no static portion at all. In the next section, we'll explain why this is not recommended.

While at the other extreme, an implementation could permanently compute the entire chipmap. In this case, no texture data generation is required except at startup, at the cost of potentially enormous amounts of memory. Note that this scenario effectively transforms a chipmap back into a clipmap.

3.5 Rendering Architecture

There should be no limit to the speed at which users are allowed to change the viewpoint. At any moment, the user should have the ability to move to an arbitrary viewpoint. Given this, what should be rendered on the display when the viewpoint changes too fast for all of the appropriate texture tiles to be created? One possible solution is to delay redraw until all texture tiles have been created. While this ensures optimum visual fidelity, it totally undermines tool responsiveness, creating an atmosphere of user frustration. At the outset of this thesis, one of the stated goals was a visualization system that provided the best user experience. This means absolute responsiveness regardless of the speed of viewpoint changes.

To meet this goal, we employ a multi-threaded rendering architecture: One "drawing" thread renders the texture tiles on the display while one or more "worker" threads create the tiles. The number of worker threads is usually set to the number of processors on the host platform. When the drawing thread does not have access to all of the texture tiles, 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. This scheme exposes another benefit of the static portion of the chipmap discussed in Section 3.4. Because these tiles are guaranteed to exist, *some* coarser view of the desired area will always be available. The

blurriness acts as a visual cue of the overall progress to the user, and is not a distraction since tool responsiveness is always instantaneous.

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.

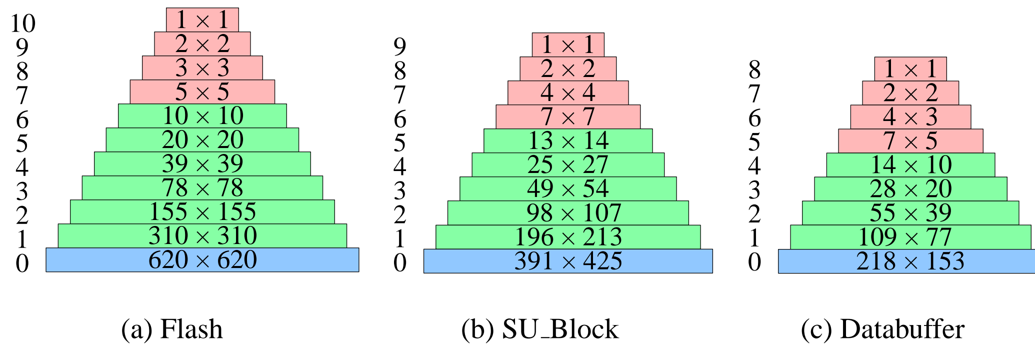
Having now described our rendering architecture, let us now revisit two topics that we deferred discussion from earlier in this chapter; possible texture tile replacement policies, and the placement of the static/dynamic boundary in the chipmap pyramid.

There are many possible replacement policies that one could use in selecting a suitable tile to evict from the texture tile cache when necessary. Since viewpoint changes exhibit spatial locality, one possible policy evicts the tile farthest away from the current viewpoint. We have found, however, that a *Least Recently Used* (LRU) policy provides good enough performance so that other, more exotic, policies are not necessary. Generally, the LRU tile will be far away from the current viewpoint, but more importantly, our rendering architecture, since it guarantees that some data will always be drawn, hides the delay of not having all of the needed texture tiles.

For the rendering architecture to be effective, at least one level must exist in the static portion of the chipmap pyramid. If there is no static portion, then some fast viewpoint changes may cause areas of the display to be vacant of data since *no* coarser view of the design exists. So we advise that the static portion of pyramid no only provides a benefit, but is a necessity if the display is always to be completely covered with data.

3.6 Example Chipmaps

Having laid out the groundwork for the chipmap structure, we now present the chipmaps (Figure 3.3) for the three benchmark designs introduced earlier. The colors delineate the regions as in Figure 3.1. The figures shown on the chipmaps are the dimensions of the texture tile grids for each level. Using a texture tiled grid has allowed the total size of each level to approximately match the scaled size of the designs. These dimensions are also shown in the figure. The pre-computed/dynamic texture boundary was computed assuming



(a) Flash (b) SU_Block (c) Databuffer

Figure 3.3: The chipmap structures for the three example designs.

a screen resolution of 1600×1200. All three designs have only one level in the geometry region, because each design has rectangles with an average shorter dimension mostly near the grid resolution. Readers should not get the impression that having a single level in the geometry region is always guaranteed. Throughout the course of our research, we tested nearly a dozen other designs, some having up to four levels of the chipmap pyramid devoted to the geometry region. These three designs, because they were hand designed with Magic (or, as in the case of the Flash design, converted to the same scale of the Databuffer design) have a single level of geometry because manual designs tend to have a minimum dimension at or near the grid resolution.

3.6.1 Viewing Very Large Designs

The preceding example show chipmaps of varying sizes. Earlier we discussed Magic's limitations, and correspondingly our limitations, with regard to viewing the largest designs that exist in 2002 with our test implementation. Let us now speculate on the chipmap for one such design.

Figure 3.4 is the approximate chipmap for Intel's McKinley design that we introduced earlier. We will demonstrate in Chapter 4 that the time to generate a texture tile is strictly dependent on the amount of data contained in a tile, with tiles at the highest levels taking the longest because they, by definition, contain the most data. For McKinley, the amount of time required to create a viewpoint high in the pyramid will simply be longer than any

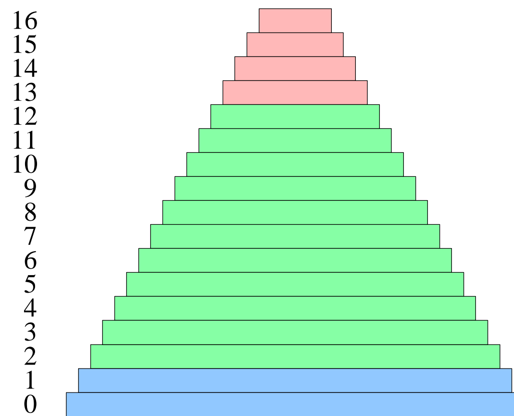


Figure 3.4: Intel's McKinley design as a chipmap. If we assume two levels of geometry, the equivalent clipmap size for this design is 25 TB.

design that we have currently tested. Besides this extra time, there is no fundamental limitation to the size of design that can be viewed with a chipmap. A system's ability to display McKinley is more dependent on its overall system capacity than its graphical capability since the size of the internal database may require main memory not only in excess of the 4 GB 32-bit memory limit, but also an amount practically unreasonable for today's high-end 64-bit workstations. Ultimately, this memory requirement effects all applications that wish to operate on a design so large.

3.7 Summary

This chapter introduced the concept of a chipmap. With this structure, we can efficiently navigate an arbitrarily sized VLSI layout. At high zoom, the geometry is directly drawn, taking advantage of the fact that it is computationally and visually feasible. At the lowest resolutions, we permanently keep the texture data so that drawing a full screen view, a common operation, is always instantaneous. And in between these regions, the texture data is dynamically generated as the viewpoint changes, but the memory consumption is constant, reasonable and independent of the size of the design. The next chapter shows how the characteristics of VLSI design can be leveraged to improve the quality and speed of texture generation.

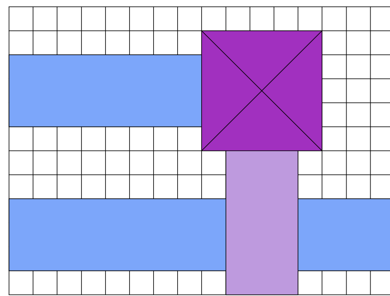
Chapter 4

Image Generation

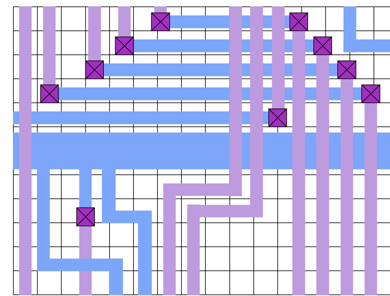
Earlier we stated that the fast and accurate dynamic generation of texture data was a key to the success of chipmap as a visualization tool. Chapter 3 laid the groundwork for the structure that houses the data, allowing it to be rendered on the display in a timely and memory efficient manner. This chapter focuses on explaining how the image data is created to accurately represent the layout data at any resolution. The basic problem we are solving is that of sub-pixel resolution, or how should we model rectangles that are shorter than one pixel in one or more dimensions. By modeling this correctly, we help accomplish one of our stated goals: accurate visualizations.

This chapter is divided into four parts. First, we consider techniques to rasterize the sub-pixel geometry in an anti-aliased fashion. We focus on developing a technique that is accurate but also has a low memory overhead and is comparably fast. Second, we describe how the output of rasterization is composited together in different ways, depending on specific situations. Here we use the special properties of VLSI designs to give us more accurate results than might otherwise be possible. Third, we describe two ways in which we rasterize and composite the geometry and explain in which scenario each one is most appropriate. Fourth, we discuss how explicitly instantiated hierarchy can be pre-rasterized at start-up, speeding up on-the-fly image creation.

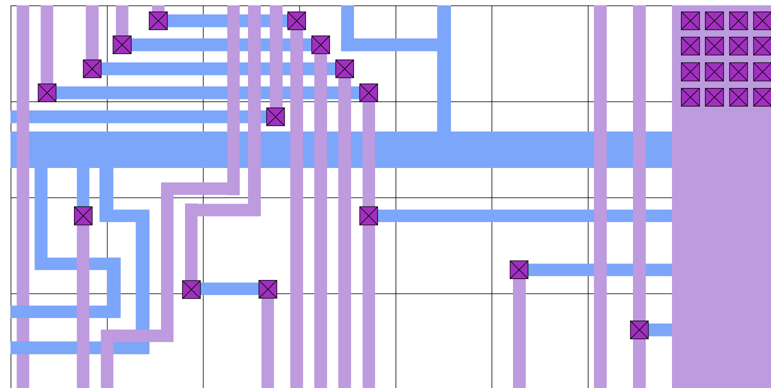
For the rest of the chapter, we will refer to the data as texture data, as opposed to image data, since we will focus on using the data in an environment with texture mapping capability. For the same reason, we will talk about the individual data elements as texels



(a) Rectangles larger than texels.



(b) Rectangles slightly smaller than texels



(c) Rectangles much smaller than texels.

Figure 4.1: Figure 4.1(a) shows rectangles at greater than the given texel resolution. Figure 4.1(b) and figure 4.1(c) show rectangles increasingly smaller in relation to texel size.

rather than pixels. However, the data generated could be used as images or on platforms that do not explicitly support texture mapping.

4.1 Anti-aliasing Techniques

At each higher level of a chipmap pyramid, the texture tiles, and the texels that make up those tiles, correspond to more layout area than the levels below. Figure 4.1 shows three examples of texel grid superimposed on a representative layout at different resolutions.

In figure 4.1(a), the layout rectangles are all nicely aligned with the texels. Rasterizing the layout information into texel information in this case is trivial. Either a texel is covered

by a rectangle or it is not. The other cases, however, are not so simple. In figure 4.1(b), the texels are only partially covered by a rectangle and in figure 4.1(c), each texel is covered by many rectangles, some representing different layers. Yet, each texel will ultimately be a single color.

Consider that all other known layout viewers take no measures to accurately anti-alias layout information in this situation. This leads to the massive visual errors we discussed in the introduction. The problem now before us – the key to visual fidelity – is to accurately compute texel values given that an arbitrary amount of information may lie within their boundaries, a process we described as anti-aliasing in Chapter 2.

In this section we explore three possible ways to compute the texel data accurately. For each method, we consider creating a texture tile on a level of the chipmap pyramid that contains rectangles with sub-texel dimensions similar to Figure 4.1(c). Clearly, it would be desirable to have an anti-aliasing technique that creates perfectly accurate texel values while using minimal resources (memory, computation time). We will see that it is necessary, however, to trade-off between accuracy and resource cost in order create texels in a timely fashion.

The first technique we consider produces the most accurate texel values but at an unacceptable computation and memory cost. The second technique has an initially feasible accuracy/cost trade-off, but ultimately it must be discarded because in order to be useful it must be used in a such a way that it degenerates into the first technique. Ultimately, we decide on the third technique, which is fast, has a very low memory overhead, and though not as accurate as the first technique, is accurate enough for our requirements.

4.1.1 Level Zero Rasterization & Averaging

Perhaps the easiest way to solve the problem of anti-aliased rectangles is to avoid it altogether. An arbitrary tile in the chipmap pyramid can be computed by rasterizing the equivalent layout area at a resolution where no sub-texel resolution occurs, similar to the case shown in Figure 4.1(a). After the rasterization is complete, the resultant texels are averaged down to the size of the texture tile. This technique incurs a high computation

cost of averaging down the huge, high resolution texture to the small texture tile and a high memory cost of creating, storing, and touching the high resolution texture.

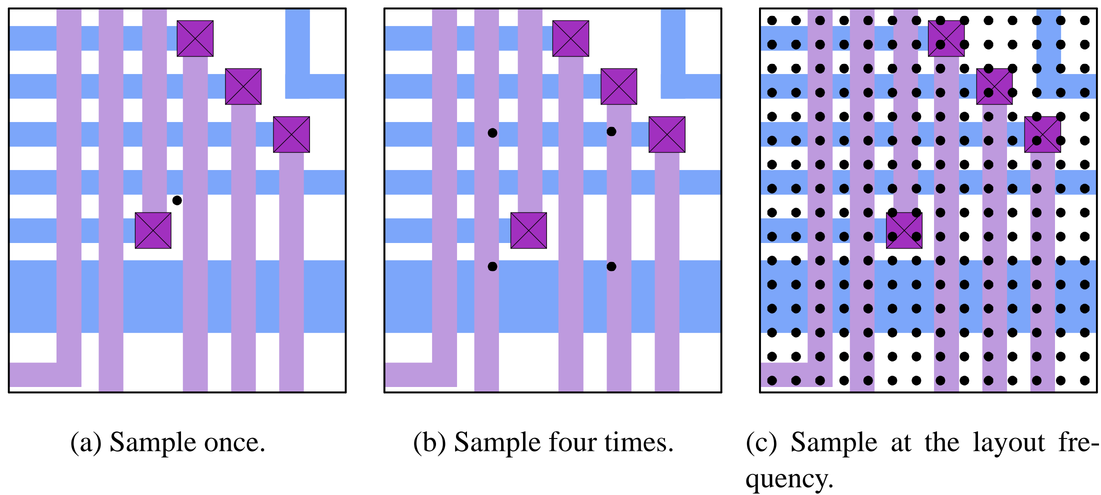
While this method is simple and produces very accurate texel values, we generally reject it because of the enormous cost of touching the required memory. A texture tile high up in the pyramid may represent a huge amount of memory when rasterized at pyramid level zero. Even if the cost of rasterization and down-sampling were zero, the time it would take to touch all the memory precludes this technique from being practical for real-time use.

Consider a texture tile of dimension 256×256 on level seven (a reasonable number of levels for a large design) of a chipmap. That would map to a texture of size $32,768 \times 32,768$ at level zero. If each texel were three bytes, this would translate into a 3 GB texture! Worse, this cost would be duplicated for each tile visible on the display (a display resolution of 1024×1024 would require 48 GB of data). The memory bandwidth and footprint requirements are too severe to be feasible.

In conclusion, while rasterizing at level zero and averaging has the advantage of creating very accurate texels, it essentially converts a chipmap back into a clipmap, incurring a huge memory resource cost and making it impractical for real-time use.

4.1.2 Point Sampling

A very common anti-aliasing technique in computer graphics is *point sampling*. Point sampling involves testing the value of the geometry at some number of points to determine the image value. In our case, point sampling would test the value of the VLSI database at some frequency to determine the texel values. Sampling theory tells us that in order to adequately characterize the image, we must sample at an appropriately high frequency. When that frequency requires more than one sample per texel to be taken, it is known as *super sampling*. We can reexamine Figure 4.1 and see that the minimum frequency required for each of the three magnifications is very different. While one sample per texel would be adequate in Figure 4.1(a), perhaps two would be needed in Figure 4.1(b), while many would be needed in Figure 4.1(c).



(a) Sample once. (b) Sample four times. (c) Sample at the layout frequency.

Figure 4.2: The same view of a VLSI layout contained within a single texel on a texture tile high in the pyramid is drawn three times with sample grids of differing frequencies. The black dots are the sample points.

In Figure 4.2, we show three times same portion of a VLSI layout representing one texel of a texture tile high up in a chipmap pyramid. Over each view, we have superimposed a sample grid of varying frequency.

Figure 4.2(a) shows the simplest sampling pattern, one sample. It is obvious that one sample would produce not only an incorrect value, but possibly a misleading value the majority of the time.

Figure 4.2(b) shows sampling at a higher frequency. We have missed some rectangles altogether and our samples will not accurately represent the composition of the geometry. While the amount of error when compared to Figure 4.2(a) is less, the final result is still likely to be misleading.

To accurately sample this texel, the dense sample pattern shown in Figure 4.2(c) is required. Unfortunately, this situation is no different than the level-zero rasterization technique discussed in Section 4.1.1. In fact, level-zero rasterization is just a special case of point sampling where the sampling frequency matches the data frequency.

Point sampling presents a speed/accuracy trade-off. To maintain a constant amount of error regardless of resolution, the number of samples must be increased as one traverses higher levels of the pyramid – with a huge computation and memory cost for the highest

levels. On the other hand, a constant computation cost can be had if one is willing to accept an increasing amount of error at higher resolutions.

In conclusion, point sampling is rejected as an anti-aliasing technique because under-sampling would cause some rectangles to be skipped, while appropriate sampling would degenerate into the same memory cost problem as level-zero rasterization, again transforming the chipmap into a clipmap.

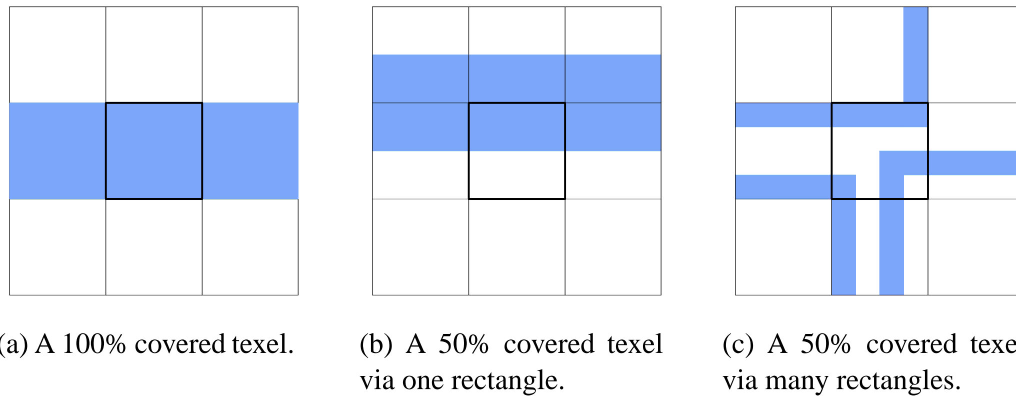
4.1.3 Area Sampling

The last anti-aliasing technique that we will consider is *area sampling*. With area sampling, each rectangle will contribute to a texel's final value in some proportion to the amount that it overlaps the texel.

Computer graphics theory defines different types of area sampling with accuracy/speed trade-offs. There are two factors that effect this trade-off: how one chooses to model the shape of the texel, and whether the overlap contribution is weighted or not. The simplest filter shape is a square because computing bounding-box overlaps with squares is computationally easy, although not the most accurate since texels are not really shaped like squares [Smith, 1995]. More complex filter shapes, like circles, produce more accurate results, but require far more computation. The simplest type of overlap contribution technique is *unweighted area sampling* in which the overlap contribution is independent of the distance of the overlap from the texel's center. Alternatively, *weighted area sampling* weighs the contribution so that overlap near the texel's center contributes more than overlap at the texel's edge.

The combination of using square texels with unweighted area sampling is known as a *box filter* since the shape of the filter, considering both the texel shape and how overlap contributes to the texel's value, is a box. Contrast this to the cone shape of a circular filter in combination with weighted area sampling. Interested readers can see [Foley et al., 1996], Chapter 14.10 for more information on other more advanced filtering techniques.

Our implementation used box filter to determine coverage values because of its computational simplicity and low perceived error. Section 6.2 on Page 80 shows that our final



(a) A 100% covered texel. (b) A 50% covered texel via one rectangle. (c) A 50% covered texel via many rectangles.

Figure 4.3: The middle texel is covered in three different ways. In Figure 4.3(a), it is fully covered giving a trivial coverage value of 1.0. The other two figures are 50% covered via different means.

image quality was very good using a box filter. Let us now describe a box filter in more detail.

Figure 4.3 shows three different resolutions where the middle texel is covered by rectangles to varying degrees. Figure 4.3(a) is a familiar view of a texel that is fully covered by a rectangle. The coverage value in this situation is 1.0. This case is equivalent to point sampling at pyramid level zero. Figure 4.3(b) shows a texel with 50% coverage contributed by a single rectangle while Figure 4.3(c) shows a texel also with 50% coverage; this value, however, is contributed by summing the contributions of multiple individual rectangles.

We find many favorable qualities to area sampling as compared with level-zero rasterization and point sampling. First and foremost is that the memory requirements of area sampling are small. While point sampling degenerates into level-zero rasterization to avoid missing any rectangles, area sampling will never miss any rectangles. With area sampling, the time to create a texture tile is totally dependent on the number of rectangles that overlap it and the cost of touching the associated memory is small.

When using area sampling to anti-alias rectangles of different colors that will overlap and occlude each other, it is necessary to render them in a sorted front-to-back ordering to achieve correct results. This requirement is a potential downside of area sampling since rectangle sorting may incur a large performance penalty. Fortunately, VLSI layouts have inherent properties that eliminate this problem because they are already stored in a sorted

fashion requiring little or no cost to render them from front-to-back. Also, as we will see, since we perform the rasterization on the CPU instead of the GPU, we can use our own data structures to eliminate the sorted requirement altogether.

4.1.4 Rasterization

We now use a unweighted area sampling to turn a rectangle with sub-texel dimensions into fragments. The basic idea is to scale the coordinates of the rectangle by an amount equal to 2^{level} where *level* is the current level of the chipmap pyramid. For example, if we are computing a texture tile on level 5 of the chipmap pyramid, then the coordinates of the rectangles would be divided by 2^5 or 32. After the scaled rectangle coordinates have been computed, area overlap tests are performed for each affected texel to determine the fragment values.

A key point needs to be made here. Notice that when scaling rectangle coordinates, we will always be dividing by a power of two since pyramid levels differ by a factor of two in each dimension from level to level. It is critically important to realize that it is a fundamental property of a mipmap that allows us to divide by two, a computationally trivial task (simply a right shift), rather than being forced to divide by some other number. Mipmaps were invented so that some level would always more or less correspond on a one-to-one texel space to pixel space basis. We take advantage of the fact that scaling coordinates to mipmap levels is trivial because each level is some power of two different in size than the base level.

4.2 Compositing

Now that we have chosen unweighted area sampling as our anti-aliasing technique and described how rasterization will convert these samples into fragments, we will describe the algorithms used to combine these fragments into texel values. Final texel values will carry only color (RGB) information; Since the fragments will mostly be sub-texel in resolution, however, we have to create an additional field for every texel that tracks the amount that they contribute.

So let us define *coverage* to be the amount that a fragment contributes to a texel. A fragment's coverage value is bounded between 0.0 and 1.0 inclusive. The value 0.0 signifies that the fragment does not touch the texel and the value 1.0 signifies that the fragment fully covers the texel.

The steps of creating the texel values in a texture tile are as follows. Each rectangle is rasterized using the unweighted area sampling techniques to generate fragments. These fragments are combined in some manner to the texels they affect. Finally, we use the coverage values to modulate and compute the final full color (RGB) value of the texels.

While computer graphics literature defines many ways to combine fragments, the fragments generated from VLSI rectangles have special properties that allow them to be combined in ways that maximize visual fidelity and minimize perceived errors.

4.2.1 Coverage Equals Transparency

The idea of coverage has a mathematical equivalence to the common graphics idiom *alpha (α)*, which represents transparency. A rectangle that overlaps a texel by 50% will have a coverage value of 0.5. Equivalently, a rectangle that completely covers a texel but is represented with 50% transparency will also have a 0.5 coverage value. Throughout the rest of this thesis, we use the symbol α to represent coverage and we use the terms alpha and coverage interchangeably.

Because of this equivalence, it is trivial to represent layers transparently. In addition, since our area sampling anti-aliasing technique already requires that we keep track of coverage, no additional computation cost is incurred.

4.2.2 Different Layer Compositing

In combining fragments from different layers, thus different colors, we assume *arbitrary* overlap. This behavior is captured in Equation 4.1:

$$\alpha'_{src} = (1.0 - \alpha_{dst})\alpha_{src} \tag{4.1}$$

α'_{src} is the resulting amount of contribution given that the texel is already covered by α_{dst} and is now overlapped by the incoming fragment by α_{src}. If the incoming fragment completely covers the texel (i.e.. α_{src} equals 1.0), then the contribution will be the uncovered portion of the texel, given by $(1.0 - \alpha_{dst})$. Remember that we must allow α_{dst} to occlude α_{src} for correct front-to-back anti-aliased compositing.

The arbitrary overlap assumption is the only assumption one can make if no other special information is known about the relationship of the geometry. This assumption has been the standard in computer graphics for independent geometry since it was put forth by [Porter and Duff, 1984].

4.2.3 Same Layer Compositing

While we do not have any special information about how different layer rectangles are aligned to help us composite them, same layer rectangles may have such information because of the way that the data is stored. In particular, Magic's corner-stitched data structure guarantees that same layer rectangles within a subcell will not overlap. The only way that same layer rectangle can overlap in Magic is to have different subcells overlap. Each design is different and pathological cases can be constructed, but in general we have found that most same layer rectangles do not overlap in Magic. Our three designs, for example, as a percentage of rectangle area, have a same layer overlap of 6.9%, 14.7%, and 0.06% for Databuffer, SU_Block, and Flash respectively.

These numbers tell us that the assumption of non-overlap will produce the most accurate images for these designs, however, with other database formats it could make more sense to assume other properties about how the rectangles overlap.

$$\alpha'_{src} = min(1.0 - \alpha_{dst}, \alpha_{src}) \tag{4.2}$$

Equation 4.2 shows that a same layer fragment will contribute the maximum amount possible given the amount of the texel not already covered[1]. α'_{src} is the amount that the incoming fragment, α_{src}, contributes to the new texel and $(1.0-\alpha_{dst})$ is the amount uncovered.

In the worst case, if two fragments completely overlapped, Equation 4.2 would be wrong by 50%. On the other hand, if we used Equation 4.1 for same layer compositing, the worst case error would be 25%. Why have we chosen then to use a compositing equation that would give a maximum error that is twice as large? It is far more likely for same layer rectangles to be exclusive of each other. So while Equation 4.2 gives a large maximum error, it produces texel values closer to the correct value a greater percentage of the time given that same-layer geometry does not actually overlap that often.

4.2.4 Color Compositing

We now define how α'_{src}, the ultimate contribution made by a fragment, is used to generate the full color value of the texel. Texels are made up of four channels, RGB color channels, and α, alpha, the coverage or transparency. Together, we have a color, C. Given a texel is a color, C_{dst}, and the incoming fragment represents a layer with color, C_{src} which has a coverage value of α'_{src}, we can blend the colors together with Equation 4.3:

$$C_{dst} = C_{dst} + C_{src}\alpha'_{src} \tag{4.3}$$

This equation is the equivalent to the standard computer graphics blending relation with the destination's blend factor set to 1.0. Since α'_{src} was derived from α_{dst}, their sum can never be greater than 1.0 which means the equation will never overflow. Also notice that the alpha channel of the source color could itself have an independent value that would represent the layer's transparency separate from the coverage. As we noted earlier, since transparency and coverage are mathematically equivalent, they simply multiply together.

[1]Dependent geometry is not new in computer graphics. Most 3-D models are composed of triangles that are perfectly aligned with each other so that they appear solid. When rasterizing these models, it is important to composite the seams between the triangles correctly or visually disturbing artifacts will result. Equation 4.2 fits the bill. In fact, readers familiar with OpenGL will recognize Equation 4.2 as the blending factor `GL_SRC_ALPHA_SATURATE`.

4.3 Texture Tile Creation Strategies

We have now selected our anti-aliasing technique and determined how we will composite fragments from different layers and same layers. We have explained how transparency is added to the compositing process and how we arrive at color texel values. This section draws all these techniques together and describes two techniques to actually populate a texture tile from a chipmap with RGB texel values. Our two algorithms differ in both visual quality and speed. One algorithm tends to be faster very high up in the pyramid, while the other is better at resolutions that are more zoomed in low in the pyramid.

4.3.1 Coverage Map Tile Creation

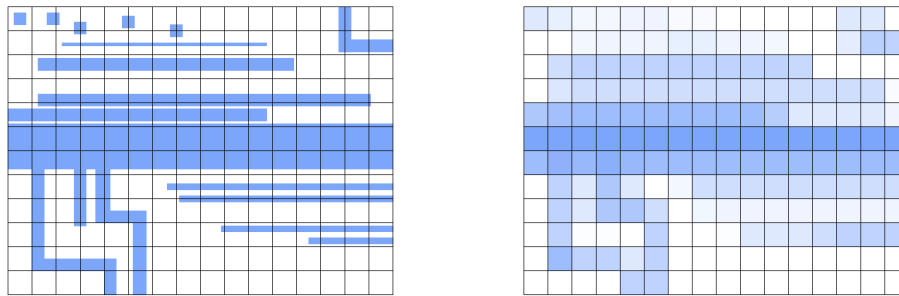
The first tile creation algorithm we will discuss is called *coverage map tile creation*. Fundamental to this algorithm is a structure called a *coverage map* shown in Figure 4.4.

A coverage map is the coverage information for a single layer of layout data. While a texel in a texture tile contains full RGB color information and is usually three bytes wides, a texel in a coverage map holds only aggregated coverage information for one layer and is usually one byte per texel.

Figure 4.4(a) shows rectangles from a single layer of a VLSI layout with a texel grid superimposed on top of it. Then Figure 4.4(b) shows the corresponding coverage map encoded with gray scale values. Since coverage maps always contain information from just a single layer, we invoke the assumption of non-overlap presented in Equation 4.2.

Tile creation using coverage maps is simple and occurs in two steps. First, a set of coverage maps, with the same dimensions as the texture tile, are used to rasterize each layer separately. Then the individual coverage maps are blended from front to back to form the final texels.

The memory overhead of the coverage map structure is reasonable. For a texture tile that is 256×256 on a side, each coverage map is 64KB given 8-bit precision. An average design might have a few dozen visible layers, which would be a memory cost of a few megabytes. An extraordinary design could have hundreds of visible layers, but still the total cost would be tens of megabytes, small in comparison to the size of the design itself.



(a) A single layer with sub-texel resolution.

(b) The corresponding coverage map.

Figure 4.4: On the left is a single layer of layout information with sub-texel resolution. On the right is the corresponding coverage map encoded with gray scale values. Darker values indicate more coverage.

If an implementation uses a multi-threaded approach, then one coverage map structure is needed for each worker thread.

4.3.2 Direct Tile Creation

The other tile creation algorithm is called *direct tile creation*. In direct tile creation, the rectangles are directly rasterized into the final texture tile without a coverage map. The layout rectangles are visited in a front-to-back manner and composited directly into the texture using Equation 4.3.

Equation 4.3 requires a value of α'_{src}. Which value should we use? If the incoming coverage value is from the same layer as the existing coverage value, then Equation 4.2 would be appropriate. However, if we are compositing different layers, then arbitrary overlap would be more correct and Equation 4.1 would be a better choice. To allow a distinction to be made, we introduce a fifth texel element to be used in direct tile creation. We call it a *layer identifier*, or *lid*, and it is simply a small integer that reflects the very first type of rectangle to be composited into a texel.

The first time a texel is written, the *lid* is set to equal the layer identifier of whatever layer type is writing to it. Then, whenever another coverage value is to be blended in via Equation 4.3, α'_{src} is computed via Equation 4.2 if the *lid*s match and via Equation 4.1 if they do not.

The use of the lid is a heuristic and does not guarantee that the correct assumption will always be made. If one type of rectangle first establishes the lid value, and then two coverage values from the same rectangle type as each other but different from the first rectangle type are composited later, each will be blended using the arbitrary overlap assumption since neither of their lids will match the texel lid. In practice though, this is not a significant problem because direct tile creation is not used in situations where an incorrect assumption would be noticeable.

4.3.3 Discussion

The two tile creation algorithms presented in the previous sections each have strengths in different situations.

For texture tiles high up in the pyramid, coverage map tile creation is preferred. These tiles contain the most rectangles, so rasterization time is the biggest performance bottleneck. To minimize this bottleneck, coverage map tile creation is generally faster for two important reasons.

First, since each coverage map is made up of a single value, Equation 4.3 need only operate on a single alpha channel, making the rasterization as fast as possible. The other three values of the texel do not come into play until the final blending stage.

Second, the coverage map tile creation requires only one pass through the layout database, since rasterization can take place in an arbitrary order. For certain designs and/or layout database implementations this could make a huge difference in total computation time. Imagine a layout database that simply kept all rectangles in an unordered linked list. Coverage map tile creation would require only a single pass over this list while direct tile creation would require N passes given N types - taking N times as long. While not as pathologically bad as an unordered linked list, Magic's corner-stitched data structure also incurs a performance penalty due to the need to traverse multiple times.

On the other hand, for texture tiles low in the pyramid, the direct tile creation algorithm is preferred. Viewpoints low in the pyramid by definition consist of fewer, larger rectangles. With coverage map compositing, many of the individual coverage maps would be sparsely populated, and the performance cost of blending these sparse maps could easily

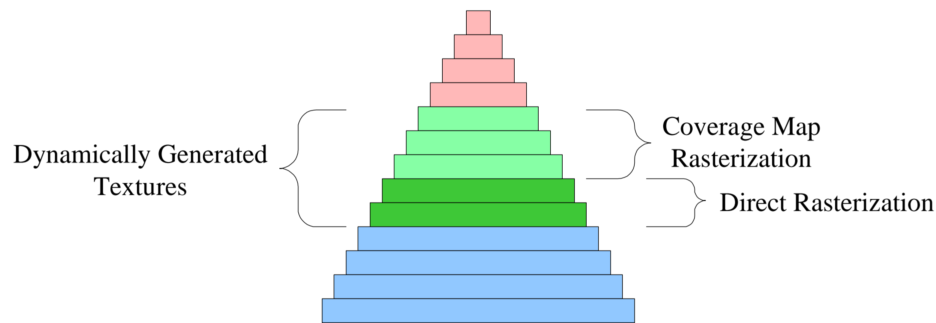


Figure 4.5: A cross-section of a chipmap showing the coverage map vs. direct rasterization boundary.

outweigh the cost of direct tile creation. Since the total number of rectangles is small, the direct tile creation cost of visiting them in a sorted front-to back ordering is not a bottleneck. Finally, since texels will likely be covered by a small number of rectangles low in the pyramid, errors, like the ones due to using *lid*s discussed in the previous section, by definition do not occur as often.

Another way to choose between the different tile creation strategies is to consider how many rectangles will be represented by each texel. If each texel will overlap many rectangles, which is likely for tiles high up in the pyramid, coverage map tile creation is best to minimize the rasterization computation. On the other hand, if each texel will overlap just one or two rectangles, which occurs at high zoom, it is better to use direct tile creation which does not have the overhead of a final blending stage.

Coverage Map Tile Creation vs. Direct Tile Creation Boundary

Given that we have two ways to create texture tiles, and it is probably faster to use coverage map tile creation high in the pyramid and direct tile creation low in the pyramid, where should the boundary be drawn between the two? Unfortunately, there is no hard and fast rule. The speed of one technique versus the other is highly design- and database-dependent so any boundary must be based on a heuristic.

Figure 3.1 is redrawn in Figure 4.5 to show this new dividing line. The boundary further divides the dynamically generated texture data region discussed in Section 3.3.

In our implementation, the boundary level is two levels above the primary geometry/texture boundary. That is, the first two pyramid levels in the dynamically generated region use direct tile creation, while all the levels above it use coverage map tile creation. While empirical evidence generally supports this number (for our three example designs this boundary represents the cross-over point between the two algorithms), we can also derive this number based on our previous design decisions.

Looking back to Section 3.2.1, see that we have chosen five pixels to be the average shorter dimension of rectangles at the boundary between geometry and texture. This means that the average dimension two levels above this boundary would be about one pixel. Since direct tile creation is more effective when each texel has only one or two visible rectangles overlapping it, it makes sense that we place our boundary two levels above the geometry dividing line. For levels higher than this, the average texel will cover more than one rectangle, making coverage map tile creation more attractive.

4.4 Pre-rasterized Hierarchy

By far, the bottleneck in the process thus far is the rasterization step required for each rectangle. Now, we will now describe an optimization that can speed up the tile creation process by as much as a factor of four (based on experimental results), by pre-computing rasterization results for selected parts of the design. To do this, we take advantage of the explicitly instantiated hierarchy (also known as *sub-designs* or *cells*) that frequently exists in VLSI designs. The level of instantiation can be nested arbitrarily, providing a way to describe an enormous amount of redundant complexity. One of our example designs, the Databuffer, has 100× more total rectangles than unique rectangles, which is not uncommon for these types of designs.

We *pre-rasterize* selected sub-designs such that when the sub-design overlaps a particular texture tile, its pre-rasterized hierarchy data is simply copied and the on-the-fly rasterization step is eliminated. This *hierarchy cache* is a block of preallocated memory used to store the hierarchy data. For a VLSI layout viewer, the cache is created and populated at startup and remains static throughout the life of the program. For an editor, the

cache can be updated dynamically with different designs or modified as changes to existing designs are made.

The rest of this section will explain the entire process of using hierarchy data in the texture tile creation process. This includes how the data is stored and used in tile creation, how sub-designs are selected to be included in this cache, how the size of the cache is set, and an explanation of the error artifacts that are generated as a consequence of using hierarchy data.

4.4.1 Hierarchy Data Format

The hierarchy data is a set of coverage maps for each level of resolution in the main design's chipmap pyramid. The coverage maps themselves are very similar to the coverage map structure described in Section 4.3.1. Unlike that structure, however, the maps' dimensions are whatever the scaled size of the cell happens to be. This data does not have to be tiled or a power of two in size because it is never used directly for display; it is only copied into the final texture tile. We compute a set of coverage maps for each level in the pyramid where textures are created dynamically so that the data is immediately available when needed.

A particular sub-design can itself contain other child sub-designs. When the hierarchy data for a cell is created, it is advantageous to "flatten" the design data from all the child cells directly into the parent's coverage maps. For deeply nested designs, this yields large performance benefits because time is saved traversing down the hierarchy. In addition, since the memory requirement of a cell is determined by its overall bounding box, there is no memory penalty to flattening the data. If an implementation is designed as an editor, then any modification of a child design will cause all parent designs and their descendants to be recomputed.

4.4.2 Using Hierarchy Data In Tile Creation

The process of using hierarchy data to create a texture tile via the strategies discussed in Section 4.3 is straightforward. Both strategies visit each rectangle contained in a tile to rasterize it. Along the way, it is necessary to descend through the design hierarchy to access the data. We now add an extra step. For each sub-design that intersects the tile and

has been included in the hierarchy cache, we can simply use the hierarchy data directly, copying it into the appropriate structure. In particular, for coverage map tile creation, each of the appropriate coverage maps from the hierarchy data is added to the main coverage map structure using Equation 4.2. The final blending step is unchanged. For direct tile creation, the hierarchy data is added directly into the final texture using Equations 4.1 and 4.3. In both cases, we have eliminated the total on-the-fly rasterization time that would have been needed for the cell and, because the hierarchy data is flat, we also have avoided descending further into the design to process more rectangles.

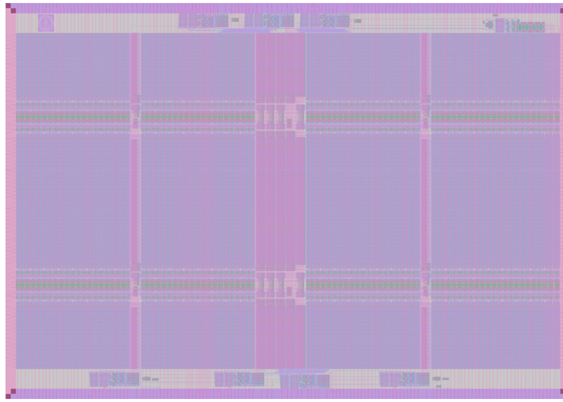
It is now clear why the hierarchy data is stored in coverage map form. Correct compositing requires that fragments be added to texels in a given layer stacking order. If the hierarchy data were pre-composited, it would be impossible to accomplish this. Even if the stacking order were fixed, rectangles from other cells or the parent design could overlap a sub-design's bounding box, causing incorrect results.

4.4.3 Sub-design Selection Algorithm

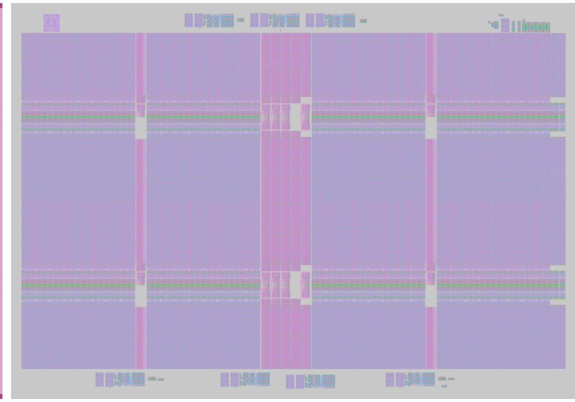
How does one select which cells out of the whole design to pre-rasterize? First, we would like to pre-rasterize the sub-designs that would show the most benefit. For every sub-design, it takes some amount of time, X, to copy in its pre-rasterized data, and a different amount of time, Y, to rasterize the data on-the-fly. The *speed-up* is the ratio Y/X and there are two variables that determine the speed-up for a sub-design: the amount of geometry contained in the sub-design and the current level of the chipmap pyramid for which tiles are being created.

To understand the geometry dependence variable, consider two sub-designs that have exactly the same dimensions but differ in the number and size of the rectangles they contain. The amount of time to copy each sub-design's pre-rasterized data into a tile would be the same, but the amount of time necessary to rasterize the geometry on-the-fly could be very different for each sub-design, yielding very different speed-up values.

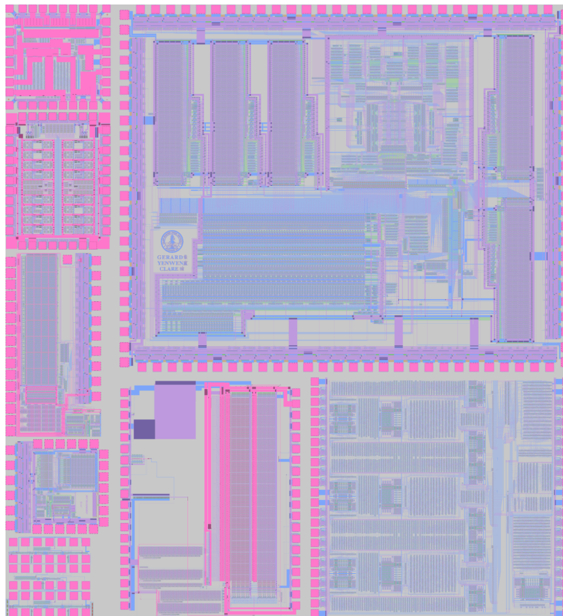
To explain the speed-up dependence on the current level of the pyramid, think about a sub-design with N rectangles and an area of M texels at some chipmap pyramid level. The time to rasterize data on-the-fly is strongly related to the number of rectangles N while the



(a) Databuffer



(b) Databuffer



(c) SU_Block

(d) SU_Block

Figure 4.6: The images on the left show the Databuffer and SU_Block designs in their entirety. Compare them with the corresponding images on the right which renders them only showing the contents of their 32 MB hierarchy caches. The Databuffer design, being much more hierarchical, is able to store 87% of its layout area in the cache while SU_Block can only store 28%.

time to copy the pre-rasterized data is dependent on the number of texels to copy, M. Now move up one level of the pyramid and the number of rectangles to rasterize on the fly is still N while the number of texels to copy for pre-rasterization would now be $M/4$. As we continue higher, each level reduces the computation to copy by a factor of four while the amount of time necessary to rasterize on-the-fly is still strongly related to N, the number of rectangles. Thus, as we move up the chipmap pyramid, the effectiveness of using hierarchy data increases.

To account for each of these variables, each sub-design's speed-up at each level of the pyramid could be measured and ranked. However, we have found this is not necessary because we can generalize both effects. First, we ignore the fact that there is a geometry dependence by making the hierarchy cache large enough to hold most of the designs that have the largest speed-up benefit. (See Section 4.4.4 for more details.) Second, we account for the pyramid level dependence by assuming that because most rectangles are similarly sized, we can determine a cut-off level below which hierarchy data will not be used. For the same reasons discussed in Section 4.3.3, hierarchy data coverage maps are only used when the coverage map tile creation algorithm is active.

Given that we have decided to ignore the variability in sub-design speed-up, the factor we use to preferentially rank sub-designs is the number of instantiations of each design. This maximizes the number of texels on the display that are represented in the cache. We now have the problem of trying to maximize the total number of instantiations present in the cache given that it is a fixed size. This is a bin-packing problem which has no optimal solution that can be computed efficiently so we must rely on heuristics. We have experimentally determined that ranking sub-designs strictly by instance count is basically as good as other more complex heuristics in packing the hierarchy cache, so we simply place sub-designs by instance count greedily into the cache until no more sub-designs fit into the cache.

4.4.4 Selecting Hierarchy Cache Size

In the previous section, we concluded that placing sub-designs with the highest instance count in the hierarchy cache would generally yield the most speed-up, but we have not

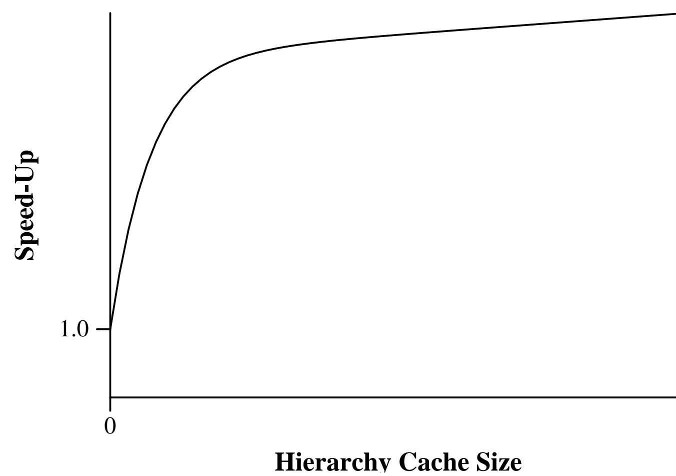


Figure 4.7: For hierarchical VLSI designs, the speed-up possible as hierarchy size increases grows quickly and then tails off quickly. See Section 6.3.2 on Page 86 to see how speed-up varies with hierarchy cache size for our three example designs.

answered the question of exactly how large to make the hierarchy cache. Experimentally, we have found that the hierarchy cache size versus speed-up graph generally follows the pattern shown in Figure 4.7.

Figure 4.7 shows that a small amount of memory generally accounts for the majority of the speed-up. This is because it is always the case that the sub-designs that are instanced the most, thus yielding the largest speed-up, are also the smallest, and thus are the most beneficial to add to the cache from a memory perspective. This property is guaranteed because a parent sub-design must be at least as large as each child and a parent cannot be instanced more times than its children.

Further examining Figure 4.7, it seems most appropriate to set the hierarchy cache size just to the right of the knee in the curve to get the majority of the speed-up with the minimum memory cost. As we will see in Section 6.3.2, the smallest knee in the curve for our three example designs is about 32 MB, which is what we use as a default in our implementation. As designs continue to grow in the future, the knee in this curve will continue to move to the right, mandating that more memory be allocated to the hierarchy cache.

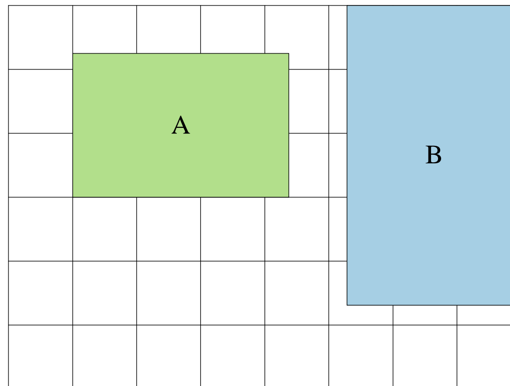


Figure 4.8: A texture tile superimposed with its texel grid and the outline of two sub-designs. Sub-design A is perfectly aligned with the grid, while sub-design B is not which will cause errors when its data is used.

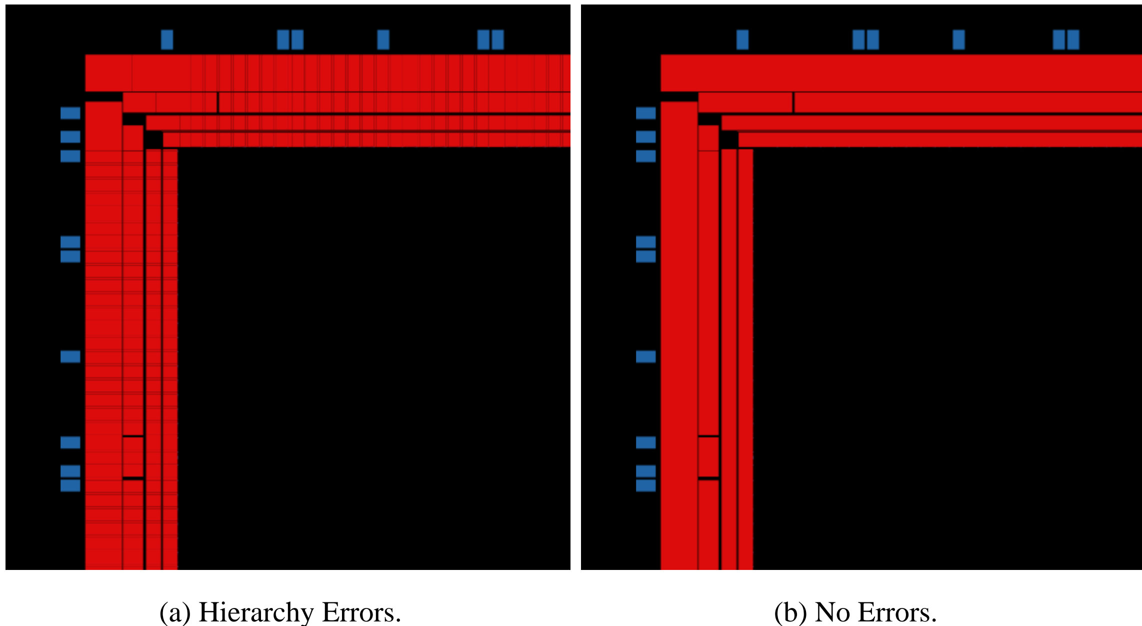
4.4.5 Hierarchy Errors

The use of hierarchy data is not without a drawback. Unfortunately, errors are introduced in the compositing stage that can produce noticeable visual artifacts. The problem is that both hierarchy data and texture tiles are aligned to their own local coordinate systems but they are arbitrarily aligned to each other.

Figure 4.8 shows a texture tile with a texel grid superimposed on it in addition to two sub-designs, A and B. The bounding box of sub-design A is exactly aligned with the texel grid so no errors are incurred because the local coordinate systems of the sub-design and the texture tile are the same. However, the bounding box of sub-design B is mis-aligned with the texel grid. This sub-design must be snapped to the closest texel grid point, which is down and to the left, in order to be copied into the main tile. The error occurs when the hierarchy data is composited into the main tile because the coverage values from the hierarchy data are not exactly aligned with the area they are supposed to represent.

The worst case error occurs when the sub-design falls in the middle point of a texel, yielding a $1/2$ texel error in both directions. Additionally, the probability that *some* error will occur increases with higher levels of the pyramid since each texel covers a larger area of the overall design.

It is not possible to create one set of hierarchy data that always matches the coordinate system of each instantiation because the offsets could be different. While it is possible



(a) Hierarchy Errors.

(b) No Errors.

Figure 4.9: Figure 4.9(a) shows a portion of just the Flash design rendered with a large number of hierarchy errors. Figure 4.9(b) shows the same view without any errors.

that some sub-designs may only be instantiated once, or that all instantiations will have the same offsets, neither of these scenarios is very likely to be true with any frequency.

In practice, this error occurs and does produce noticeable visual artifacts. Figure 4.9 shows just such a situation. Both images show the pad ring of the Flash design. Each individual pad is exactly flush with the metal neighboring it, producing a smooth ring that circles the chip. This is correctly shown in Figure 4.9(b). The hierarchy errors shown in Figure 4.9(a) manifest themselves as thin black lines between each pad. This artifact could mislead the viewer into believing that space does indeed exist between the pads and the rest of the pad ring.

Fixing Hierarchy Errors

Even though it is not possible to create one set of hierarchy data that matches the offsets of every sub-design instance, it is possible to create *every possible* offset at the expense of extra memory.

Each coverage map derives four averaged down coverage maps with different offsets. One is centered at the $(0,0)$ point in the texel as before, but the other three are created so that they are centered at $(0,\frac{1}{2})$, $(\frac{1}{2},0)$ and $(\frac{1}{2},\frac{1}{2})$ respectively. The process continues recursively creating a coverage map structure whose memory size is constant per level. Said another way, one level of area N derives 4 coverage maps, each with area of $N/4$ and a combined area of N. The next higher level consists of 16 coverage maps each with an area of $N/16$, and so on.

Using this structure is as simple as choosing the correct coverage map given the offset of the sub-design in the main design. This is how Figure 4.9(b) was produced. The glaring drawback of this method is that it incurs a $4\times$ memory penalty. Given a fixed size hierarchy cache, this decreases the number of sub-designs it can hold by the same factor of four. Although hierarchy memory does not consume any graphics memory (a very limited resource), an implementation could just choose to increase the size of its cache. Our implementation used a 32 MB hierarchy cache, and even with this modification, the cache was large enough to provide $2\times$ to $4\times$ performance boost for the three designs presented here. See Section 6.3.2 for more information.

4.5 Summary

The purpose of this chapter was to explain how the specific properties of VLSI layouts could be exploited to achieve certain goals. We have demonstrated how to create the texture tiles that populate the chipmap structure described in Chapter 3, allowing for the fast and accurate creation of VLSI layout visualization data with a very minimal memory overhead.

The techniques presented in this chapter were given in the context of creating an accurate image of a layout. In Chapter 5 we will show how the same techniques can be used to create texture tiles that are used to create other useful visualizations in the VLSI design process.

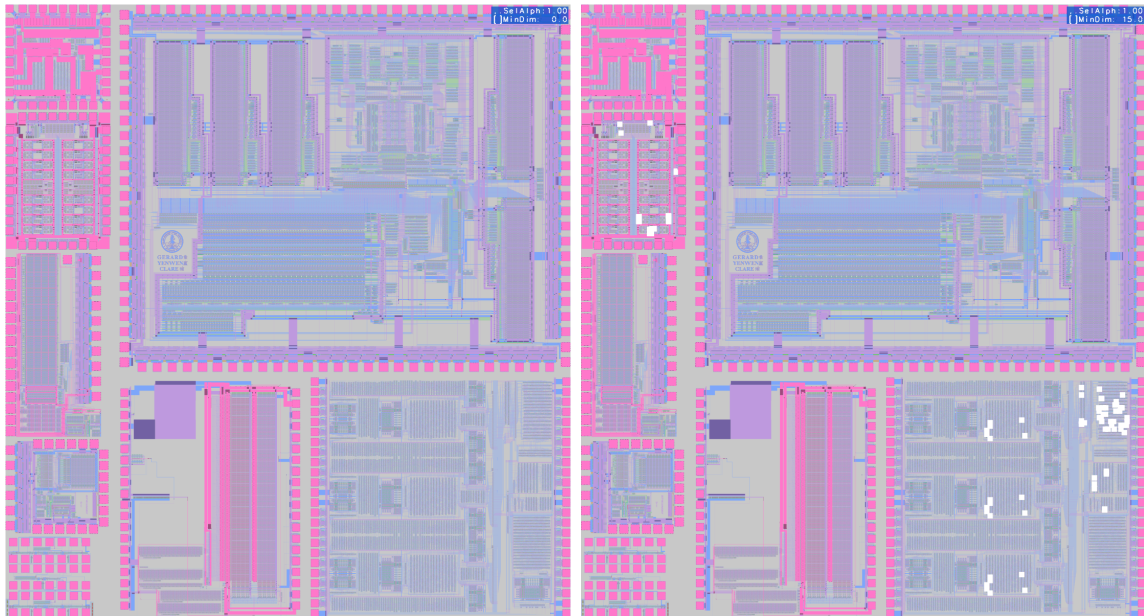
Chapter 5

Other Visualizations

The previous chapter detailed how photo-realistic VLSI layout image data is created. While viewing VLSI layout in a photo-realistic manner is very useful, it is not the only type of visualization that would be helpful to a chip designer. In this chapter, we will detail the other ways that VLSI physical design data could be displayed within the chipmap infrastructure.

The purpose of this chapter is to demonstrate what is possible with a chipmap. All of the visualizations we will present have been created in one form or another on other systems, although those systems have the same speed and accuracy limitations in displaying this type of information as with layout data. We believe that by using a chipmap, these visualizations become even more powerful and intuitive.

First, we will show how we can use exaggeration and layering to enhance the data selection process. Next, we detail how a chipmap can be used to visualize a chip floorplan and cell placement information. Finally, we explain how we can back-annotate analysis information onto a layout to help a designer gain insight into the quality of the analysis results.



(a) SU_Block errors selected but not exaggerated (b) SU_Block errors exaggerated to 15 pixels

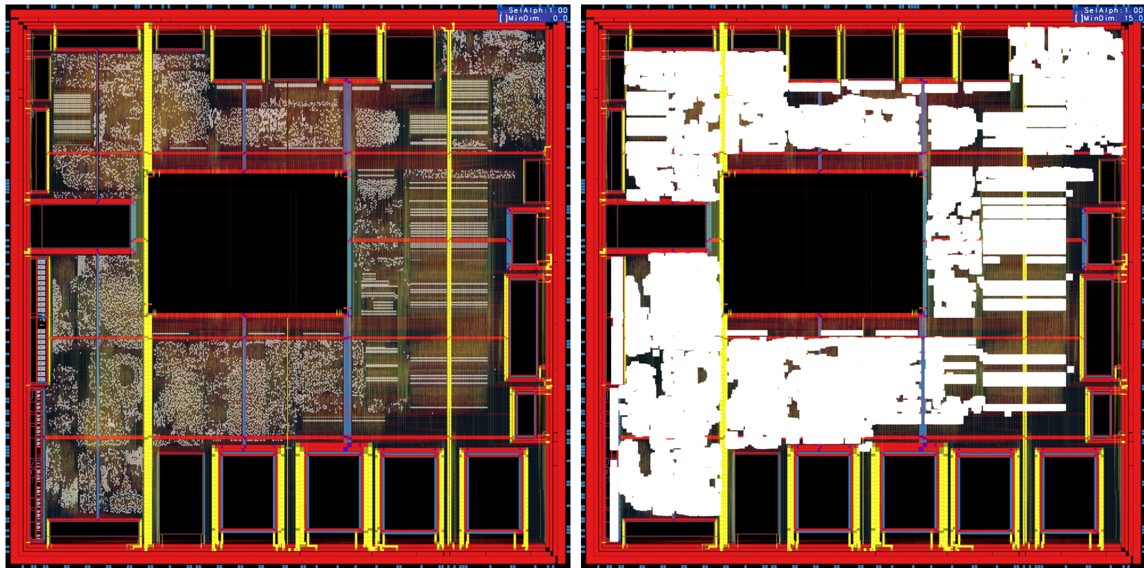
Figure 5.1: The image on the left shows the DRC errors of the SU_Block design selected without exaggeration. One the right, is the same data except that it has been exaggerated to at least 15 pixels on a side.

5.1 Data Selection Visualization

A very common operation in VLSI design is *data selection*. A designer often wants to highlight a particular feature or group of similar features either to physically find it, or to determine if that group of features presents a pattern.

The first example of data selection, shown in Figure 5.1, is the DRC error information in the SU_Block design. In the image in Figure 5.1(a), however, the errors have not been exaggerated, while in Figure 5.1(b) they have been exaggerated to a minimum 15 pixels on a side. As the viewpoint zooms in, the relative amount of exaggeration needed gradually goes to zero so that, at some point, the errors appear as they would in the layout. This level of exaggeration is important to make the sparse nature of the errors stand out.

In other situations, this level of exaggeration would only obscure the selection. In Figure 5.2(a), the 17,539 flip-flops of the Flash design have been selected with no exaggeration, while in Figure 5.2(b), they have been rendered to have a minimum dimension of 15



(a) Flash flip-flops selected but not exaggerated (b) Flash flip-flops exaggerated to 15 pixels

Figure 5.2: The image on the left shows all the flip-flop devices of the Flash design selected without exaggeration. One the right, is the same data except that it has been exaggerated to at least 15 pixels on the side.

pixels on a side. In this case, no exaggeration is desired because of the size and density of the flip-flop cells. When the exaggeration is too great, the selected elements overlap each other, making it difficult to discern the individual flip-flops. This results in hiding their total number and also hiding any patterns that might be present in their placement – the un-exaggerated view more clearly shows the individual hand-placed rows of flip-flop cells.

The visualizations in Figures 5.1 and 5.2 were both created with *overlay textures*, which is an additional layer of texture data that is blended into whatever image is already on the display. On systems supporting texture mapping, blending an additional layer of data can usually be done with little performance penalty. In this case, the overlay data is a coverage map-like structure that has the capability to display a single color modulated from fully opaque color to completely transparent. In other words, the selection texture data is another set of different texture tiles that is also computed at the same time as the regular image texture tiles are created. The advantage to keeping the selection data separate from the primary image is that changing the overlay texture data requires no recomputation of the primary image.

To produce visualizations similar to those in existing systems, a *hardware overlay plane* must be employed to achieve the same effect. A hardware overlay plane is a separate area of framebuffer memory that is available with some higher-end graphics systems. There are two main drawbacks to using a hardware overlay plane versus an overlay texture. First, The data in the overlay plane can either fully occlude the data underneath it, or it can be totally transparent. It is not possible to blend the values of the overlay plane with the main framebuffer as it is with an overlay texture. Second, the number of layers that can be added is limited to the number of physical overlay planes present on the system. This number can be as high as four, but is usually just one. On the other hand, the number of overlay textures that can be blended is only limited by system memory constraints.

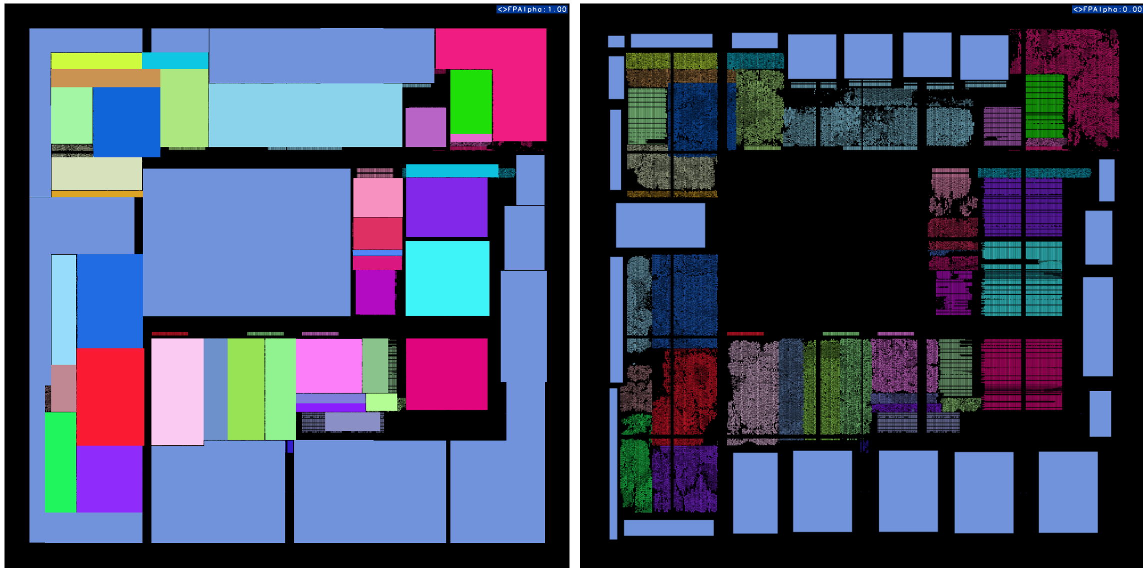
The important point about using overlay textures with with a chipmap is it presents a formalized way of adding an arbitrary number of additional data layers without disturbing the original image.

5.2 Floorplan Data Visualization

Floorplanning a VLSI design involves partitioning the die at a global level with the locations of the functional blocks, hard macros, and embedded memories. After a floorplan is in place, it is usually used by another tool as a starting point for cell placement. That is, the cells contained within a particular floorplan block will nominally be placed within that block. The chipmap infrastructure can visualize this process.

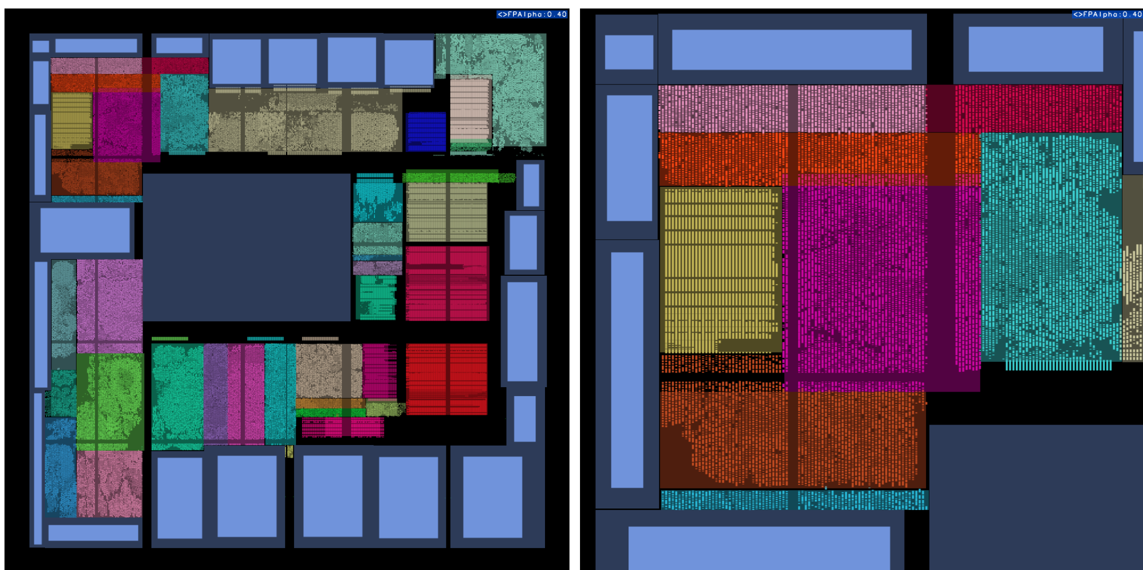
Figure 5.3(a) shows the Flash design's floorplan blocks. The embedded memories are all shown in the same shade of lighter blue while every other functional block has a color randomly assigned. Figure 5.3(b) shows the approximately 87,000 placeable cells of the Flash design colored to match the blocks they are a part of. The cells were rendered into texture tiles using the same anti-aliasing techniques that were used to render the layout rectangles.

Figure 5.4 demonstrates how transparency can be used to further give insight into the floorplanning process. Here we show two views of the floorplan where the blocks have been transparently blended on top of the cells. By using transparency, this visualization gives better insight into how the cells were placed. In Figure 5.4(b) it is apparent that few



(a) Flash floorplan blocks (b) Flash placeable cells

Figure 5.3: Screenshots of the Flash floorplan. On the left are the global blocks. On the right are the placeable cells colored to match the block from which they originate.



(a) Flash floorplan blocks and cells (b) Flash floorplan zoomed in

Figure 5.4: Screenshots showing the Flash floorplan blocks rendered with 40% transparency on top of the cells. The image on the right is a zoomed in portion of the upper left corner of the design.

cells were placed outside of the blocks to which they were assigned. The fact that the placer rigidly respected the hard boundaries of the blocks may or may not have been the desired behavior and using transparency allows a designer to see that immediately. Contrast this to Figure 5.3, where the blocks were either shown or hidden, making it more difficult to see the placer's behavior.

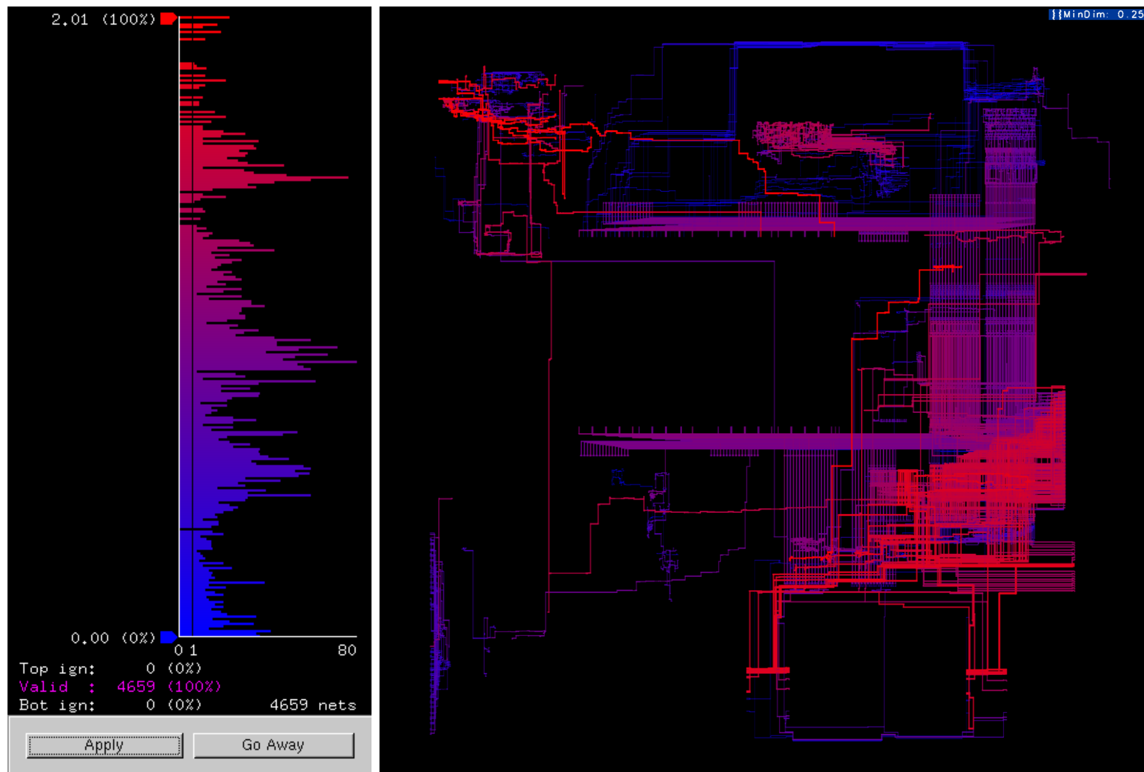
The floorplan visualization is important for two reasons. First, it shows that the chipmap infrastructure can be used to display information other than layouts. Fundamentally, a floorplan and a layout are both made up of rectangles, which allows them to be created in the same way. Second, displaying the floorplan blocks and cells simultaneously using transparency is a powerful way to view this information. While stipple patterns or outlines could be used to achieve this effect, true adjustable transparency allows the designer a flexibility that other techniques do not.

5.3 Back-annotated Data Visualization

We now consider two examples of back-annotating the results of an analysis tool into a chipmap. This demonstrates how we can use color, thickness, and ordering to make the most interesting parts of the analysis stand out. Both analysis examples are static timing related. The first highlights the worst violated paths, and the second shows the clock nets with the greatest skew.

5.3.1 Static Timing Analysis Visualization

The first type of specific visualization we will consider is mapping the results of a timing analysis run on to the VLSI layout. Typically, timing analysis log files are as large as the layout databases themselves and difficult to comprehend in total. Designers usually will examine the top offending paths in an attempt to notice a pattern or they will try to discover an area of the design that is particularly troublesome by matching the text names of the nets. Using the text names can be misleading because two nets that are textually very similar may not be similar at all in the way that they are routed.



(a) Histogram of violated setup time paths

(b) Flash

Figure 5.5: The image on the right is the Flash design colored to only show the nets that are part of paths that have setup time violations. The number and severity of the violation is represented by the histogram on the left.

Back-annotating the timing analysis log file information onto the layout minimizes these problems. Now, instead of coloring the rectangles based on their logical layer association, they are colored based upon their back-annotated value.

In Figure 5.5(b), we rasterize and color only those rectangles in the Flash design that make up nets that are part of violated timing paths. In this visualization, each net is colored based on the worst timing path of which it is a part. In Figure 5.5(a), a histogram of all the violated paths is shown. The Flash design had a target clock frequency of 100MHz. The histogram shows the timing of all 4659 (out of 85201) nets that have violated the 10ns period. Three visualization techniques are used to encode the information to make it intuitive to understand.

First, the paths with the worst timing have been colored pure red while the least violated have been rendered in pure blue. The color of intermediate violators has been linearly interpolated between red and blue based on the amount of the violation. The blue/red color scheme gives immediate meaning to the paths and allows a designers to quickly digest meaning and discern patterns. Visualization theorists call this type of color scheme *sequential* [Brewer, 1994] since the data has an ordering. Contrast this to the *qualitative* coloring scheme of a regular layout where no importance is implied by the colors chosen for the individual layers.

Second, the minimum thickness of the rectangles has been increased to a minimum overall value that is additionally scaled thicker based on the amount of violation. The largest violators are rendered with thicker rectangles, causing them to stand out. In this visualization, the minimum rectangle dimension is 0.25 pixels for paths that are barely violated while the max violators are rendered as 2.0 pixels wide.

Lastly, the most violated paths are rasterized to appear on top of other less violated paths. This is an additional measure to ensure that the biggest violators stand out the most.

The combination of the histogram and the visualization shown in Figure 5.5 gives a powerful view of the setup-timing violations of the Flash design. The histogram shows that there is no long tail of outliers, as a designer would hope to see, because it would indicate that fixing a few paths would speed up the entire design. (This is because this example shows the final timing of the Flash design after all possible timing optimizations had been performed.) The visualization itself shows dramatically where the timing hot-spots are located over the surface of design. In the lower right corner, we can see a large number of bright red wires which happens to indicate a timing problem between an internal register file and a computation core.

Without the color or thickness encodings, the visualization loses meaning and becomes more difficult to interpret. Consider Figure 5.6.

In Figure 5.6(a), the nets are rasterized at their natural thickness without any minimum value, making them difficult to discern. In Figure 5.6(b), the nets have minimum thickness, but they have all been colored identically making it impossible to tell where the biggest violators are located.

(a) No minimum thickness encoding (b) No color encoding

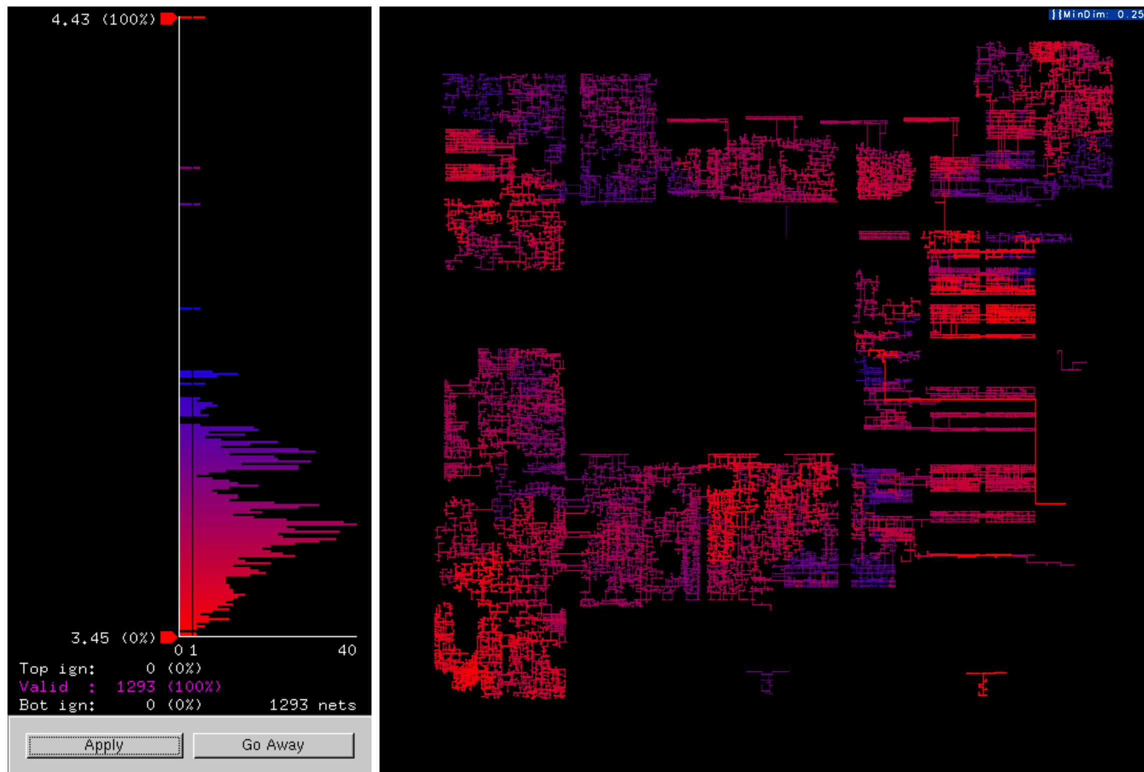
Figure 5.6: The image on the left removes the minimum thickness encoding from Figure 5.5(b), while the image on the right shows the visualization without color. Both images are difficult to interpret without the data encoding.

5.3.2 Clock Skew Visualization

Figure 5.7 colors the paths of the clock tree in the Flash design to show their delay from the main clock driver to the net that drives the clocked elements. In contrast to the *sequential* color scheme we used in the timing analysis visualization, here we use a *diverging* color scheme to highlight the end values the worst, with the central value most desired.

Figure 5.7(a) shows that while the majority of the 1293 clock nets are bunched around a median value, some outliers are causing the majority of nets to be skewed towards the lower bound (red). This makes Figure 5.7(b) useless since almost all the nets are various shades of red. The histogram is interactive, however, allowing us to filter out the highest nets in the histogram so that it is possible to separate the bunch at the bottom. The result of that filtering is shown in Figure 5.8.

Once we have determined that the 9 nets filtered out are unimportant or erroneous for Flash (and in fact they are), we can look to Figure 5.8(b) and see that the worst skew



(a) Histogram of clock skew delay

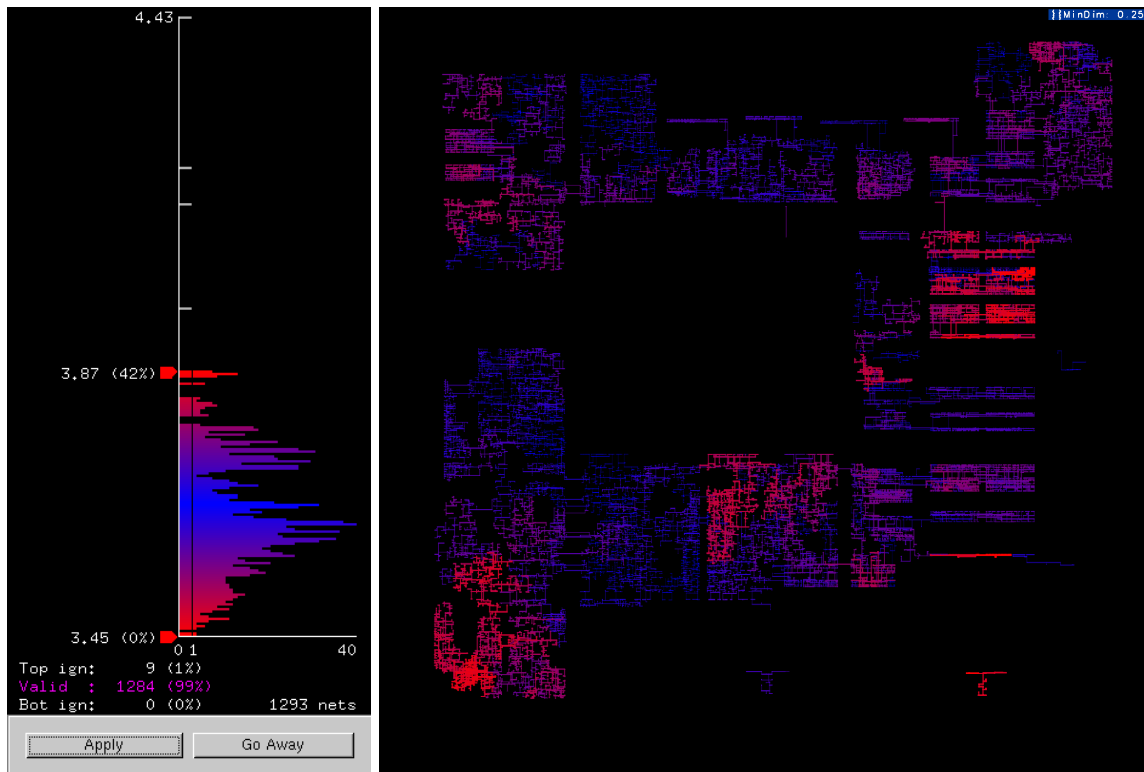
(b) Flash

Figure 5.7: The clock skew delays of the Flash design are represented. Because a small number of outliers make the majority of the paths located closer to the histogram's edge, the meaning of the visualization is obscuring because most of the paths are colored mostly red.

in the Flash design is in the lower left and upper right corners and the global skew is approximately 420ps.

5.4 Data Density Visualization

Our last example is a visualization of wire density in the Flash design. This visualization is different from the others in that the layout information is aggregated in a very non-photo-realistic manner. Here, we will show how the density of particular metal layers varies across the die.



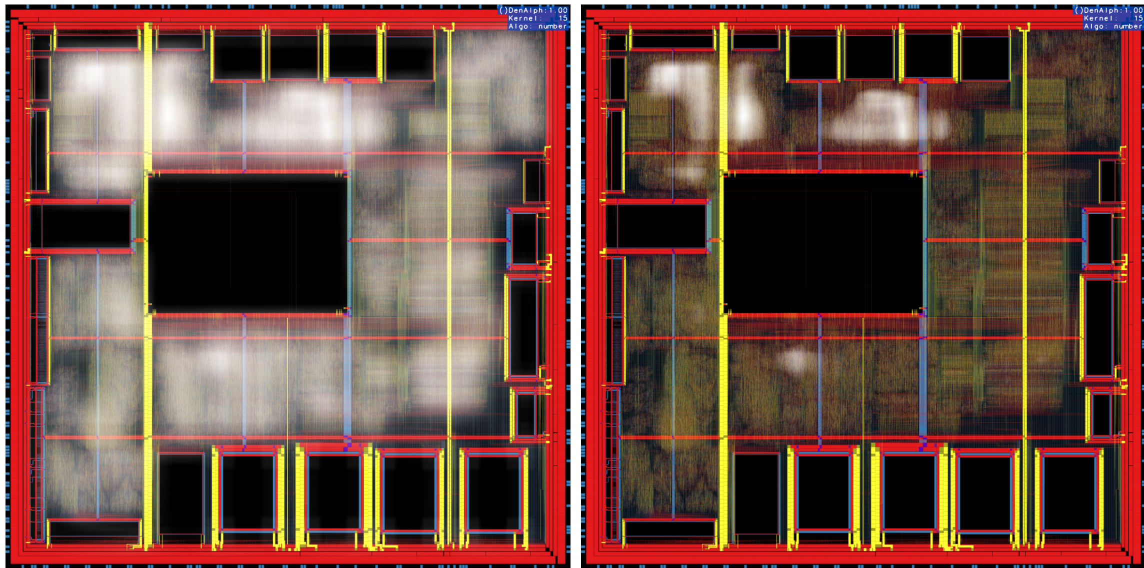
(a) Histogram of clock skew delay

(b) Flash

Figure 5.8: The clock skew delays of the Flash design are represented. By filtering outlier datapoints, a better picture of global clock skew emerges.

Typically, placement and wire routing is run multiple times depending on the quality of results. In order to determine how successful a particular placement or routing run has been, it is useful to see any potential hot-spots of cell or wire congestion.

Figure 5.9 shows the Flash design overlaid with a density map of metal layers 2 and 3. To construct the visualization, a grid is created over the design that corresponds to the routing channel dimension of the design. For Flash, this corresponds to a 1024×1024 grid. Then, the number of metal 2 and 3 wires that cross each grid is counted. Finally, to capture the averaged effect of wire density over an area, a 2-D convolution kernel is run over the grid of data. The width of kernel can be varied to affect the amount of averaging that takes place. The result of using a kernel width of 15 is shown in Figure 5.9(a). The areas of



(a) Flash metal layers 2 & 3 density

(b) Flash metal layers 2 & 3 filtered density

Figure 5.9: The image on the left shows the complete density of metal layers 2 & 3 in the Flash design. The image on the right shows the same data but filtered to highlight the areas of highest density.

brightest white reflect the highest wire density and the areas most transparent are the least dense. To further highlight the areas of high density, Figure 5.9(b) shows the bottom $2/3$ of the data filtered out. Now it is very easy to identify the areas of the design with the highest wire density.

The data density visualization is unique in this chapter because it is the only visualization where the data displayed does not directly correspond to layout features. Also, it is the only visualization where the data does not increase in resolution as the viewpoint zooms in since the number of elements in the grid is constant.

We show the data density visualization as an example of using the chipmap infrastructure to make more apparent information that is secondarily related to the actual layout.

Chapter 6

Results

This chapter evaluates the techniques described in Chapters 3 and 4 along three dimensions – image quality, performance, and memory footprint. We begin by providing a brief overview of how the system was implemented, what hardware was used, and the other layout viewers to which it will be compared. Section 6.2 then examines at image quality in more detail, providing both quantitative and qualitative results. Section 6.3 compares our implementation's rendering speed under various conditions against a state-of-the-art commercial layout editor. Lastly, in Section 6.4, the memory footprint of our implementation is compared against the same commercial layout editor.

6.1 Implementation

A demonstration program called "ChipMap" was created. The Magic Layout System was used as the source code base to read and store the layout data; its user-interface was then modified using a combination of C and C++ and the OpenGL API for graphics rendering.

ChipMap was run on two test systems. One system was a Dimension 8200 PC from Dell Computers [Dell, 2001] running the Redhat Linux 7.2 [Redhat, 2002] operating system. It had a 1.8 GHz Pentium 4 [Intel, 2000b] processor with 1 GB of PC800 RDRAM memory [Rambus, 2000] and a Geforce4 Ti 4600 [Nvidia, 2002] graphics adapter. The second system was a SunBlade 2000 workstation from Sun Microsystems [Sun, 2002] running Sun's Solaris 8 operating system. It had two 1050 MHz Ultrasparc III processors

with 8 GB of memory and an XVR-1000 graphics adapter. In one test concerning multiple processors, a four processor Sun Microsystems 400MHz UltraSparc II system with 2 GB of memory was used. This use is noted.

Comparisons were made between ChipMap running in different configurations and also against the following systems:

Cadence Design Systems' Virtuoso [Cadence, 2002b] (v4.4.6) Virtuoso is currently the most popular and widely-used VLSI layout editor and we compare to it to relate real-world performance. Virtuoso was configured to always draw filled rectangles and to have no threshold for discarding small rectangles. This best matches the work that ChipMap performs when it renders a layout. Virtuoso uses the X11 graphics library to render layouts.

Magic Magic and ChipMap share a common code base for storing the layout database. We include Magic in our comparisons to eliminate this factor as a variable. Similarly to Virtuoso, Magic also uses X11 to draw layouts.

OpenGL Finally, we compare ChipMap configured to draw all rectangles individually with OpenGL. OpenGL rectangle rendering is different than X11 rendering in that drawing in an anti-aliased fashion is possible. To achieve this, OpenGL was configured to have `GL_POLYGON_SMOOTH` enabled and to use the blending factors `GL_SRC_ALPHA_SATURATE` and `GL_ONE`. We compare against OpenGL to see how an explicitly hardware optimized path to draw anti-aliased rectangles compares against ChipMap.

All tests were run on both the PC and Sun systems where this was possible. The exceptions were that no multiple processor tests were run on the PC since it has only a single processor. Also, the comparisons with Cadence's Virtuoso were only run on the Sun workstation because Cadence's software is not supported on a PC. The data given in this chapter does not differentiate between runs that came from the PC or the Sun because both machines showed similar relative performance numbers.

All benchmarks were run using the three example designs introduced in Section 2.2 on Page 18.

6.2 Image Quality Comparison

To measure image quality, a rigorously created “perfect” image was generated using an algorithm that is approximately 50 times slower than the image generation techniques described in Chapter 4, but which results in images containing no visible artifacts. To do this, the “perfect” algorithm generates a pixel value by computing an exhaustive visibility test for each rectangle that intersects that pixel. No approximations are done.

6.2.1 Formal Image Quality Comparison

To formally compare the quality between the perfect images and those created by other methods, each pixel error is individually computed and then the Root Mean Square (RMS) error of all pixels errors is computed. Each pixel error, E, is found with equation 6.1:

$$E = \frac{|R - R'| + |G - G'| + |B - B'|}{3} \tag{6.1}$$

where RGB is the color of a pixel in the perfect image and R’G’B’ is the corresponding pixel in the test image. The RMS error is given by equation 6.2:

$$RMS = \frac{\sqrt{\sum E^2}}{N} \tag{6.2}$$

where N is the number of pixels that were compared. Pixels from the two images that match the background color exactly are not used in the comparisons.

We conducted comparisons rendering the three example designs in three ways - ChipMap native rendering, X11 rendering emulation, and OpenGL rendering. Each resulting image was a full-screen view of each design on a monitor with 1600×1200 resolution. X11 rendering emulates the way in which Virtuoso and Magic draw layouts. When these systems draw rectangles using the X11 system, rectangles are guaranteed to be at least one pixel on the display. Table 6.1 gives the RMS error results of the comparisons.

The ChipMap images are the closest match to the perfect images. The OpenGL antialiased images show larger discrepancies primarily because only one assumption of dependent overlap was made when blending fragments together. Section 4.2 explained how

	Databuffer	SU_Block	Flash
ChipMap	2.1%	1.9%	1.7%
OpenGL Anti-aliased	9.3%	6.5%	17.2%
X11 Emulation	12.1%	11.7%	38.0%

Table 6.1: RMS error between the "perfect" test image and images generated via three other methods for the three example designs.

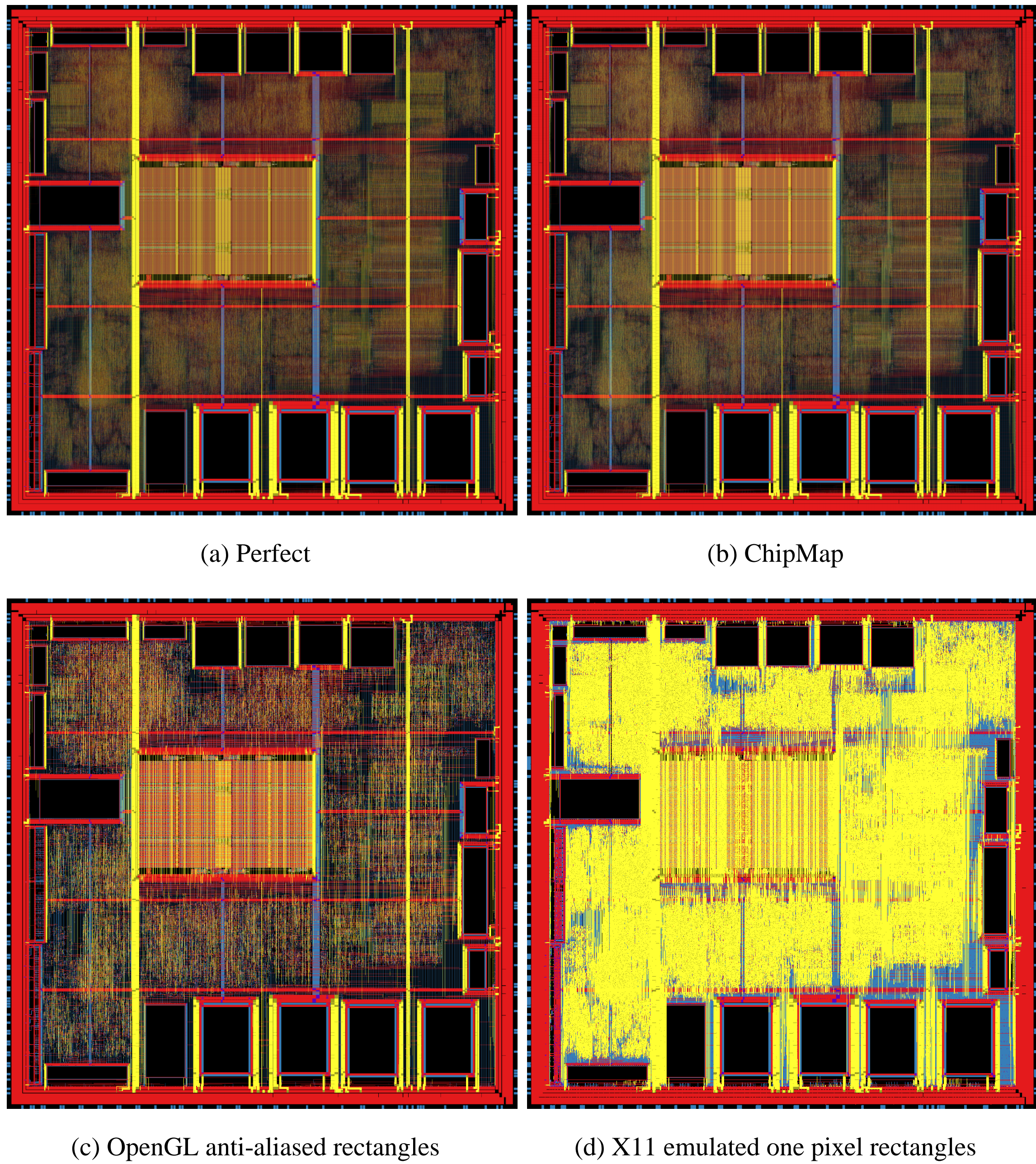
different assumptions were necessary for different and same-layer rectangles for accurate compositing to be achieved. The X11 emulation shows the biggest difference from the perfect images because all rectangles are forced to be at least one pixel on the display, causing massive errors.

Keep in mind that an error of only 33% would result if just one color of the RGB pixel were 100% different while the other two channels matched identically. The only way to measure 100% error overall would be to compare all white pixels to all black pixels and that two randomly generated images would have, on average, an error of 33%.

The images for the designs are shown in Figures 6.1 and 6.2. The Databuffer has been placed in context inside the Flash design to save space. While the differences between the perfect image and the ChipMap image are not easily discernible, the errors between the OpenGL rendering method and the X11 emulated method are severe enough to be seen easily. Figure 6.1(c), showing an image of the Flash and Databuffer designs using OpenGL, may appear to be "sharper" than both the perfect image and the ChipMap image, but the distinct lines seen in the image are the result of systematic rounding errors that have caused an evenly spaced number of wires to appear as lines.

6.2.2 Ad-hoc Image Quality Comparison

Performing actual image comparisons between ChipMap and currently existing tools is difficult on a pixel by pixel basis because lining up comparable images exactly is almost impossible. This is why, in the previous section, the formal comparisons were done between images each generated by our implementation. An ad-hoc comparison can be done however by comparing screenshots. We compare images created with Virtuoso and Magic against the perfect image shown in Figures 6.1(a) and 6.2(a). All of the images in Figure 6.3 show the same visual artifacts as those seen in Figures 6.1(d) and 6.2(d).



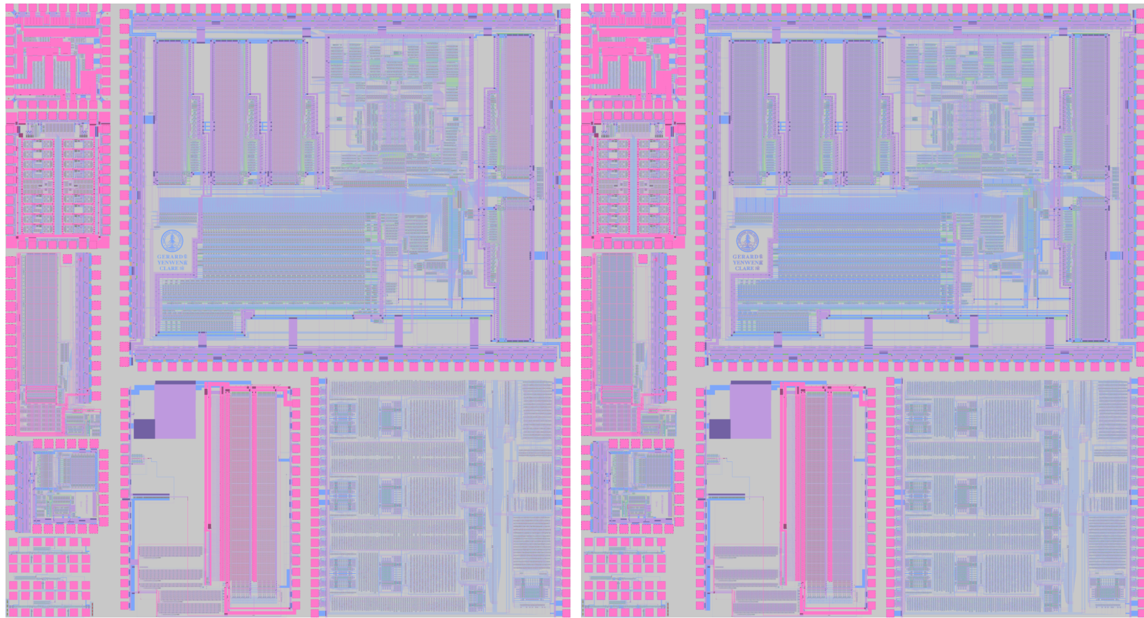
(a) Perfect

(b) ChipMap

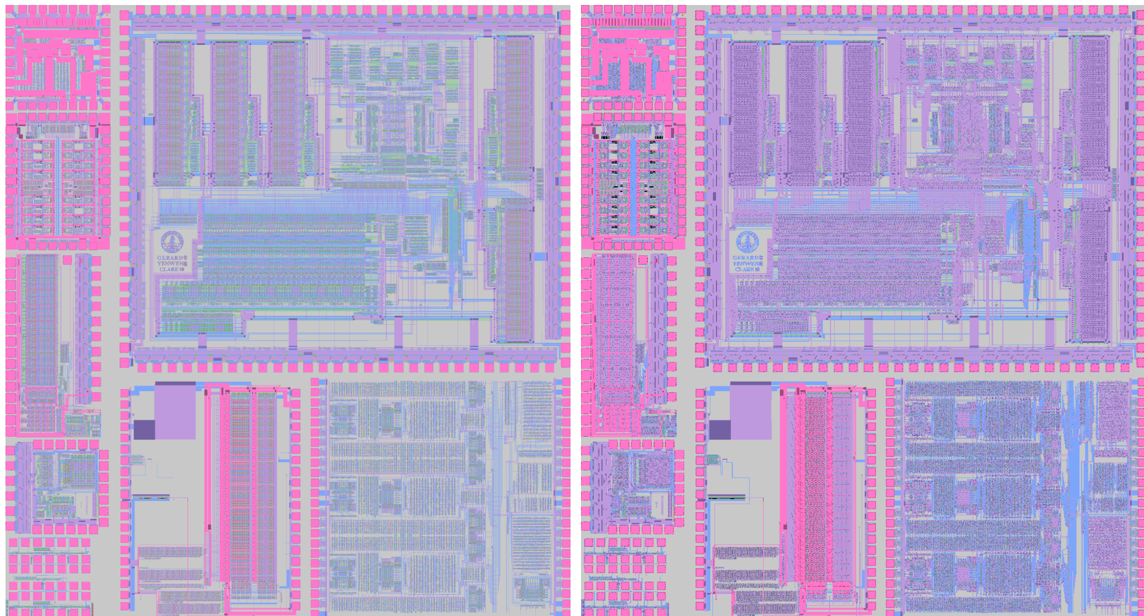
(c) OpenGL anti-aliased rectangles

(d) X11 emulated one pixel rectangles

Figure 6.1: Screenshots of the Flash and Databuffer designs. The Databuffer design is shown instantiated in the Flash design.

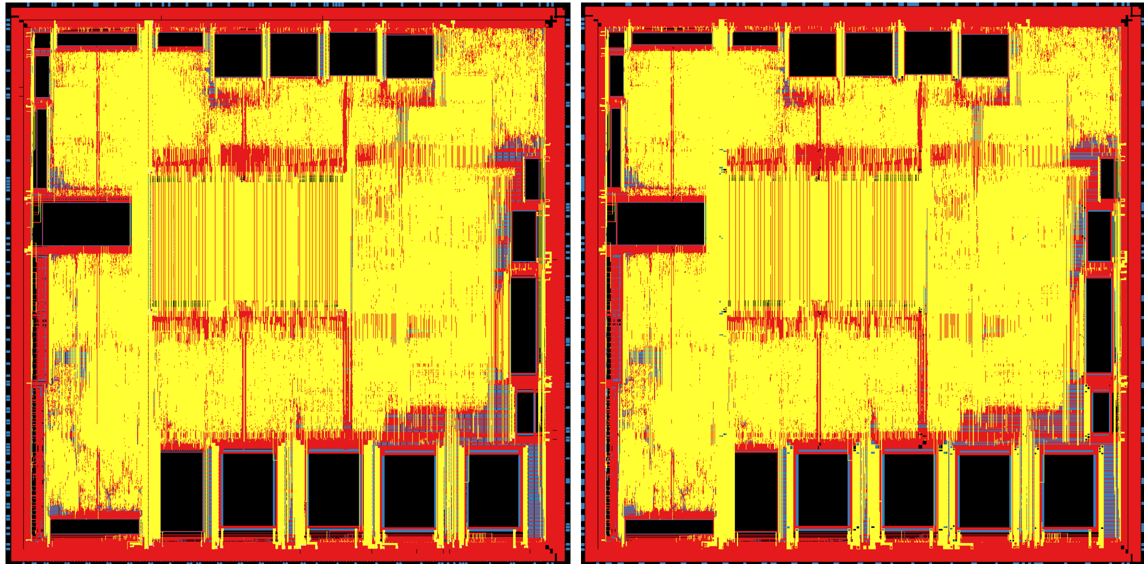


(a) Perfect (b) ChipMap

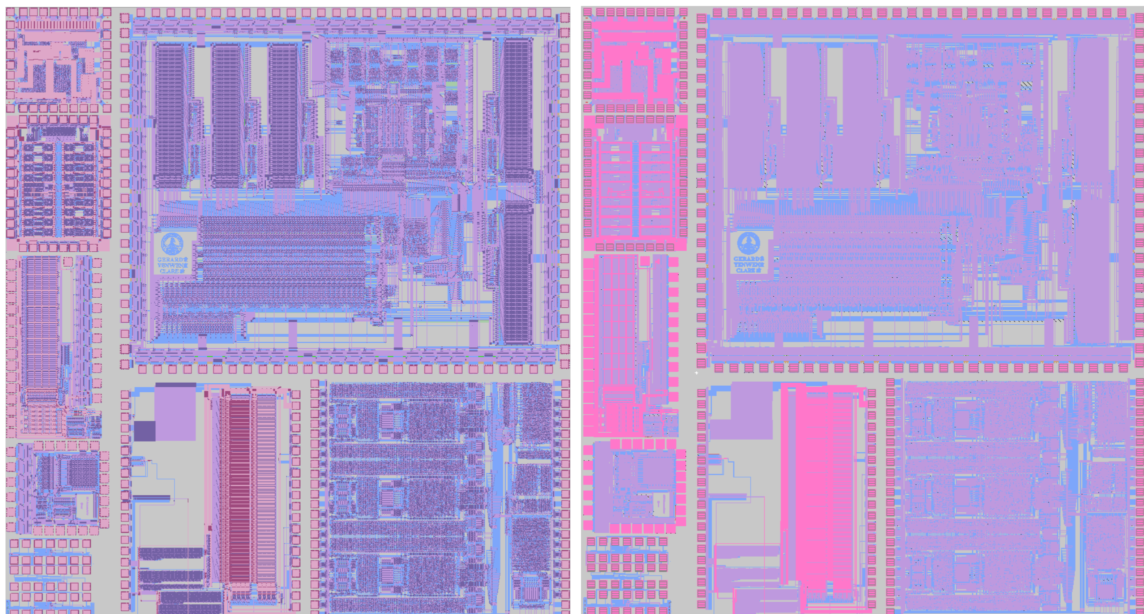


(c) OpenGL anti-aliased rectangles (d) X11 emulated one pixel rectangles

Figure 6.2: Screenshots of the SU_Block design.



(a) Virtuoso rendering Flash (b) Magic rendering Flash



(c) Virtuoso rendering SU_Block (d) Magic rendering SU_Block

Figure 6.3: Screenshots of Flash, SU_Block and Databuffer designs rendered by Virtuoso and Magic.

	Databuffer	SU_Block	Flash
ChipMap	1.0	1.0	1.0
Virtuoso	1.3	1.4	1.2
OpenGL	1.3	1.5	1.6
Magic	1.8	2.1	2.7

Table 6.2: ChipMap, configured with no hierarchy cache and running on a single processor, rendering times versus OpenGL, Virtuoso and Magic. All times have been normalized so that ChipMap's time is 1.0.

6.3 Rendering Speed Comparisons

We now examine how quickly our implementation can create an image under various conditions. We start with the design already loaded into memory and then measure how long it takes to render the entire design once in a window the size of the display at a resolution of 1600×1200.

We begin by comparing ChipMap in an unaccelerated configuration, with no hierarchy cache and running on a single processor. Then we introduce both optimizations to see how they affect performance.

6.3.1 Unaccelerated Rendering Times

Table 6.2 shows the rendering times for the three designs, comparing ChipMap configured with no hierarchy cache on a single processor to OpenGL, Virtuoso and Magic. All times have been normalized so that ChipMap's time is 1.0.

Table 6.2 shows ChipMap to be comparable to, or slightly faster than, to both OpenGL and Virtuoso and for all three systems to be somewhat faster than Magic. The disparity between Magic and the other systems is probably due to the particularly inefficient way that Magic renders a design, making multiple passes through the database for each rectangle[1].

The message from Table 6.2 is that ChipMap's implementation is reasonable since it performs slightly faster than Virtuoso, the standard in commercial layout editors, when drawing the same number of rectangles.

[1] In Magic, each rectangle's appearance is the sum of any number of *styles*. A pass is made through the database for each style.

	Databuffer	SU_Block	Flash
OpenGL - 32 MB VA cache	0.17	0.38	1.4
ChipMap - 32 MB hier. cache	0.26	0.34	0.8
ChipMap - no hier. cache	1.0	1.0	1.0
OpenGL - no VA cache	1.3	1.5	1.6

Table 6.3: Normalized rendering times using a 32 MB hierarchy cache for ChipMap and a 32 MB vertex array cache for OpenGL versus using no caches at all.

6.3.2 Hierarchy Cache Rendering Times

The first acceleration structure we benchmark is enabling a hierarchy cache of 32 MB for both ChipMap and OpenGL. The OpenGL runs were configured with a 32 MB vertex array cache (1 million equivalent rectangles), allowing OpenGL to take advantage of design hierarchy. Table 6.3 shows the results, with ChipMap's time without a hierarchy cache is normalized to 1.0. OpenGL's times with its vertex array cache disabled are duplicated from the previous table for comparison.

As we would expect, the design with the most hierarchy, Databuffer, shows the largest benefit from the hierarchy cache, almost a 4× speed-up with ChipMap. Next, SU_Block, with less hierarchy, shows a speed-up of about 3×, while Flash, the design with the least hierarchy, only shows a speed-up of about 20%.

Note that OpenGL does substantially *better* (6× versus 4×) than ChipMap for the Databuffer design. This is because 32 MB is enough vertex array memory to fit the *entire* design in a vertex array. When this occurs, rendering speed is limited by the rasterization rate of the hardware. For SU_Block, the vertex array data makes OpenGL nearly as fast as ChipMap, but more vertex array memory is needed to become rasterization limited. For the Flash design however, 32 MB is not sufficient to contain a large enough portion of the database to improve its time significantly. It is interesting to note that in the case of the Databuffer, ChipMap, with its 32 MB hierarchy cache is nearly as fast as the fully accelerated, rasterization-limited OpenGL configuration.

Varying Hierarchy Cache Size

Figure 6.4 shows how the speed-up varies given different hierarchy cache sizes for the three designs. A cache size of zero bytes corresponds to 1.0, or no speed-up. Keep in mind

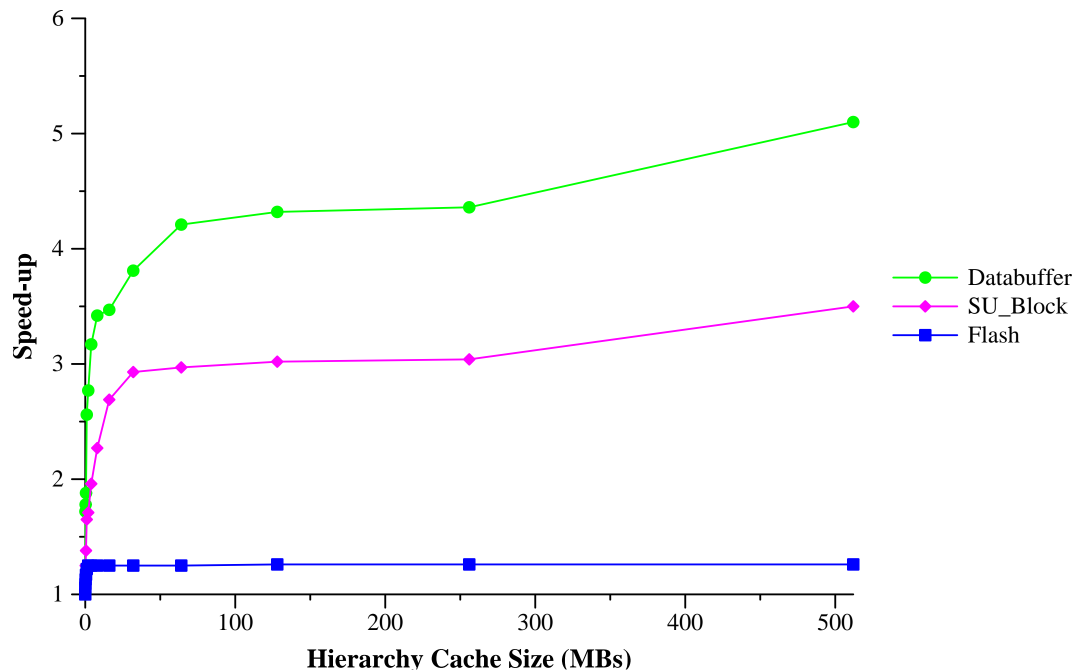


Figure 6.4: The size of the hierarchy cache is varied to show its effect on rendering speed-up.

that unlike all the other tables in this section, Figure 6.4 plots *speed-up*, not time, so higher is better in this figure. The cache size is varied from zero up to 512 MB. The SU_Block design shows cache effectiveness rolling off at about 32 MB while the Databuffer really flattens out at about 64 MB although the majority of the speed-up occurs also by 32 MB. Databuffer, being more hierarchical, shows a steeper slope and a higher final value than SU_Block, while Flash shows little speed-up with larger cache sizes since it contains little hierarchy to begin with.

Figure 6.4 generally matches the shape of the curve in Figure 4.7, that is, most of the speed-up possible can be obtained with a small amount of memory when compared to the amount of memory that would be required to pre-rasterize *all* sub-designs.

6.3.3 Parallelized Rendering Times

In Section 3.5 it was claimed that our chipmap rendering architecture allowed for an efficient parallel implementation. In this section, we put that to the test by rendering each

	Databuffer	SU_Block	Flash
4 processors	0.26	0.27	0.29
2 processors	0.51	0.51	0.51
1 processor	1.00	1.00	1.00

Table 6.4: Rendering times on machines with multiple processors.

	Databuffer	SU_Block	Flash
ChipMap - 4 processors	0.26	0.27	0.29
ChipMap - 2 processors	0.51	0.51	0.51
ChipMap - 1 processor	1.0	1.0	1.0
ChipMap - no hier. cache	3.8	2.9	1.3
Virtuoso	6.4	3.8	1.4
Magic	8.9	6.1	3.5

Table 6.5: Rendering times summary. Lower is better.

design on a 1, 2 and 4 processor machine. The times are shown in Table 6.4. For the four processor case, ChipMap was run on a 4× Sun Microsystems 400MHz Ultrasparc II processor system with 2 GB of memory. The speed-up was computed by comparing this time to the time using only a single processor on this system. Table 6.4 shows nearly ideal speed-up for the two processor and four processor case for the Databuffer design. The speed-up falls off a bit in the four processor case for SU_Block and Flash because of load imbalances. With Flash this is especially evident, as the imbalance created by the missing data of the internal memories results in an uneven distribution of work.

6.3.4 Rendering Speed Comparison Summary

The previous three sections have compared speed-ups and rendering times of ChipMap against differing hierarchy cache sizes, processors and other programs. Let us put all the information together to see the combined effects of our optimizations.

Table 6.5 shows the information from Tables 6.2, 6.3, and 6.4. All times have been normalized such that the configuration of ChipMap on 1 processor with a 32 MB hierarchy cache is 1.0. Table 6.5 shows that ChipMap's acceleration structures have a pronounced

	Databuffer	SU_Block	Flash
ChipMap	0.004 s	0.004 s	0.003 s
Virtuoso	9 s	12 s	5 s

Table 6.6: Full design view refresh times in seconds for ChipMap and Virtuoso.

combined effect on rendering speed. Running on four processors with a 32 MB hierarchy cache is between 5× and 25× faster than Virtuoso depending on the design, making ChipMap the fastest way to render a VLSI layout that we are aware of.

6.3.5 Refresh Speed Comparison

The previous section focused on rendering the full design view once. None of the data tested the time to refresh the display given an incremental change in viewpoint. Because ChipMap explicitly saves the display with textures, the time to refresh the display when the viewpoint changes incrementally is close to zero[2]. However, all other known VLSI layout editors/viewers have to completely redraw the entire design for even the smallest changes in viewpoint. This behavior is captured in Table 6.6, which shows the refresh time for each design in seconds. The times shown are not normalized.

As Table 6.6 shows, any type of fluid interaction with Virtuoso would be impossible at a full design view.

6.4 Memory Usage Comparison

The last performance metric we compare is memory usage. As Table 6.7 shows, the memory cost of ChipMap is dominated by the texture tile, hierarchy and the top pyramid caches. The sizes of the texture tile and top pyramid caches are not directly related to design size but more related to the resolution of the display. This design independence allows the total cost of ChipMap to remain small. The hierarchy cache size can be tuned to trade off rendering speed-up for memory footprint, although a modest size like 32 MB is usually sufficient.

[2]This time is dependent on the fill-rate of the GPU and can possibly be bounded by the vertical refresh rate of the monitor. On the Linux test system, this time varied between 3 ms and 11 ms.

	Databuffer	SU_Block	Flash
Texture Tile Cache	64 M	64 M	64 M
Hierarchy Cache	32 M	32 M	32 M
Top Pyramid Cache	16 M	23 M	12 M
Design Pyramid	1 M	4 M	10 M
ChipMap Total	117 M	125 M	119 M
OpenGL	32 M	32 M	32 M
Virtuoso	0?	0?	0?

Table 6.7: Memory usage in MBs for ChipMap, OpenGL with 32 MB of vertex array data and Virtuoso. The ChipMap statistics are broken down further in the top part of the table.

The memory usage for OpenGL and Virtuoso are also shown. Since Virtuoso is proprietary, it is impossible to know exactly the memory cost of rendering a design, but a first order guess would be that little or no extra resources are expended based on the size of the process footprint we observe while Virtuoso is running. The OpenGL memory footprint is also modest and the amount of memory that one could devote to vertex arrays is scalable and tunable in the same way as ChipMap's hierarchy cache size.

6.5 Summary

This chapter has presented results showing that ChipMap can render layouts as fast as or faster than any other system that we are aware of. Happily, this speed does not come at the expense of image quality. In fact, ChipMap's image quality is also the most accurate of any interactive system that we are aware of. Finally, ChipMap's architecture allows it to redisplay a layout in essentially zero time, giving the user the most fluid interaction of any system that we know about. The cost of ChipMap's enhancements is a modest amount of memory which is generally independent of design size and dependent on display resolution.

Chapter 7

Conclusions

This thesis has focused on improving large dataset visualization in VLSI design. We believe that our primary accomplishment was in solving the "layout redraw" problem that has plagued designers over the last decade. Secondarily, we demonstrated how the techniques we developed to help redraw could also be used help visualize other types of design information.

With current methods, as the designer attempts to view a larger and larger section of a VLSI layout, two undesirable things occur: the *time* that it takes to create the visualization is so great, perhaps dozens of seconds, that interactivity is impossible, and the *accuracy* of the visualization once it has been created is very poor. Both of these problems adversely effect designer productivity. By drawing on advancements in the field of computer graphics, we were able to create methods that address both of these problems.

In order to solve the redisplay speed problem, we defined a structure called a chipmap, which is an extension of mipmaps used in graphics, that formalized a way to view the tremendous amounts of data in large VLSI layouts, and we used a rendering architecture that allowed the viewpoint to change as quickly as the user desired. A key point is that we did not fundamentally change the initial time it takes to draw a layout. In fact, our methods are approximately equivalent to current methods in drawing a layout for the first time. What we did accomplish was to formally specify a way to re-use already rendered parts of the image given incremental viewpoint changes. Then, we decoupled the drawing of

the display from the creation of the displayable image with multi-threading so that viewpoint change speed could be unconstrained. Ultimately, an implementation using these techniques can redraw very efficiently because graphics hardware exists to accelerate the process.

To solve image accuracy, we also looked to the field of computer graphics. We used unweighted area sampling with a box filter as our rasterizing technique to anti-alias layout features, and we used standard computer graphics compositing techniques to combine those rasterized fragments. To increase accuracy even more, we exploited the fact that the relationship between rectangles from the same layer is known beforehand. Along the way in producing the image data we also found ways that the process could be further sped up. One way was to pre-rasterize hierarchical subcells so that they could be copied in directly during the image creating process; the other way relied upon multiple processors to create the image in parallel.

Using the chipmap architecture has a number of advantages. First, we are able to produce accurate images with a minimal memory overhead. Basically, the image data is produced directly from the design database each time. We were never forced to expand the layout to its full size image, even temporarily, which we showed could be many orders of magnitude larger than the layout in its canonical state. Another way to think about it is that the layout database is a compressed form of the layout image. Because we only decompress the visible portion, the amount of memory and computation required is always the minimum.

Our architecture is also very amenable to parallelization. This is possible because all of the rasterization takes place on the CPU, not the GPU. Current methods are fundamentally different because they pass off all rendering to the GPU, which becomes the rendering bottleneck, making an efficient parallel approach impossible since only one GPU exists. At the outset of this research, this parallelizing property was not something that was sought, but we believe it is an advantage to our approach. In fact, many CAD tools users develop on large compute machines which have 8, 16, or 32 processors, making our approach that much more attractive.

Another advantage of having the rasterization take place on the CPU is that efficient display over a possibly slow network is feasible. This is because the bandwidth of information sent to the GPU is determined not by the number of objects to draw, but by screen resolution, and also because image re-use even further reduces the bandwidth requirement. This enables the display of the largest layouts in situations where it previously would have been so slow as to be impractical.

Our implementation is solely a data viewer, no editing is possible. Extending this work to include editing, which is essential for a commercial tool, would not pose any fundamental new problems. We have stated that the time it takes to create the layout initially is about the same for our approach as it is for current methods. Since this is true, then the time to create any smaller part of the image initially is also the same. Since editing is about incrementally re-creating small parts of the display, we believe that our approach could support editing at least as well as current methods.

Besides solving the layout redraw problem, we also demonstrated a glimpse of what might be possible in the future with this technology. We showed how exaggerated features, texture overlays, and transparency could create new visualizations that convey powerful messages to the user. Ultimately, we believe that the usefulness of our research lies in this direction. In the future, we imagine a visualization environment for the designer which is rich in its ability to view all types of VLSI data, from high level code, to global routing, to waveforms, to manufacturing errors, and to layout.

At the outset of our research, the vision was: how can visualization help VLSI design? From that question, the vision was narrowed to a single application which seemed the most natural since it was a compelling problem with an equally compelling solution. We would like to address the question again: how can visualization help VLSI design?

We think the field is wide open. The reality is that the systems being sold by the major EDA vendors have visual interfaces deeply rooted in X11 and legacy display methods and it is generally a point of major irritation for its users.

We hope that readers find chipmap not only a compelling application on its own, but also an indication of the myriad things that are possible when you combine the fields of computer graphics, visualization, and VLSI design.

Bibliography

[Akeley and Hanrahan, 2001] Kurt Akeley and Pat Hanrahan. CS448A: Real-Time Graphics Architectures, 2001. http://graphics.stanford.edu/courses/cs448a-01-fall/lectures/lecture5/.

[Ansoft, 2002] Ansoft. Ansoft, 2002. http://www.ansoft.com/.

[Bentley and Friedman, 1979] J. L. Bentley and J. H. Friedman. A Survey of Algorithms and Data Structures for Range Searching. *ACM Computing Surveys*, 11(4), 1979.

[Brewer, 1994] Cynthia A. Brewer. *Color Use Guidelines for Mapping and Visualization*, chapter 7, pages 123–147. Elsevier Science, 1994.

[Cadence, 2002a] Cadence. Cadence, 2002. http://www.cadence.com/.

[Cadence, 2002b] Cadence. Virtuoso, 2002. http://www.cadence.com/.

[Dell, 2001] Dell. Dimension 8200 PC, 2001. http://www.dell.com/.

[Epic Games, 2000] Epic Games. Unreal, 2000. http://www.unreal.com/.

[Foley et al., 1996] James D. Foley, Andries van Dam, Steven K. Fiener, and John F. Hughes. *Computer Graphics Principles and Practice*. Addison–Wesley, 1996.

[Hseuh, 1979] M. Y. Hseuh. Symbolic Layout and Compaction of Integrated Circuits. *University of California, Berkeley, Tech. Rep. UCB/ERL/M79/80*, December 1979.

[Intel, 2000a] Intel. Moore's Law revealed, 2000. `http://www.intel.com/research/silicon/mooreslaw.htm`.

[Intel, 2000b] Intel. Pentium4, 2000. `http://www.intel.com/`.

[Kedem, 1982] G. Kedem. The Quad-CIF Tree: A Data Structure for Hierarchical On-line Algorithms. *19th Design Automation Conference Proceedings*, pages 352–357, June 1982.

[Keller and Newton, 1982] K. H. Keller and A. R. Newton. KIC2: A Low Cost, Interactive Editor for Integrated Circuit Design. *Digital Papers for COMPCON*, pages 305–306, 1982.

[Kuskin et al., 1994] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford Flash Multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[Magma, 2002] Magma. Magma, 2002. `http://www.magma-da.com/`.

[Mentor Graphics, 2002] Mentor Graphics. Mentor Graphics, 2002. `http://www.mentor.com/`.

[Moore, 1965] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.

[Naffziger and Hammond, 2002] S. Naffziger and G. Hammond. The Implementation of the Next-Generation 64b Itanium Microprocessor. In *ISSCC Slide Supplement*, page 276, February 2002.

[Nvidia, 2002] Nvidia. GeForce 4 Ti 4600, 2002. `http://www.nvidia.com/`.

[Ousterhout et al., 1984] John Ousterhout, G.T. Hamachi, R.N. Mayo, W.S Scott, and G.S. Taylor. Magic: A VLSI Layout System. In *21st Design Automation Conference Proceedings*, pages 152–159, June 1984.

[Ousterhout, 1981] John Ousterhout. Caesar: An Interactive Editor for VLSI. *VLSI Design*, pages 34–38, November 1981.

[Ousterhout, 1984] John Ousterhout. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on CAD*, pages 87–100, January 1984.

[Porter and Duff, 1984] Thomas Porter and Tom Duff. Compositing Digital Images. In *SIGGRAPH 84 Conference Proceedings*, pages 254–259, July 1984.

[Rambus, 2000] Rambus. RDRAM, 2000. `http://www.rambus.com/`.

[Redhat, 2002] Redhat. Redhat Linux, 2002. `http://www.redhat.com/`.

[Restle, 2001] Phillip J. Restle. Technical Visualizations in VLSI Design. In *38th Design Automation Conference Proceedings*, pages 494–499, June 2001.

[Segal and Akeley, 1999] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1), 1999. `ftp://ftp.sgi.com/opengl/doc/opengl1.2/`.

[Silvaco, 2002] Silvaco. Silvaco, 2002. http://www.silvaco.com/.

[Smith, 1995] Alvy Ray Smith. A Pixel Is *Not* A Little Square, A Pixel Is *Not* A Little Square, A Pixel Is *Not* A Little Square! Technical report, Microsoft, Inc., 1995. ftp://ftp.alvyray.com/Acrobat/6_Pixel.pdf.

[Solomon and Horowitz, 2001] Jeffrey Solomon and Mark Horowitz. Using Texture Mapping with Mipmapping to Render a VLSI Layout. In *38th Design Automation Conference Proceedings*, pages 500–505, June 2001.

[Sun, 2002] Sun. Blade2000, 2002. http://www.sun.com/.

[Synopsys, 2002] Synopsys. Synopsys, 2002. http://www.synopsys.com/.

[Tanner et al., 1998] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap: A Virtual Mipmap. In *SIGGRAPH 98 Conference Proceedings*, pages 151–158, July 1998.

[Williams, 1983] Lance Williams. Pyramidal Parametrics. In *SIGGRAPH 83 Conference Proceedings*, pages 1–11, July 1983.

[X Consortium, 1986] X Consortium. The X11 Windowing System, 1986. http://www.x.org/.