Cache Design for a Hardware Accelerated Sparse Texture Storage System

by

Wai Min Yee

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2004

Abstract

Hardware texture mapping is essential for real-time rendering. Unfortunately the memory bandwidth and latency often bounds performance in current graphics architectures. Bandwidth consumption can be reduced by compressing the texture map or by using a cache. However, the way a texture map occupies memory and how it is accessed affects the pattern of memory accesses, which in turn affects cache performance. Thus texture compression schemes and cache architectures must be designed in conjunction with each other.

We define a **sparse texture** to be a texture where a substantial percentage of the texture is constant. Sparse textures are of interest as they occur often, and they are used as parts of more general texture compression schemes. We present a hardware compatible implementation of sparse textures based on B-tree indexing and explore cache designs for it. We demonstrate that it is possible to have the bandwidth consumption and miss rate due to the texture data alone scale with the area of the region of interest. We also show that the additional bandwidth consumption and hideable latency due to the B-tree indices are low. Furthermore, the caches necessary for these textures can be quite small.

Acknowledgements

I would like to thank my supervisor, Professor Michael McCool.

I would like to thank my readers, Professors Craig Kaplan and Eugene Zima.

This research was funded by the National Science and Engineering Research Council of Canada (NSERC) and the University of Waterloo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In computer graphics, the mapping of images onto object surfaces rendered on a computer screen as in Figure 1.1 is a basic example of the use of **texture mapping**. With appropriate hardware support, texture mapping is a simple and effective way to enhance the complexity and quality of computer generated images. Moreover, texture mapping is not restricted to the pasting of images onto surfaces; sampling from a discrete representation of a vector valued field, or **texture map**, has been used to perturb surface normals [Bli96], render shadows [FFBG01], and compute functions [MGW01, MH99], among many other uses. In fact, texture mapping is considered to be a fundamental hardware shading primitive [MH99].

Using texture mapping typically results in millions of texture lookups per rendered frame, which results in a need for high bandwidth and low-latency access

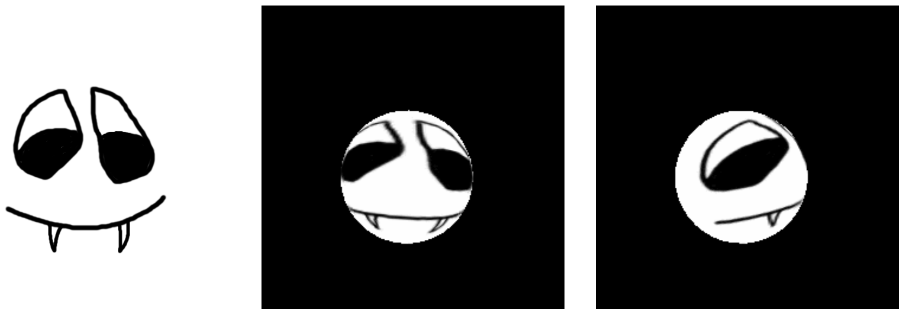


Figure 1.1: A face image texture mapped onto a sphere.

to texture memory. Unfortunately memory bandwidth is the resource that bounds performance in current graphics architectures [IEH99]. This motivates us to find ways to reduce memory bandwidth consumption.

One way to reduce bandwidth consumption is by compressing the texture map. The traditional implementation of a texture map is as a contiguous $2^{n_1} \times \cdots \times 2^{n_m}, n_i \in \mathbb{N}_0$ array. Each element in the array corresponds to a sample from the vector field the texture map is generated from, or **texel**. Texture compression makes it so that fewer bits on average need to be accessed for each texel lookup, scaling down the bandwidth consumption. Texture compression also reduces the memory footprint of a texture map. This is desired as high quality rendering requires high resolution textures, which when uncompressed take up a lot of memory, severely bounding the number of unique textures that can be accessed. Texture compression would increase the upper bound on the resolution and number of unique textures available to render an image.

Compression schemes can be divided into two types: lossy and lossless. Lossless compression schemes produce output from which the original data can be recovered perfectly. Lossy compression algorithms produce output from which the recovered data is an imperfect approximation of the original data.

In this thesis, lossless compression of what we consider to be a sparse texture is of particular interest. We define a sparse texture as the following:

sparse texture: A texture where a substantial percentage of the texture is some constant d. We call d the **default value** of the texture.

One common problem lossless sparse texture compression can be aimed at is the case where the region of interest is non-rectangular or not close in size to a power of two in each dimension, which wastes memory in a traditional implementation. A more general reason why lossless sparse texture compression is of interest is that it comes up repeatedly as a step in texture compression in general.

For example, using multiresolution to break down a texture into a hierarchy of textures, with each subsequent level refining the detail, is an example of creating an **adaptive texture map**, which is based on **adaptive sampling** from the original texture. Adaptive sampling, where the sampling density is correlated with the need for detail *i.e.* high sampling density in high detail areas and low sampling density in low detail areas, is frequently used in software implementations of data compression where interpolation is used to fill in "missing" values, as the quality/space tradeoff for a particular area is easily controlled locally and independently of other areas. As the detail added at the higher resolution levels is sparse, sparse textures are particularly suited for this type of data. Examples of ways to decompose a texture to produce sparse textures include adaptive sampling, wavelet compression, and predictive pyramidal coding.

Another way to reduce bandwidth consumption is based upon the Principle of Locality [HP03] which can be divided into two parts:

Spatial locality: If items x and y are close together in memory address, then they are likely to be accessed close together in time.

Temporal locality: If item x is accessed recently, it is likely to be accessed again in the near future.

If the Principle of Locality applies to a stream of memory accesses, then memory bandwidth consumption can be reduced by the use of a small fast local memory holding recently accessed items that is checked before going to slower memory to retrieve the item; this local memory is called a **cache**. There is a high degree of both spatial and temporal locality in texture map accesses so using a cache for texture memory reduces bandwidth consumption [HG97]. Previous work on caches for texture mapping has focused on tuning rendering and memory representation issues that affect locality and characterizing the cache parameters needed for texture mapping.

While there has been work on both texture compression and on cache design for texture mapping, there has been no work combining the two. The effect texture compression has on cache design is of concern as the change in the way a texture occupies memory and how memory is accessed affects the pattern of the memory accesses, which in turn could affect cache performance and memory bandwidth consumption. Thus it is necessary, when examining a texture compression scheme, to also look at how using it affects the cache performance.

Contributions

Of the previous work on texture compression, the only method that both provides adaptive random sampling and is compatible with a hardware implementation is the uniform grid index method devised by Kraus and Ertl [KE02]. We test the cache performance of this method and show that we do not get significant bandwidth consumption savings for a sparse texture unless the cache is very large. Furthermore, this method requires the use of dependent texturing which adds latency to the memory accesses. This motivates an alternative implementation.

We present an implementation of lossless sparse texture compression based on B-tree [LD91] indexing. The extra memory needed due to the index is low and scales with the area of the region of interest instead of with the resolution of the texture (as in the uniform grid index method) resulting in a smaller index memory overhead. With an appropriate cache design, the latency due to the indexing is low and hideable, and the additional bandwidth consumption due to the B-tree index is negligible. This implementation does not have many of the disadvantages that the uniform grid method has; a detailed comparison is given in Chapter 4.

We also build upon previous work on caches for texture mapping to show which rendering and cache parameters work best for our implementation. Though we tune the rendering parameters and the cache design to our B-tree indexing scheme, the results on the rendering parameters are still applicable to texture mapping in general and the results on the cache design can apply to any sparse texture scheme based on storing only the necessary data and accessing the data indirectly via an index.

Outline

Chapter 2 reviews the background necessary to understand the thesis. Chapter 3 covers previous work in texture compression and cache design for texture mapping.

Chapter 4 describes our implementation of sparse textures, cache design for it, and the motivation behind it. The setup for testing is detailed in Chapter 5. Chapter A gives the results and analyses. Conclusions and future work are in Chapter 7.

Chapter 2

Background

Before we proceed to the research contributions, we give a general overview of the graphics pipeline, texturing, and caching [HP03]. The overview will establish the necessary background and vocabulary for the discussion to follow.

2.1 The Graphics Pipeline

Interactive 3D graphics hardware imposes a series of transformations on a scene description to produce a final picture. The sequence of stages the scene description goes through are called the **graphics pipeline**. There are many implementations of the graphics pipeline athich vary in structure, features and the order in which they perform their operations [McC00, Owe02, Ige00, Bli96], but for our discussion

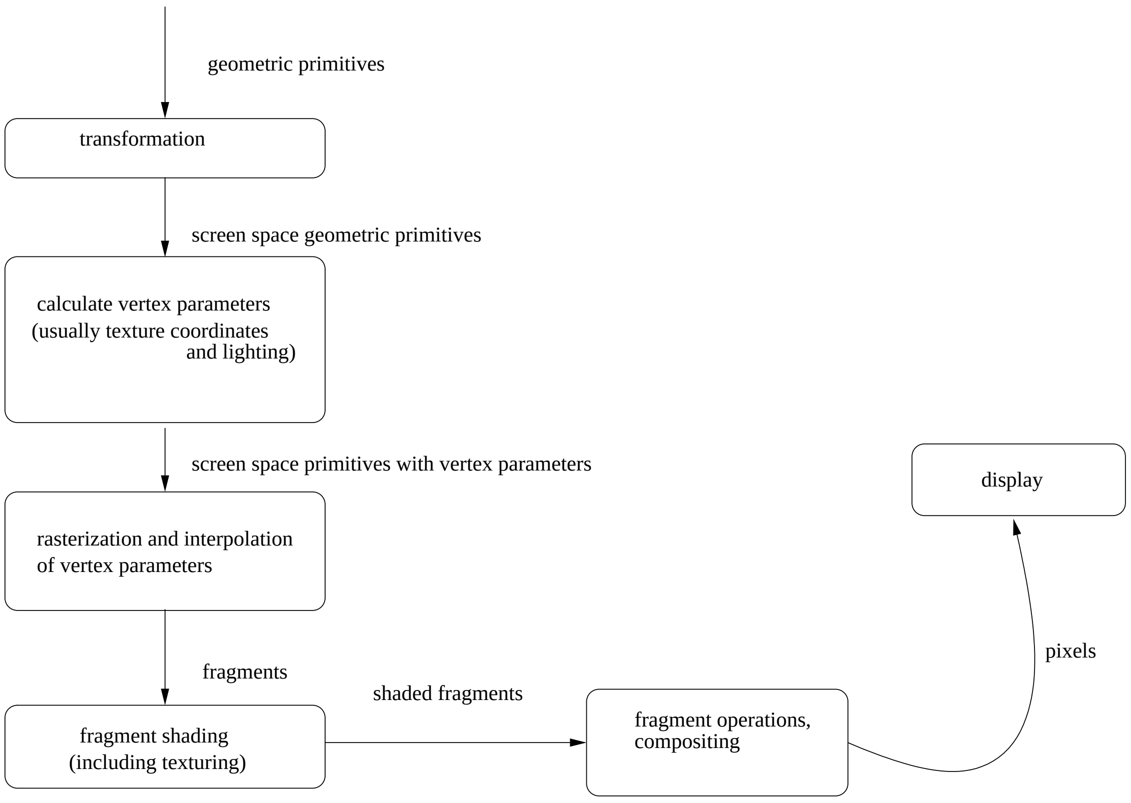

geometric primitives
transformation
screen space geometric primitives
calculate vertex parameters (usually texture coordinates and lighting)
screen space primitives with vertex parameters
rasterization and interpolation of vertex parameters
fragments
fragment shading (including texturing)
shaded fragments
fragment operations, compositing
pixels
display

Figure 2.1: A representation of a generic graphics pipeline.

we present a generic graphics pipeline into which most of the implementations described in the references fit. This generic pipeline is shown in Figure 2.1.

We are mainly concerned with two stages in the pipeline: rasterization and interpolation and fragment shading.

2.1.1 Rasterization

In the rasterization stage of the pipeline, each geometric primitive is broken up into fragments, each fragment corresponding to a pixel that the primitive intersects in screen space. Each primitive may also have parameters associated with each of its vertices. The vertex parameters could be colours, texture coordinates, vertex normals, or other properties used to enhance the rendered image. These vertex parameters, if present, are interpolated across the surface of the primitive to give each fragment an associated set of interpolated parameters.

The order in which the fragments are produced is called the **rasterization order**. We use the term "**rasterization curve**" to refer to a path drawn through the pixels in screen space in the order they are visited by the rasterizer. Because we expect texture accesses to follow the rasterization curve, and caches depend on the principle of locality to work well, improving the spatial locality of the rasterization curve will have a significant impact on cache performance.

For simplicity of discussion, we assume for now that the only geometric primitive we have to deal with is the triangle. We also assume that the rasterization units are rasterizing one triangle at a time, and that all triangles from a single triangle are processed serially within a single fragment unit.

Horizontal scanline order, or **raster order** rasterization [OG97], is shown in Figure 2.2. The fragments are generated row by row. This order has the advantage that the rasterization curve is easy to generate. It has the disadvantage of having poor spatial locality between rows.

There has been recent work [MWM01, MM00, HG97] on other kinds of rasteriza-

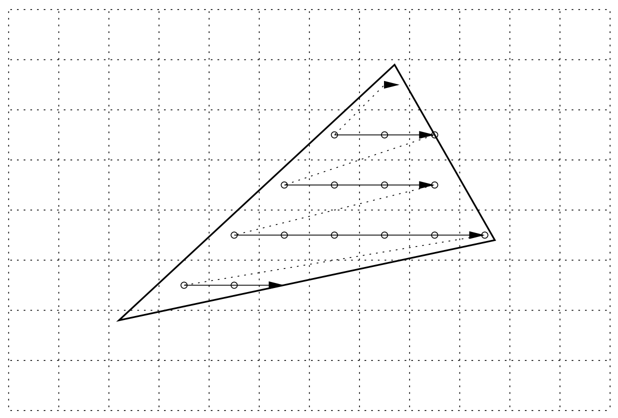


Figure 2.2: A triangle rasterized in horizontal scanline order.

tion with one motivation being better spatial locality for better cache performance. One of the rasterization curves investigated [MWM01] is based on the Hilbert curve [Sho38]. Consider the curve H_0 in Figure 2.3 and the set of rewriting rules in Figure 2.4. Let H_0 be as shown and define H_{n+1} to be the result when the curve H_n has had the rewriting rules applied to all its cells (see Figure 2.3). The two-dimensional Hilbert curve H is the space-filling curve we obtain when we take the limit of H_n as $n \to \infty$. For rasterization, we select n large enough so that we visit adjacent pixels as we go along each straight segment of the curve H_n. Figure 2.5 shows a triangle being rasterized in Hilbert order. Note that the high spatial coherence of the curve translates into relatively few large jumps as the triangle is rasterized.

Another space-filling curve is the Morton or Z-order curve, shown in Figure 2.6. Similarly to the set $\{H_n\}$, there is an obvious rewriting rule from which we obtain

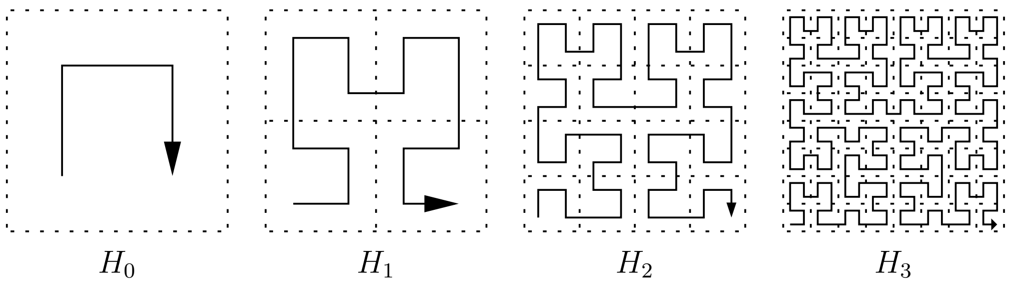


Figure 2.3: The first four iterations of the 2D Hilbert curve.

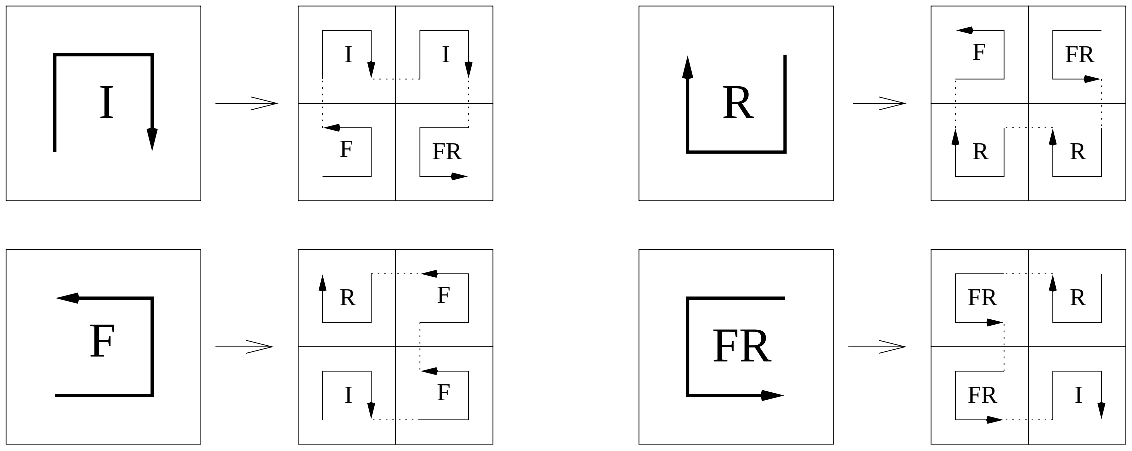


Figure 2.4: Rewrite rules for the Hilbert curve. [MWM01]

the set $\{Z_n\}$, of which the first few members are shown in the figure. While having a hierarchically self-similar structure like the Hilbert curve, it is not as spatially coherent as evidenced by the large jumps that can be seen in Z_3. It does have the advantage of being simpler to generate than the Hilbert curve. In fact, the Morton curve can be generated by bit interleaving the binary representations of x and y [OM84].

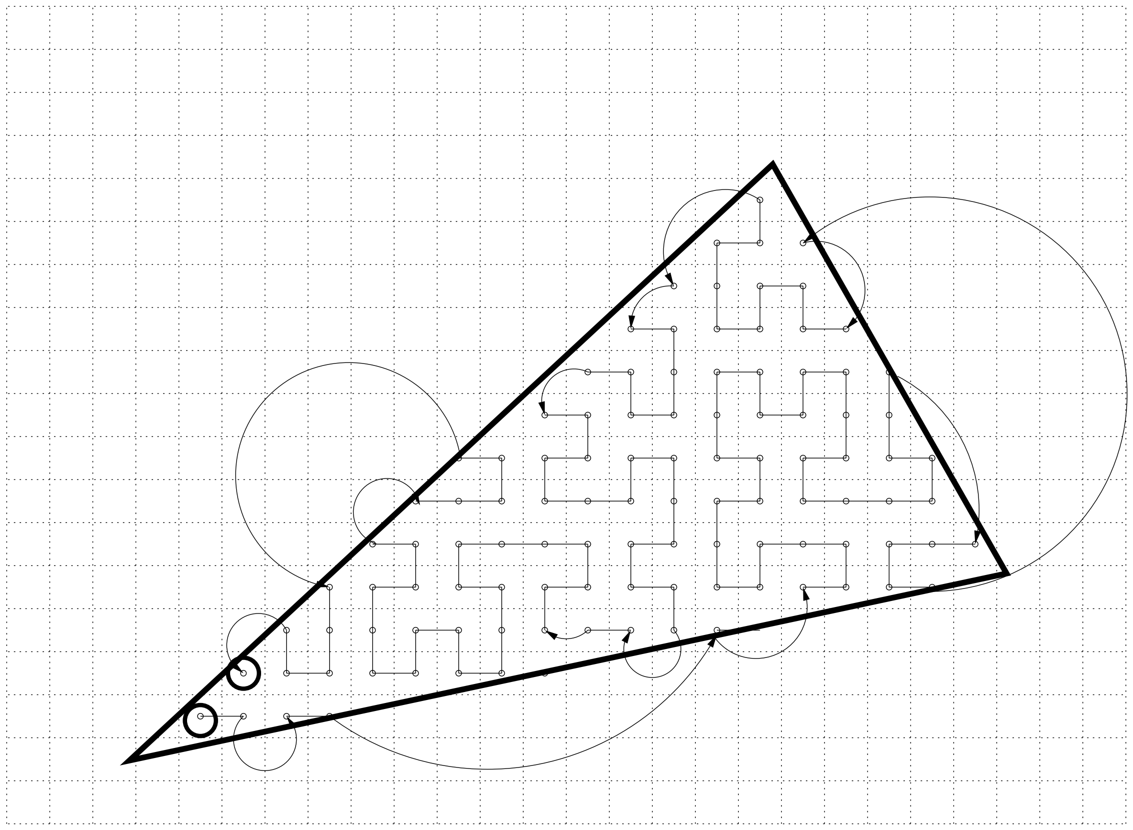


Figure 2.5: Triangle rasterized in Hilbert order.

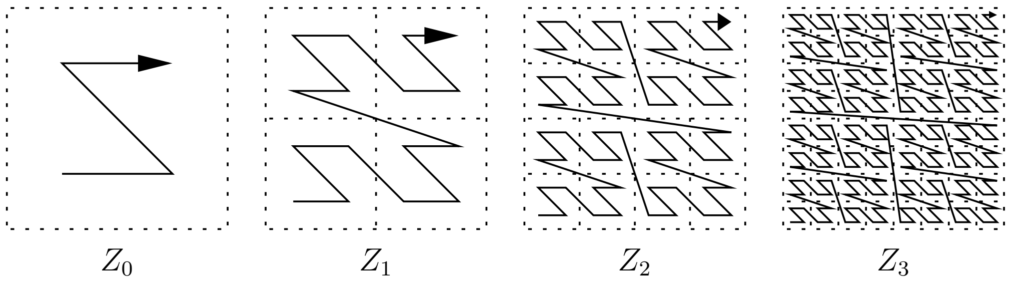


Figure 2.6: The first four iterations of the Morton curve.

While we have only run tests with a rasterizer that processes one triangle at a time using the raster order curve, Hilbert order curve, and the Z-order curve, there are many other rasterization options. Groups of triangles or primitives may be merged and rasterized together instead of individually. Triangle merging is particularly useful when there are many small triangles. Object order rasterization methods divide the primitives into groups, rendering each group of primitives in turn. Rasterization may also proceed in image order, dividing up the final image into parts and rendering each part in turn. Tile-based rasterization, shown in Figure 2.7, may also be employed. Nesting tiles is also possible. We can think of the Hilbert and Z-order curves as nesting the tiles in a particular order down to the pixel level. yield many other rasterization orders.

2.1.2 Shading and Texturing

The fragment shading part of the pipeline is where each fragment is assigned a colour by the fragment shader. The fragment shader uses the interpolated vertex parameters and other pipeline state parameters to compute a colour for the fragment. Sampling from texture maps may be used in determining the colour. If the sampling rate for the fragments is lower than the resolution of the texture map being sampled from, aliasing artifacts will occur. To minimize aliasing, filtering

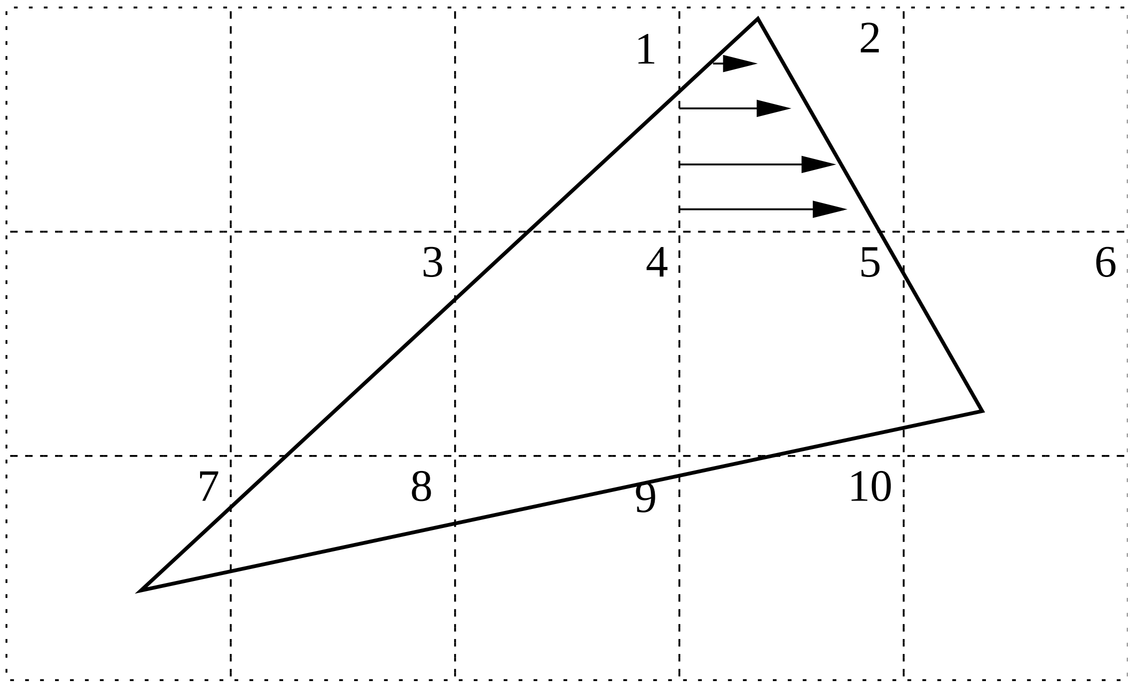


Figure 2.7: Tiled rasterization. The parts of the triangle in the tiles are rasterized as shown in tile 2 in the order the tiles are numbered.

is used. We will be concerned with two filtering techniques: MIP-mapping and FELINE sampling.

MIP-mapping

The most common hardware method for filtering is MIP-mapping [Wil83]. MIP-mapping involves constructing an image pyramid from the original texture, where each subsequent level is a half-scale copy of the image in the previous level (see Figure 2.8). Suppose we wish to sample a texture at coordinates (u, v) in the

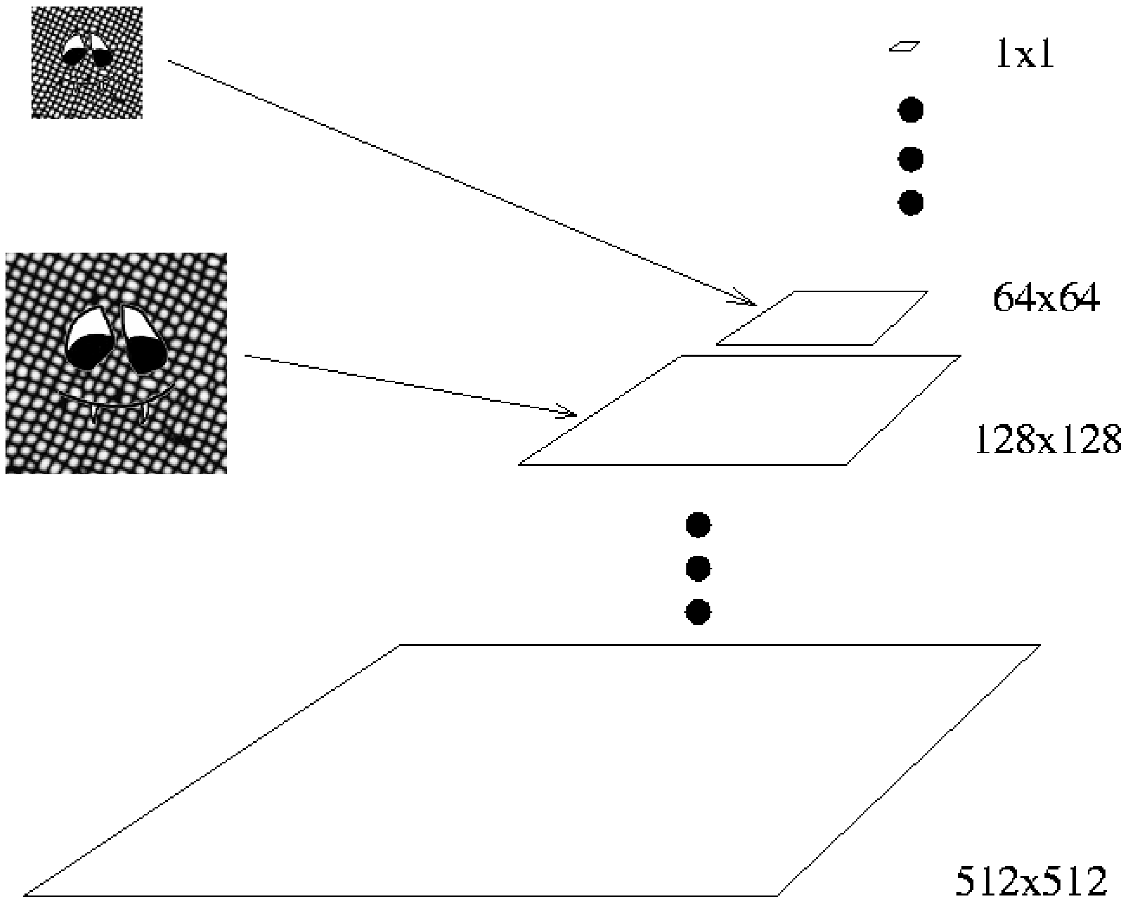


Figure 2.8: Image pyramid for MIP-mapping.

processing of a fragment at screen location (x, y). The derivatives $\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial y}$, and the base texture resolution are used to compute a level-of-detail (LOD) parameter λ [Wil83]. We think of λ as a "measure" of the texel-to-pixel ratio. This value is used to select two adjacent images in the pyramid to sample from. At each level the four nearest values in each level are bilinearly interpolated. The fractional part of λ is then used to linearly interpolate the two samples to produce the final result. This way of sampling is also called trilinear interpolation, or a trilinear probe. If the value of λ is such that it only makes sense to sample from one image, then bilinear interpolation is done on the four nearest values in that image.

When MIP-mapping is enabled, it keeps the texel-to-pixel ratio less than or equal to one so we do not get large jumps across the texture on the majority of consecutive texture accesses. Thus using MIP-mapping makes the memory accesses more coherent than they might be if MIP-mapping were not enabled. Therefore, MIP-map filtering has beneficial effects on both quality and bandwidth consumption.

Note that regardless of the relative ratio of the lengths of the derivative vectors $D_x = (\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x})$, and $D_y = (\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y})$, the same shape reconstruction filter (the trilinear probe) is used. The only thing that changes is its width, characterized by λ. While a single trilinear probe is fine if the derivative vectors look like Figure 2.9(a) up to rotation, excessive blurring occurs in the cases depicted in the other three examples in the figure [MFPJ99] (OpenGL implementations select the level based on the longer vector [SA02]). An anisotropic filtering algorithm is needed to deal with the blurring, in which case the pattern of texture accesses varies from fragment to fragment as the derivative vectors vary. This may make texture accesses less spatially coherent, affecting cache performance.

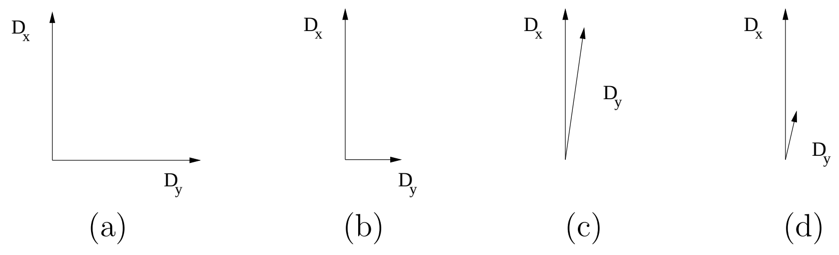


Figure 2.9: Ways in which the vectors D_x and D_y can be oriented relative to each other.

FELINE Sampling

One recent hardware algorithm developed for anisotropic texture mapping is the FELINE (Fast Elliptical LINEs) algorithm [MFPJ99]. Like MIP-mapping, it uses an image pyramid and computes a LOD value based upon the values of D_x and D_y. However, instead of using a single trilinear probe into the pyramid, it uses several trilinear probes along a line to approximate an elliptical Gaussian filter (see Figure 2.10). The filter to approximate and the line to use are computed based upon the values of D_x and D_y and take their relative lengths and orientations into account. The same λ value is used for each probe, though each the results of each probe is weighted differently.

The computations to determine the ellipse for the filter and the line to sample along can be expensive. As a result, a cheap approximation is desired. One approximation, table FELINE, uses a 2-D table of precomputed elliptical parameters

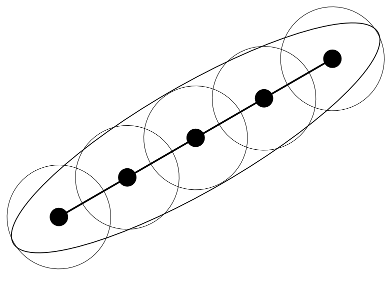


Figure 2.10: Probing along a line. The circles represent trilinear probes. The ellipse represents the elliptical Gaussian filter that the probes approximate.

to approximate the ellipse. Each table entry corresponds to the ellipse determined by two vectors, the longer one being $(0, 1)$, and the angle θ between them, which is between 0 and $\pi/2$. The table is indexed by the length of the smaller vector and a function of θ based on the tangent and cotangent functions. Naturally, (D_x, D_y) must be converted into the corresponding canonical form to use the table and the results from the table must be altered to take the conversion into account.

There has been other work on hardware anisotropic texture mapping methods [MFPJ99, Hec86]. The methods are either similar to FELINE (in that space invariant probes along a line are used), have high memory and computation time costs, or are inappropriate for sparse textures.

2.1.3 Programmable Shader Issues

A programmable shader may produce unusual patterns of texel lookups. We choose a variation on Perlin's *turbulence* function [EMP+94] as a representative of a programmable shader in our tests.

Turbulence

Similarly to the way any function satisfying certain conditions can be constructed from the sum of various sinusoidal functions via the inverse Fourier transform [Gla95], a noise function with a particular power spectrum can be constructed as the sum of various band-limited noise functions.

For example, given a band-limited noise function `noise`, we can have [EMP+94]:

```
float my_noise( point Q )
{
    value = 0;
    for( f = MINFREQ; f < MAXFREQ; f *= 2 )
        value += amplitude( f ) * noise( Q*f );
    return value;
}
```

which is built upon `noise`. The point to sample from is `Q`. The desired power spectrum for `my_noise` is determined by the function `amplitude( f )`, where `f` is considered to be the frequency variable.

Perlin's turbulence function uses `amplitude( f )` $= \frac{1}{\mathtt{f}}$. Additionally it uses the absolute value function to introduce discontinuities in the first derivative of the resulting function.

```
float turbulence( point Q )
{
    value = 0;
    for( f = MINFREQ; f < MAXFREQ; f *= 2 )
        value += abs( noise( Q*f ) ) / f;
    return value;
}
```

For our purposes we use a slightly modified version of Perlin's turbulence function as found in `povray 3.50` code. There are two extra parameters, λ and ω. The variable `f` and function `amplitude( f )` in the i-th iteration of the loop are defined to be `f` $= \lambda^i$ and `amplitude( f )` $= \omega^i$.

```
float test_turbulence( point Q, float $\lambda$, float $\omega$ )
{
    value = 0;
    amp = $\omega$;
    for( f = $\lambda$; f < MAXFREQ; f *= $\lambda$, amp *= $\omega$ )
        value += abs( noise( Q*f ) ) * amp;
    return value;
}
```

We control the degree of turbulence in our tests by varying ω. Note that $\omega = 0$ corresponds to no turbulence. The turbulence shader used to perturb texture coordinates applied to a checkerboard is shown in Figure 2.11. The different MIP-map levels have been assigned different colours. The turbulence shader applied to a picture is shown in Figure 2.13. The checkerboard example except using table FELINE instead trilinear sampling is shown in Figure 2.12.

We choose a shader that perturbs the texture coordinates using a modified version of Perlin's turbulent noise because it exemplifies two properties that are common causes of poor locality due to shaders: distortion and discontinuities. The distortion is obvious in Figure 2.11. Discontinuities are approximated by the discontinuities in MIP-map level that occur. Both of these properties are controlled by the parameter ω.

Shader Derivatives

The derivatives $\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, $\frac{\partial u}{\partial y}$, and $\frac{\partial v}{\partial y}$, which are the partial derivatives of the texture coordinate components with respect to the screen space coordinates, are used to compute a level-of-detail value, λ, which we denote here as λ_{LOD}. This value is used to select the MIP-map levels to sample from. Normally these derivatives are computed as part of the rasterization process, and any distortions introduced by the shader do not figure into the computations. This means that there may be a mismatch between the sampling rate a shader uses on a texture and the texture resolution, which can lead to poor spatial locality of the texture accesses and bad cache performance. Since the turbulence shader we use can produce significant distortion, we add as a test parameter whether or not to use **shader derivatives** to compute λ_{LOD}.

2.2 Caches

Due to filtering and interpolation many texture lookups may be needed to texture each fragment. Current processors are fast enough that memory latency and memory bandwidth needed by texturing are the bottleneck that limits performance

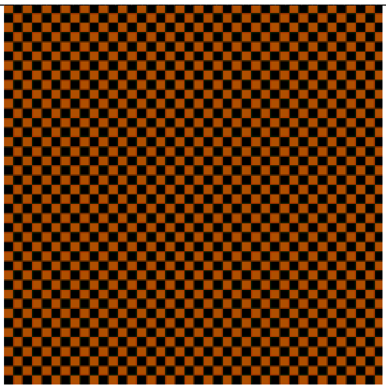
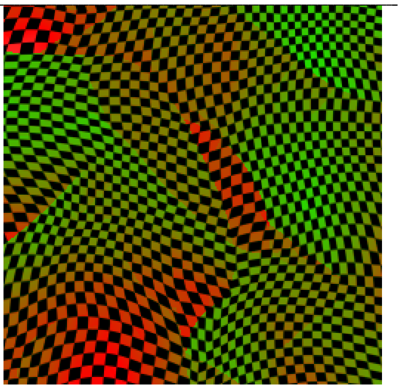
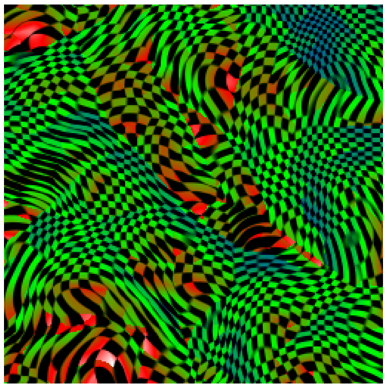
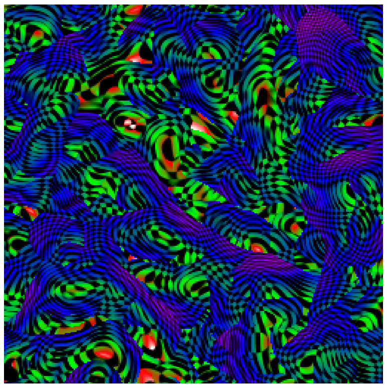
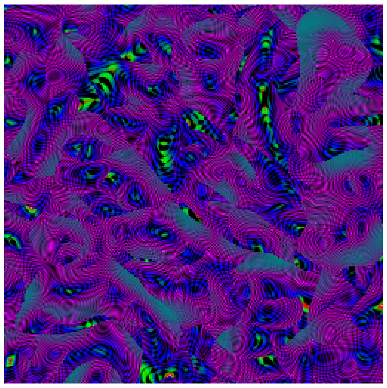
ω	Picture	ω	Picture
0.0		0.2	
0.4		0.6	
0.8			

Figure 2.11: Turbulence shader applied to a checkerboard for various values of ω.

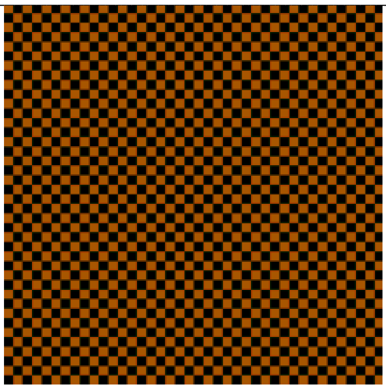
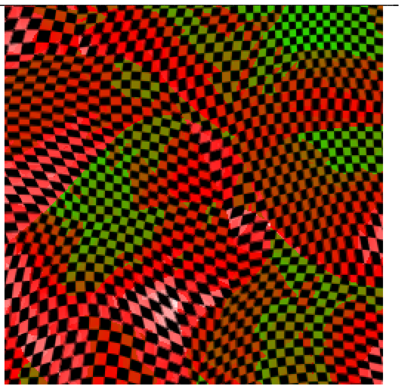
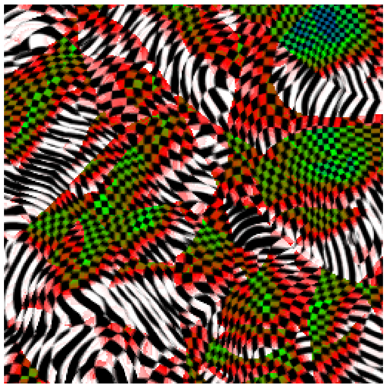
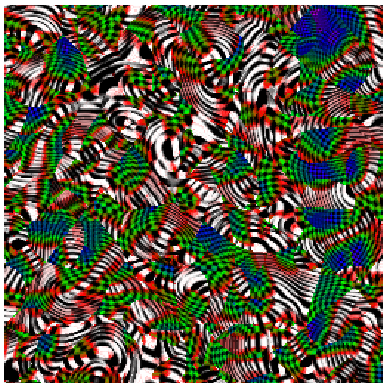
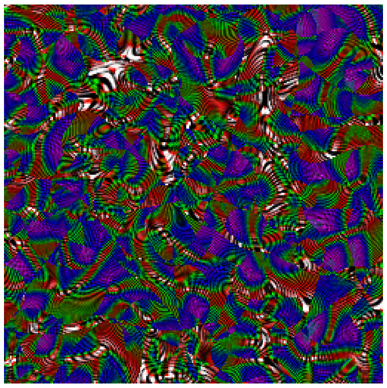
ω	Picture	ω	Picture
0.0		0.2	
0.4		0.6	
0.8			

Figure 2.12: Turbulence shader applied to a checkerboard for various values of ω with table FELINE sampling used.

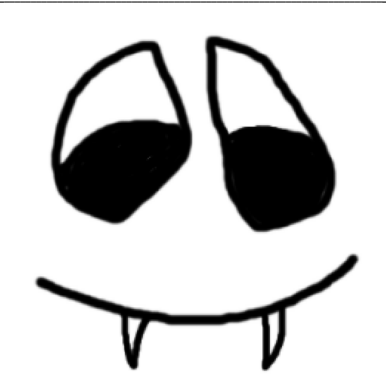
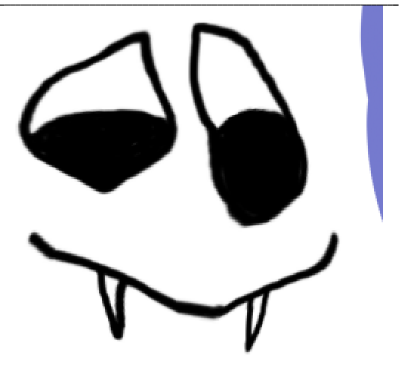
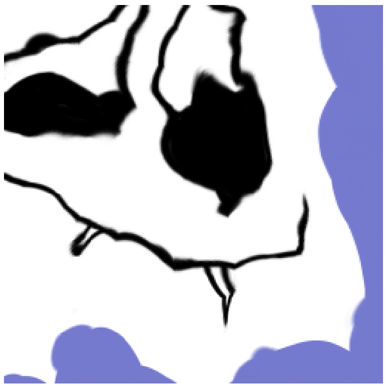
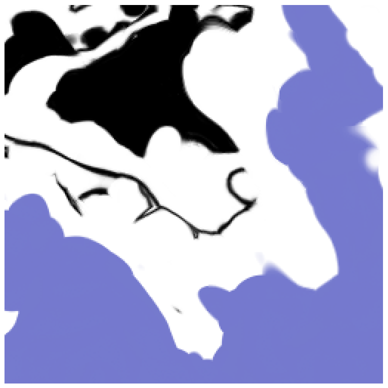
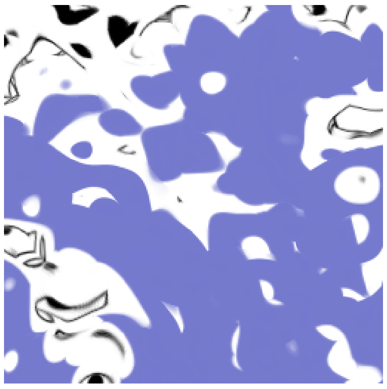
ω	Picture	ω	Picture
0.0		0.2	
0.4		0.6	
0.8			

Figure 2.13: Turbulence shader applied to a picture for various values of ω.

in graphics pipelines [Ige00]. Therefore we can significantly improve performance by decreasing the bandwidth consumption and latency in texture access. Unfortunately, high-speed memory is much more expensive per bit than low speed memory. We can exploit the locality of texture access by storing recently used items in a small **cache** of high speed memory that mediates processor access to main memory.

The cache divides a linear address space into contiguous **blocks**. The **block** is the basic atomic unit stored by the cache. A cache of size N bytes holds $\frac{N}{\text{block size in bytes}}$ blocks. A request for a block that is present in the cache is called a cache **hit**; the **hit time** is the time it takes to find the requested item in the cache. If the block is not in the cache the request generates a cache **miss**. The additional average cost above the hit time of a miss is called the **miss penalty**.

2.2.1 Placing Items in the Cache

Consider a cache that holds m blocks in m cache entries. We can divide the m entries into p sets of n entries ($m = n \times p$). The cache is **n-way set associative** if a block can be placed in only one of the p sets, but anywhere within the n entries in that particular set.

The special case where $p = m$ (and $n = 1$) is called **direct mapped**. The special case where $p = 1$ (and $n = m$) is called **fully associative**.

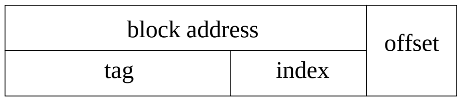


Figure 2.14: Decomposition of a memory address into the various parts used by the cache. The index field selects the set, the tag field distinguishes the block from other blocks with the same index field, and the offset is used to select the byte in the block.

Figure 2.14 shows how the block address is decomposed and Figure 2.15 shows how it is used. The index field, which is equivalent to the block address modulo p, is used to select a set S. The block's data is placed in a cache entry belonging to S. The cache entry also holds a tag which is used to distinguish the block it contains from other blocks that could be placed in the same set. The tag field portion of the block's address (*e.g.* block address div p), is placed in the cache entry as the tag for that entry.

When a lookup is performed, the index field is used to select the appropriate set S (this step is unnecessary when the cache is fully associative). Then the tags of every entry in set S are compared to the tag field, usually in parallel for speed. The appropriate entry is selected if there is a match; otherwise a miss is generated. The block offset is used to select the appropriate byte in the block.

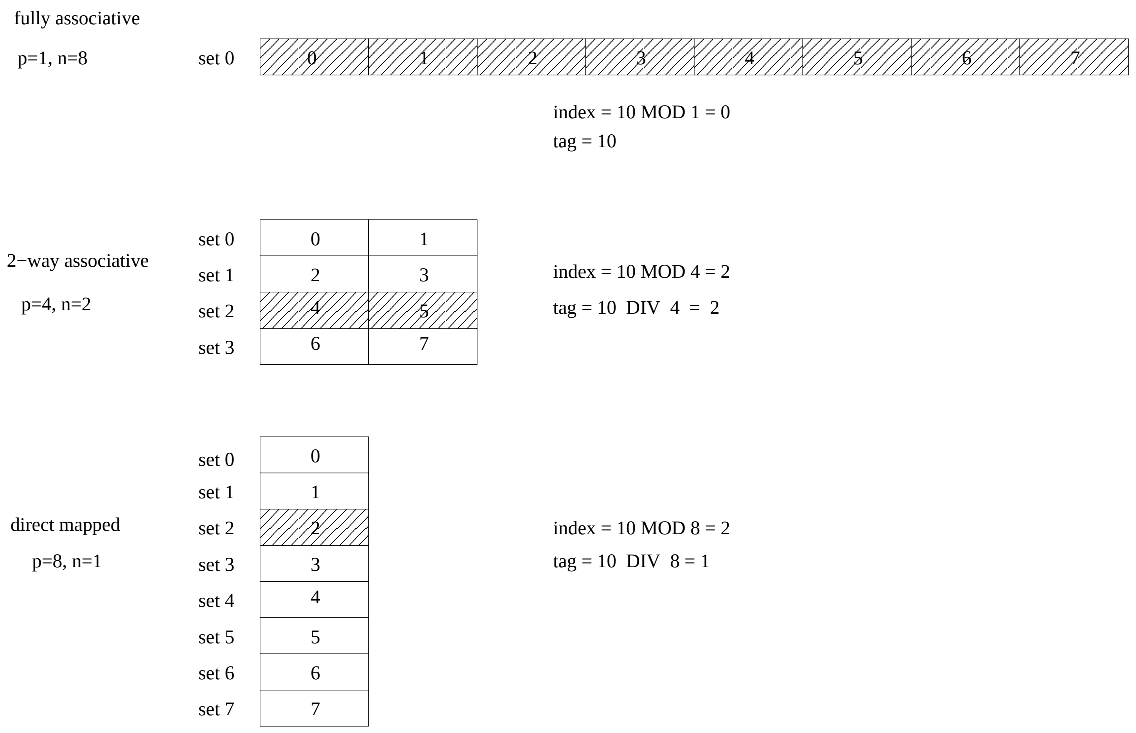


Figure 2.15: From top to bottom, we have a fully associative cache, a 2-way set associative cache, and a direct mapped cache. All caches can hold eight blocks in p sets of size n. The blocks in each cache are arranged so that each row corresponds to one of p sets. The shaded areas show where a block for block address 10 can be stored in each cache; they correspond to set 10 mod p *i.e.* the index field is 10 mod p.

2.2.2 Eviction Policies

Since a cache is smaller than the memory layer whose blocks it stores, eventually more blocks will be mapped to a set than the set can contain. As a result a block will have to be evicted from the set to make room for the new block. In the case of a direct mapped cache, the choice of which block to evict is simple. Otherwise we need an **eviction policy**, a method of choosing which block to evict.

The eviction policies used in our tests are:

optimal: Assuming knowledge of the block requests in the future, the block needed farthest in the future is evicted. This cannot be implemented in practice.

least-recently-used (LRU): The least recently used block is replaced. LRU attempts to take advantage of temporal locality but is expensive to calculate.

first-in-first-out (FIFO): Blocks are evicted in the order they entered the cache. FIFO cheaply approximates LRU when the order in which blocks enter the cache closely corresponds to the reverse of the order in which they were last used.

There are other eviction policies (e.g. random, least-frequently-used), but we do not discuss them here as they have not been used in our tests.

2.2.3 Measuring Performance

One of the most important measures of cache performance is the ratio $\frac{\text{number of misses}}{\text{number of memory accesses}}$, which we call the **miss rate**. However, the miss rate is not necessarily a good metric for overall performance. A better metric is the **average memory access time**, given by:

$$\text{average memory access time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

We can improve the processing speed by reducing the miss rate, reducing the hit time, and/or reducing the miss penalty.

2.2.4 Reducing the Miss Rate, Miss Penalty, and Hit Time

A miss occurs because a block that is requested is not in the cache. We can divide the misses into cold misses, capacity misses, and conflict misses [HP03].

Cold (or compulsory): The misses that occur in a cache of infinite size. These count the first time each memory block is brought into the cache.

Capacity: The misses that occur in a fully associative cache of finite size. These occur because the cache was too small and the missed block was previously evicted.

Conflict: The misses that occur in a non-fully associative cache of finite size. These are caused by a non-uniform distribution of requests. Some sets are more popular than others, resulting in a smaller effective cache size.

One way to reduce the number of cold misses is to increase the block size, so that there are fewer unique blocks that need to be loaded. However, increasing the block size results in an increased miss penalty as it takes longer and more bandwidth is used to load a new block. Also, the other types of misses may increase or decrease depending on the cache size and the locality of memory accesses.

Capacity misses may also be reduced by increasing the size of the cache. Increasing the cache size is expensive and will make the hit time longer for the same die area. Hit time will greatly increase if the cache is no longer small enough to fit onto the same chip as the processor.

Conflict misses are reduced by increasing the associativity of the cache. Unfortunately, increasing associativity means a higher hardware complexity cost for the same size of cache, which may increase the hit time and memory access time for similar reasons as increasing the cache size.

There is another class of approaches for reducing conflict misses without increasing the hit time for a fixed hardware cost. Way prediction and pseudoassociative caches approximate associative caches by first doing a tag check for one cache entry in the set of cache entries for the block, then checking the other tags in the set. The average hit time is reduced if the memory access pattern is such that the effect of the fast hit time provided by the first tag check dominates the effect of the slower hit time due to the other tag checks. Unfortunately it is possible that the memory access pattern may result in a higher average hit time or miss penalty than that for an associative cache of the same size.

Non-cold misses in general can be reduced by changing the way the data is arranged in memory and changing the way the data is accessed to produce better locality.

Miss penalty reduction methods include using multi-level caches to give a graduated range of miss penalties and sending the word in the block requested first. Victim caches can be used to hold recently evicted entries that may likely be requested in the near future to reduce the penalty for loading them into the cache. A non-blocking cache can reduce the effective miss penalty because cache hits can be served while the miss is dealt with. Prefetching of texture data can lower both the miss penalty and the miss rate.

Hit time is reduced at the expense of increased miss rate by keeping caches small and simple. It can also be kept low by not having to do any address translation, say translating a virtual address to a physical address. Pipelining the cache access can increase the rate of texel retrieval without actually decreasing the hit time.

Some techniques for improving cache performance not mentioned here are not directly relevant to caches for texture mapping. For example, giving the more common read misses priority over write misses is not relevant to texture mapping as the graphics processing unit partitions reads and writes so that they are not finely interleaved. Also, we only care about caching data and are not concerned with caching instructions.

Chapter 3

Previous Work

We cover previous work in texture compression and cache design for texture mapping.

3.1 Sparse Textures and Texture Compression

Compressing sparse textures is a special case of texture compression in general. Unlike image or data compression, the way textures are used imposes additional constraints other than a good compression rate on a texture compression scheme. The most important of the requirements stated by Beers *et al.* [BAC96] are:

Fast decompression: required to minimize the latency of each texture access.

Random access: needed because we typically access only small parts of the texture at a time, so there should not be any need to load and decompress the whole texture just to access a small portion of it.

The methods used to compress textures generally fall into three categories: fixed-rate compression schemes, hierarchical data structures, and rearranging the useful parts of the texture.

Most uses of textures involve two-dimensional textures resulting in most of the algorithms being designed for two dimensions and described in two dimensions. Unless otherwise noted assume two-dimensional textures, blocks, coordinates, *etc.*

3.1.1 Fixed-Rate Compression Schemes

Due to the random access requirement, most of the work in hardware texture compression and rendering has focused on fixed rate encoding schemes *i.e.* the final compressed texture size, ignoring auxiliary tables and other info, is a linear function of the resolution of the texture.

Fixed rate schemes are based on reducing the number of bits needed to represent blocks of texels so they can be reconstructed independently of each other. Approaches generally fall into two categories. One approach replaces each block of texels with an index into a small table. The other approach replaces each block of texels with a smaller block from which a “reasonable” approximation of the original block can be reconstructed.

The most familiar example of the first approach is colour palettizing, in which each texel is replaced with an index, which has fewer bits than the texel, into a palette. This is a special case of the approach used by Beers *et al.* [BAC96], which replaces 4×4 blocks of texels with an index into a codebook obtained using vector quantization (VQ) [GG91]. Both methods need special hardware to convert the texel location and index into a texel. Unfortunately, a codebook/palette needs to be loaded each time the application switches textures, and some types of textures cannot be adequately represented without a large codebook/palette. A codebook could also potentially use a lot of the cache and memory resources. However, the most important problem with indexing is the latency and memory usage from the extra memory accesses. This leads to the second approach, parameterized block approximation, which does not use indirection.

Examples of the second approach are briefly summarized here.

block truncation coding (BTC) [CDF+86]: BTC replaces each block of 4×4 texels with two 16-bit colours and 16 bits to select which colour to use for each texel in the block.

S3TC [Inc98]: This replaces each 4×4 block with two 16-bit RGB colours and sixteen 2-bit indices which are used as selectors from the two colours and two other colours derived from them.

Pereberin [Per99]: This method first converts a 4×4 block into YUV colour space. Then it applies two iterations of Haar wavelet decomposition to the block channels independently. The lowest and middle level frequency coefficients are all kept; of the highest level frequency coefficients only the five largest in absolute value for Y are kept. Wavelet reconstruction is used to reconstruct the block. This has the advantage of holding three MIP-map levels in one texture.

Fenney [Fen03]: The essentials of this approach are a reduction of the original texture to two coarse-resolution full-precision textures A and B, and one full-resolution low-precision modulation texture. Reconstruction involves bilinearly sampling from A and B to obtain colours a and b, then using the value from the modulation texture to select a value used to linearly interpolate between them. A good algorithm for finding textures A and B that can produce quality images is still being researched.

Parameterized block approximation second also needs special hardware to extract a texel from the compressed block.

Here are examples that are hybrids of both approaches:

colour cell compression (CCC) [CDF+86]: CCC is the same as BTC except the two 16-bit colours are replaced with indices into a palette.

Bajaj *et al.* [**BIP00**]**:** This paper uses an algorithm that is very similar to Pereberin's for 3D. Each $4 \times 4 \times 4$ cell is decomposed using two iterations of Haar wavelet decomposition. The less significant wavelet coefficients are set to zero, and the remaining ones quantized to 8-bit indices into a codebook.

Fixed rate encoding schemes can get very good (2-4 bits-per-pixel) encoding rates for textures. However, in some cases a texture may not be able to be reasonably represented by a small VQ codebook. Also, when blocks are reconstructed solely from information in the block, artifacts can occur along the edges of blocks. Texture modification is difficult if a VQ codebook is used and if using Fenney's algorithm. Any indirection and reconstruction also adds to the latency of each access.

From a cache performance standpoint, ignoring any possible codebooks, using a fixed-rate encoding scheme increases the temporal and spatial locality of the memory accesses since the same chunk of memory now "covers" a larger area of the texture. However, if we take a large codebook into account, we need to deal with the possibly poor spatial locality of accesses to the codebook since we will be using the texture to direct the accesses to the codebook. We may also have to deal with a potentially troublesome additional load on memory resources to store the codebook itself.

3.1.2 Hierarchical Data Structures

There has been recent work in using hierarchical data structures such as the **quadtree** [LD91] and its three-dimensional counterpart the **octree** to provide a compact representation of sparse data and adaptively sampled data. By adaptively sampled we mean areas of low detail are represented with few texels or samples, and areas of high detail are represented with many samples.

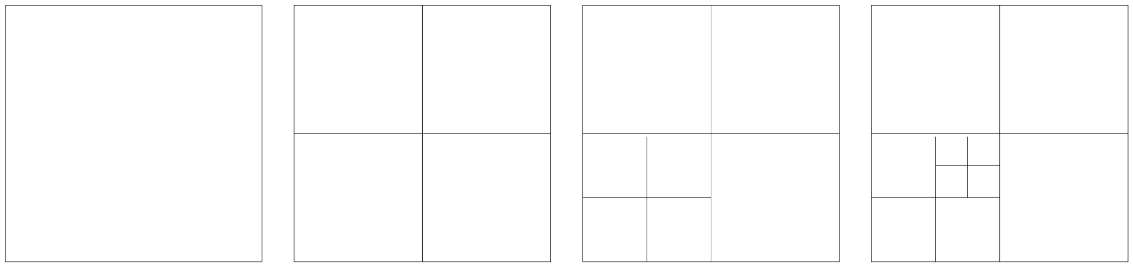


Figure 3.1: Quadtree partitioning of space.

A quadtree is based on subdividing a cell into four sub-cells, as shown in Figure 3.1. A node of the tree corresponds to a cell, and its ordered children to the corresponding subcells. One or more children may be absent (see Figure 3.2) for varying reasons depending on what the quadtree is storing.

Fernando *et al.* [FFBG01] use a quadtree to represent a shadow map efficiently. A shadow map is a depth map from the point of view of a light source used to

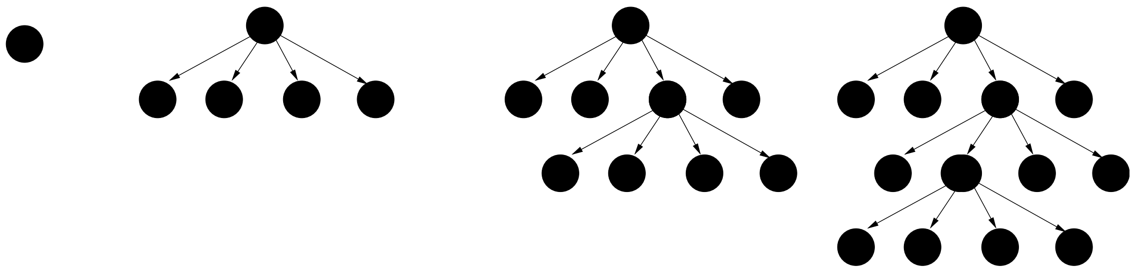


Figure 3.2: Quadtrees for space partitioning in Figure 3.1.

render shadows. Areas where there are depth discontinuities and hence need higher resolution get higher levels of subdivision of the cells.

Frisken *et al.* [FPRJ00] use an octree to store an adaptively sampled distance field. Each quadtree cell contains the sampled distance values of its four corners. Subdivision only occurs if the distance field within the cell is not well-approximated by bilinear interpolation of the cell's data.

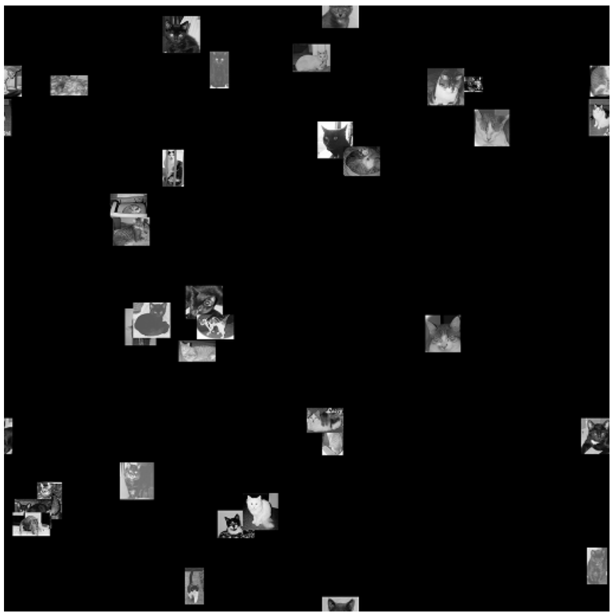
Benson *et al.* [BD02] use an octree to store a texture painted onto a three-dimensional surface. Only cells that intersect the surface are stored, and similarly to Frisken *et al.* subdivision only occurs if the texture is not well approximated by interpolation of the cell's data.

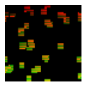
Unlike the fixed-rate encoding schemes, the quadtree/octree schemes do not waste space on empty areas or areas that are well-approximated by a few samples. The adaptive sampling also means that the quadtree/octree methods can control the level of approximation in different areas independently. It is also possible to modify the texture on the fly [FFBG01]. However, the tree structure is not directly suited for today's real-time hardware. The use of many levels of indirection results in high latency of access. The data structure is also not block oriented, which can result in memory fragmentation. Memory fragmentation is an issue for real-time systems as the methods needed to deal with it have relatively high memory and processor time needs. The above also lead to poor locality of memory accesses and wasted bandwidth consumption. Creation and editing of the data structure requires algorithms that are complex to implement in hardware.

3.1.3 Rearranging Storage

The methods discussed here are based on saving space by discarding unused parts of the texture and scaling down other parts ("adaptively sampling"), then cutting up what remains and packing the pieces into a smaller texture.

A texture atlas [CHM99, CH02] is a sparse representation of a three-dimensional (3D) solid texture. Consider an object shaded with a 3D solid texture. The surface of the object is "cut up" into pieces, say along the individual triangles. Each of the object's triangles is mapped onto a shaded triangle embedded in a 2D texture map. The triangles in the 2D texture may be packed tightly together to not waste space. Texture coordinates on the object index into the 2D texture, and the 2D


(a) original texture (512 × 512)


(b) index texture (64 × 64)

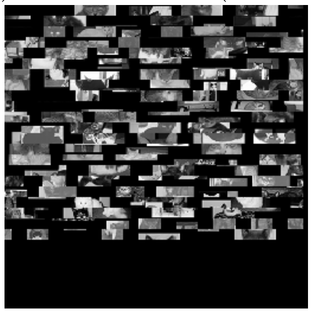

(c) data texture (256 × 256)

Figure 3.3: The corresponding index and data textures for the original texture for the uniform grid index method. We use $\alpha = 1$ for all the data blocks, which are of size 9×9 when padded. For the index texture, the R channel stores the x-coordinate, and the G channel stores the y-coordinate. Since α is fixed, we do not store scale factors in the index texture. We store the empty data block at coordinates $(0, 0)$, which means that the black parts of the index texture all point to the empty block.

texture is used to shade the object. The texture consisting of packed pieces makes it so that the existing MIP-mapping hardware does not produce proper results for MIP-mapping. The packing also results in some artifacts like seams. This method binds the texture to the object, making it unusable for any other object.

Kraus and Ertl [KE02] describe a method for adaptively representing texture maps on current off-the-shelf graphics hardware. It could be described as adding an “index” to a texture atlas so the texture is not tied to a particular object, can be adaptively sampled, and can discard unused parts of the texture. They use a two-level representation of the texture map. The first level is a coarse uniform grid covering the domain of the texture map, stored in a texture map (see Figure 3.3(b)). Each grid cell contains a reference to a “data block” on the second level and a scale factor, α. The “data block” is a scaled by α version of the part of the texture covered by the grid cell. To ensure continuous interpolation, the texels at a block’s boundary are replicated. Grid cells that do not cover any part of the texture’s “domain” (*e.g.* grid cells that cover only transparent areas of a colour texture) refer to a special “empty” data block whose texels are set to the default value (see Figure 3.4). The data blocks are packed compactly into one texture (Figure 3.3(c)). Sampling from the adaptive texture involves looking at the appropriate grid cell in the “index texture” and finding where in the “data texture” the corresponding data block is, then sampling from the data block.

This method compactly represents sparse textures through the use of the “empty” data block, and the scaling down of some blocks can be viewed as a way of “adaptively sampling” the original texture.

As with the texture atlas, the packing of parts of the original texture into one texture can result in seam artifacts. Existing MIP-mapping hardware does not work properly with this method. Texture modification is a problem, and extra

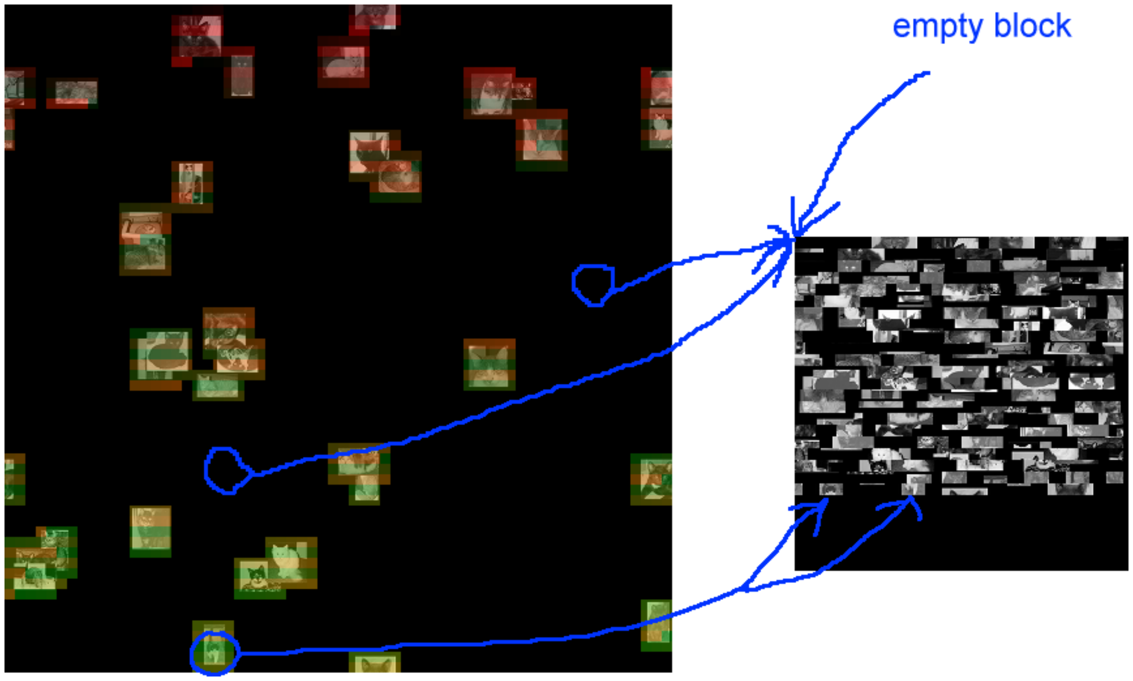


Figure 3.4: On the left we have the index and original textures from Figure 3.3 combined so that the magnified index texture is transparently laid on top of the original texture. On the right we have the data texture. The area in each circle is joined to the corresponding data blocks in the data texture by the arrows.

texels need to be stored for each data block to prevent (some) seam artifacts.

The use of an index, while allowing the storage of only useful parts of a texture and the adaptive sampling, presents its own set of difficulties. The index lookup adds to the latency of access. There is a tradeoff between the index texture resolution and the amount of memory the data blocks occupy; as the index texture resolution goes up, the minimum resolution of a data block needed to provide the same or better level of detail goes down, and as the index texture resolution goes down, the minimum resolution of a data block needed to provide the same or better level of detail goes up. Also for very large textures, say $2^{16} \times 2^{16}$, that require a high level of detail, it may be impossible to have a small index texture without resorting to a multi-level index.

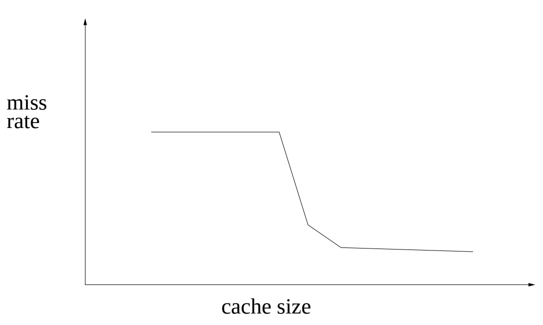
3.2 Rasterization Order, Texture Representation, and Cache Design

Before we discuss issues in cache design for texture mapping we first define the following term:

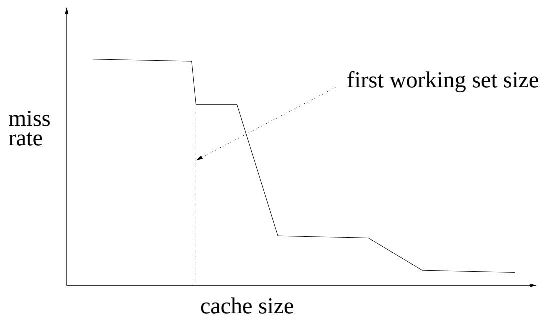
working set: The data that is actively in use at a given time [Den68].

A working set size can be detected in a graph of miss rate versus cache size or the miss rate curve by observing where "steps" (see Figure 3.5(a)) occur in the graph. Usually there is a hierarchy of working sets (see Figure 3.5(b)), each step down corresponding to some architectural factor, *e.g.* the cache becomes large enough to contain the texture required for a scanline, for a triangle, or the entire screen.

One possible texture representation in memory is to have the texels occupy a contiguous $2^{n_1} \times 2^{n_2}$ array, where $n_1, n_2 \in N$, in row-major or raster order. The consecutive memory blocks they occupy cover the texture as in Figure 3.6. It is



(a) A miss rate curve showing a step down indicating the presence of a distinct working set.



(b) A miss rate curve showing several steps down indicating the presence of a hierarchy of distinct working sets.

Figure 3.5: Miss rate curves showing working set sizes.

also conventional to rasterize one triangle at a time in raster order. Previous work on the characterization of direct texture mapping indicates that when using the rasterization method and texture representation described, the first working set is small compared to the amount of texture data used to render a scene [HG97]. Work on using a secondary cache to utilize frame-to-frame coherence indicates that the inter-frame working set size is on the order of several megabytes [CBS98]. Since a primary cache of several megabytes is unreasonable, we restrict our interest to working sets within a single frame. From now on, the term "working set" will refer to the first working set, unless otherwise noted.

Consider the following maps describing a direct texture lookup:

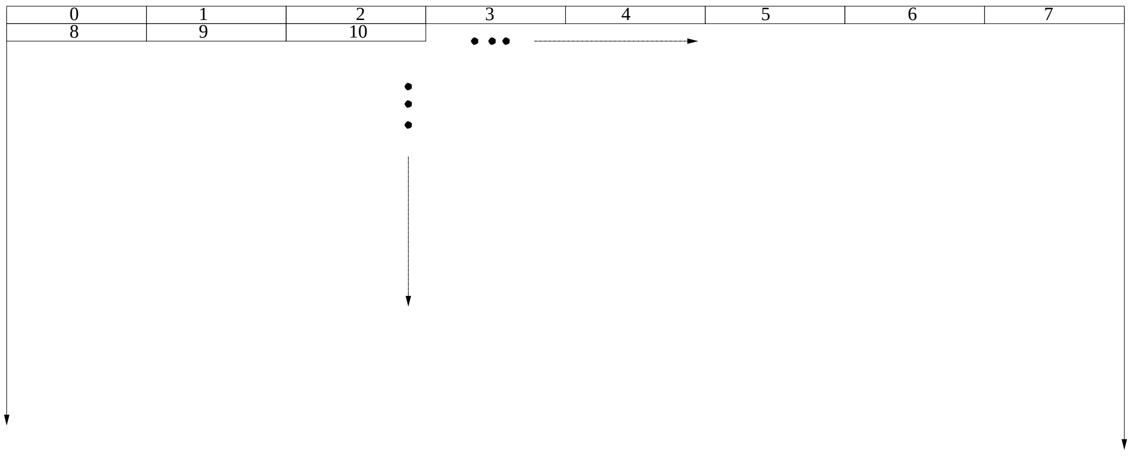
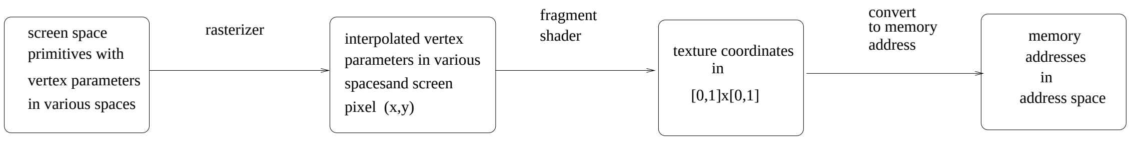


Figure 3.6: How a texture might be covered by memory blocks. Each small box indicates the texels covered by a memory block, and is 1 texel high. The boxes are arranged in texel raster order.


screen space primitives with vertex parameters in various spaces
rasterizer
interpolated vertex parameters in various spacesand screen pixel (x,y)
fragment shader
texture coordinates in [0,1]x[0,1]
convert to memory address
memory addresses in address space

As we can see from the maps, the rasterizer "generates" the initial locality that results in spatial and temporal locality in memory. Of the maps shown, we only have control over two: the texture coordinate to memory map, *i.e.* base texture representation and the rasterizer.

3.2.1 Base Texture Representation

The simple texture representation shown in Figure 3.6 has a miss rate curve for a fully associative cache that is sensitive to orientation when raster order rasterization is used [HG97]. It is easy to see why this is so as for raster order rasterization the working set of data is roughly the texture data requested to texture a scanline [HG97]. The number of memory blocks required for texturing a scanline is small if the scanline runs horizontally through the texture, and large if it runs vertically through it. Also, since cache sizes, texture dimensions, and block dimensions are powers of two, the simple representation maps texels in the same column to the same cache index, resulting in many conflict misses for a non-fully associative cache if we are unlucky enough to have the transformed rasterization curve travel vertically down the texture. One way to ameliorate these orientation related problems is to change the base texture representation.

A blocked or tiled representation is shown in Figure 3.7. Texels within each square-shaped region of the texture are stored in a contiguous array of memory. The consecutive blocks are arranged in raster order in the texture. This reduces the effect the orientation of the texture has on the working set size compared to the simple representation by reducing the variance in the number of blocks required to render, or “cover”, a scanline.

With a blocked representation comes the question of the relationship between block size and cache line size. Hakura *et al.* [HG97] determined that the best block size is one that matches the cache line size. They found that increasing the line size without blocking and increasing the block size correspondingly degrades cache

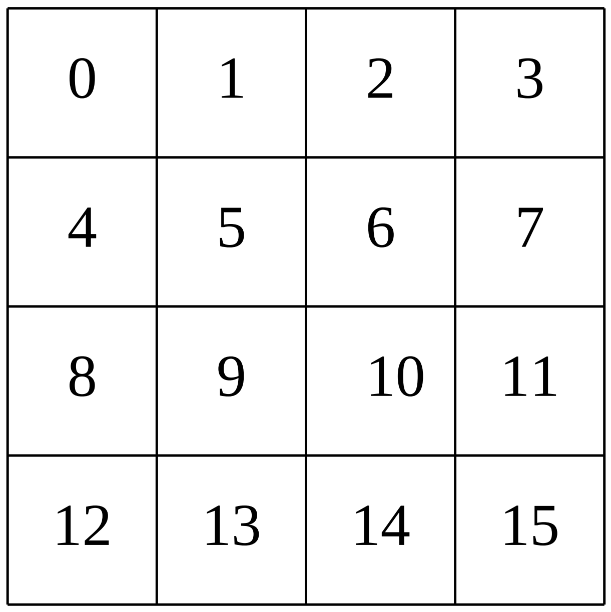


Figure 3.7: A texture using a blocked representation. Each box indicates the texels covered by a memory block. The boxes are arranged in raster order.

performance.

A blocked representation (with larger block and cache line size) reduces the number of capacity misses for cache sizes smaller than the working set size by better translating the initial spatial locality into spatial locality within a memory block. It also reduces the number of conflict misses by trading conflicts between neighbouring texels in the conventional map for conflicts between neighbouring texels in neighbouring blocks which occur far less frequently [HG97].

3.2.2 Rasterization Order

Using a blocked texture representation, Hakura *et al.* experimented with tiled rasterization in which the tiles are arranged in raster order and raster order is used to order the pixels within the tiles (see Figure 2.7). They show that the texel access pattern converges to that for conventional raster order rasterization for tiles that are too small or too big, so “medium” sized tiles are the best. They also show that for a texture block size of 8×8 texels, a tile size of 8×8 pixels is best. Tiled rasterization has little effect on scenes with small or moderately sized triangles. However, for scenes with large triangles it reduces the working set size by increasing the spatial locality. It also reduces the number of conflict misses, though not enough so that a 2 or 4-way set associative cache would have a miss rate curve near the miss rate curve for a fully associative cache. They explain that this is due to the working set being restricted to a more spatially contiguous region of the texture and the working set size being smaller so fewer blocks in it can possibly conflict.

McCormack *et al.* present metatiling [MM00], in which there are multiple levels of tiling with tiles “nested” within supertiles as in Figure 3.8. They suggest using multiple tiling levels with the tile size at each level selected so that the resulting working set sizes correspond to the sizes of the various levels of memory in use, such as the caches in a multi-level cache system and the frame buffer.

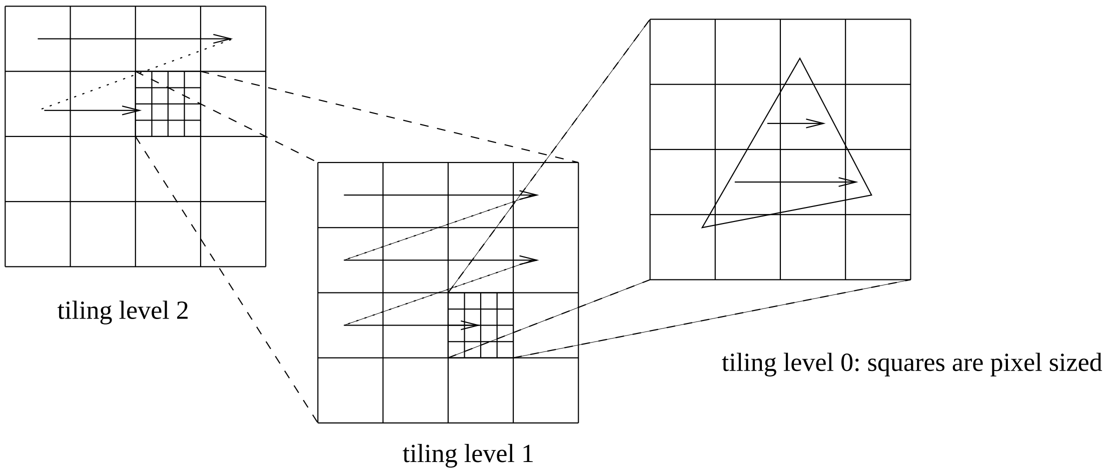


Figure 3.8: Rasterization using metatiling. Rasterization at all tiling levels for this example is in raster order.

McCool *et al.* suggest using Hilbert order rasterization to improve spatial locality at all scales [MWM01]. Such a scheme, or one based on the Z-curve, is equivalent to metatiling at all scales. This is a simple way to to ensure that each memory level has a working set size corresponding to it. The high spatial locality is also expected to be robust under a programmable shader.

3.2.3 Conflict Misses

For a single MIP-map texture pyramid using a blocked representation, most conflict misses occur for two reasons [HG97]:

1. There is a conflict between blocks in neighbouring MIP-map levels due to

trilinear interpolation.

2. There is a conflict between neighbouring blocks in the same column on the same MIP-map level.

A two-way set associative cache takes care of most of the conflicts due to MIP-map levels [HG97]. Igehy *et al.* use two direct mapped caches, one each for odd and even MIP-map levels, to deal with conflict misses between MIP-map levels [IEP98].

The second case is more difficult to deal with as the path through the texture is highly dependent on the rasterization method and the scene. One method that greatly reduces this type of conflict miss is to use tiled rasterization [HG97]. Most other research on this is based on rearranging the blocks in a blocked representation, *i.e.*, changing the texel addressing scheme.

Here is a list of different addressing schemes. Note that this does not affect capacity misses as it merely permutes the order of the blocks of texels in memory.

4D: What we have heretofore described as a blocked or tiled representation, this scheme is prone to conflicts between neighbouring blocks in the same column [HG97].

4D padded: Shown in Figure 3.9, this scheme avoids conflicts between blocks in the same column by modularly shifting each row of blocks a fixed distance

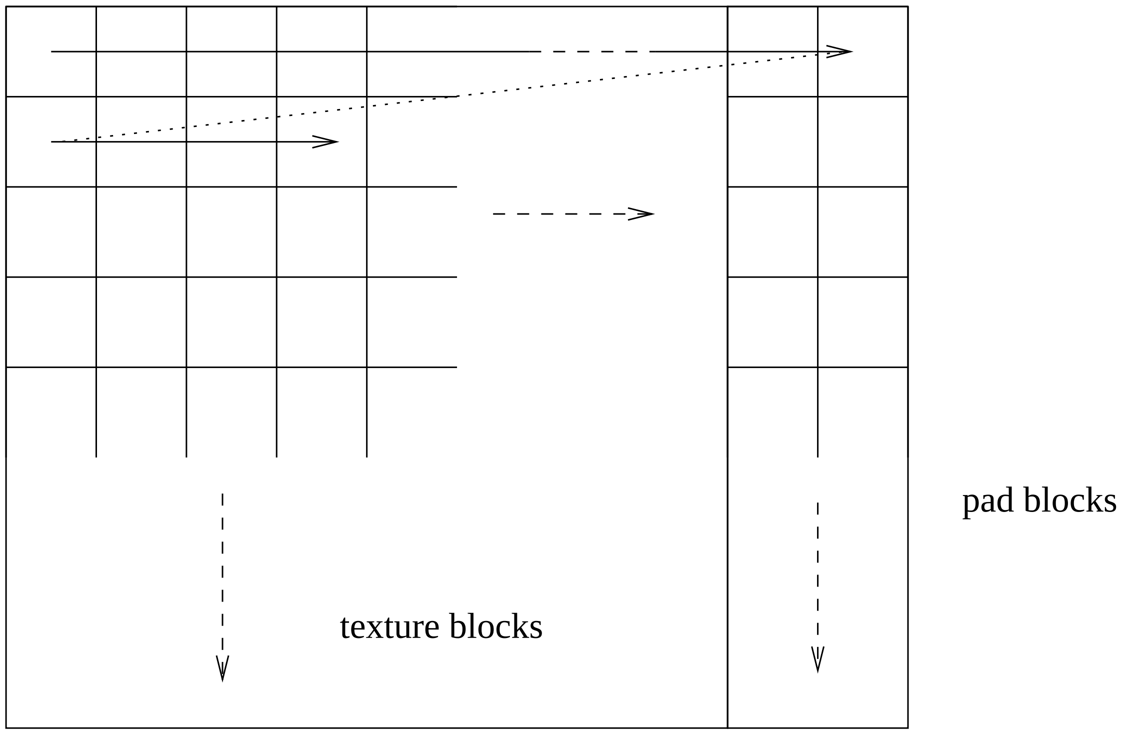


Figure 3.9: 4D padded representation, with $n = 2$. Based on a diagram from [HG97].

from the previous row in address space by padding the end of each row with n dummy blocks. Hakura *et al.* show that for their scenes and $n = 4$, it is an an improvement upon the 4D representation for a 2-way set associative cache [HG97]. It has the disadvantage of wasting memory and adding latency in texel addressing.

6D raster: This scheme has two levels of tiling (see Figure 3.10(a)). The tile size in the upper level is chosen to match the size of the cache. Both levels of tiles

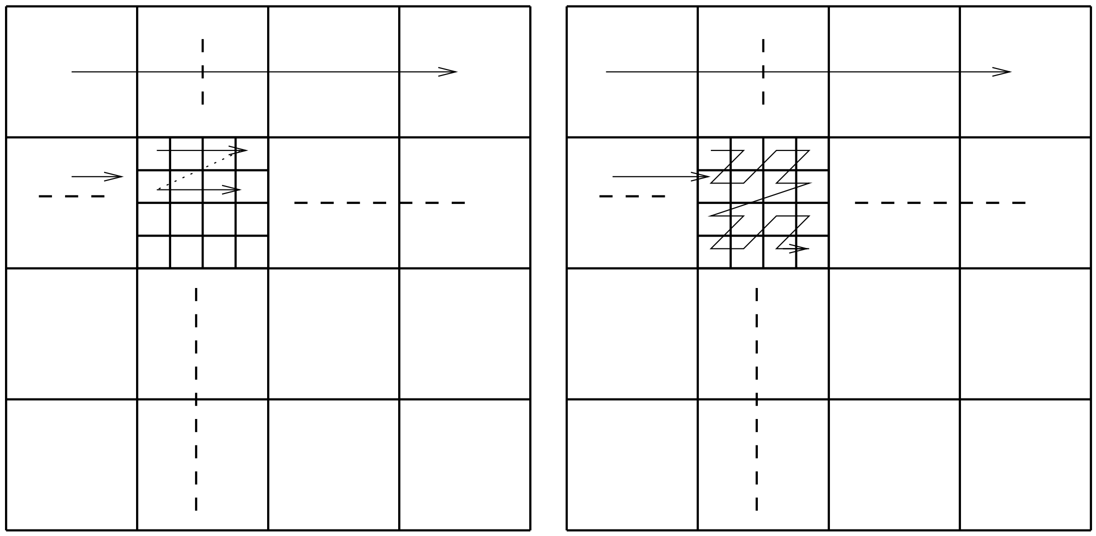

(a) raster (b) z-order

Figure 3.10: 6D blocked tiling. Based on a diagram from [HG97].

are arranged in raster order. Using it results in a reduction in conflict misses compared to using 4D-padded so that the miss rate is close to that of a fully associative cache [HG97].

6D Z-order: This scheme is the same as 6D raster (see Figure 3.10(b)) except the tiles are arranged in Z-order instead of raster order within each supertile. This arrangement makes it so that any four adjacent blocks will have different cache indices [IEP98].

These various texel addressing methods vary in the amount of time required to compute them.

3.2.4 Parallel Access to Neighbouring Texels

With MIP-mapping and trilinear interpolation enabled, texels are accessed in 2×2 blocks from adjacent levels, so it is advantageous to be able to retrieve them in parallel. One way to do this is to interleave the texels across multiple cache banks [Mol95, CBS98]. Distributing neighbouring texels across different cache banks involves arranging the texels in Z-order within each block. Similarly to arranging blocks in Z-order in the texture, this ensures that neighbouring texels do not map to the same bank and conflict with each other.

3.2.5 Reducing Latency

Igehy *et al.* use prefetching in a specialized texture mapping cache architecture to give performance near that of a zero-latency memory system, even with high memory latency [IEP98]. This performance comes with the provision that there be no dependent texturing involved.

Chapter 4

A B-tree Implementation of Sparse Textures

Of the previous work in hardware texture compression the one best suited for general sparse textures is the method devised by Kraus and Ertl [KE02]. However, their method has some disadvantages when applied to real-time shading that led us to explore another implementation and specialized caching hardware to support it.

4.1 Our Implementation

Similarly to the grid method, our implementation is based on adding an index so only the necessary parts of the texture need to be stored. We choose to index our data using a B-tree. Consequently we must also tune the B-tree parameters for good performance.

4.1.1 The B-tree

Our implementation is designed with the assumption that memory is divided up into blocks of a fixed size that are cached, and that loading a block into the cache is "expensive" while accessing a block that is already in the cache is "cheap". This models graphics hardware at the present time and for the immediate future. Under these restrictions, we will wish to minimize the number of blocks accessed when accessing a texel. Also, in order to simplify memory allocation and deallocation and avoid memory fragmentation issues, we want an implementation with data structures whose requirements for a contiguous chunk of physical memory is restricted to the size of a memory block. Assuming a one-dimensional key space, a data structure well suited for indexing under the given constraints is the B-tree [LD91] and its variants.

Before we define a B-tree we need the definition of an (a, b)-tree. Let $a \geq 2$,

$b \geq 2a - 1$. An (a, b)-tree is one of the following:

1. The empty tree.
2. A leaf.
3. A tree for which
   - All non-leaf and non-root nodes have c children, where $a \leq c \leq b$.
   - The root has c children where $2 \leq c \leq b$.
   - All leaves are at the same depth.

We now define a B-tree of order b as an (a, b)-tree where $b = 2a - 1$ and the following search property for internal nodes holds:

> If key value K is stored in an interior node between pointers to subtrees T and T', then every key in T is less than or equal to K and every key in T' is greater than K. [LD91]

In order to make b as large as possible and thus the tree height as low as possible, we only store keys and their associated data at the leaves of the tree. For our purposes, each leaf consists of a memory block containing a set of $(\text{key}, \text{data})$ pairs. The internal nodes only store index values to direct the searches and pointers to children nodes. Also, we choose b to be as large as possible while still fitting

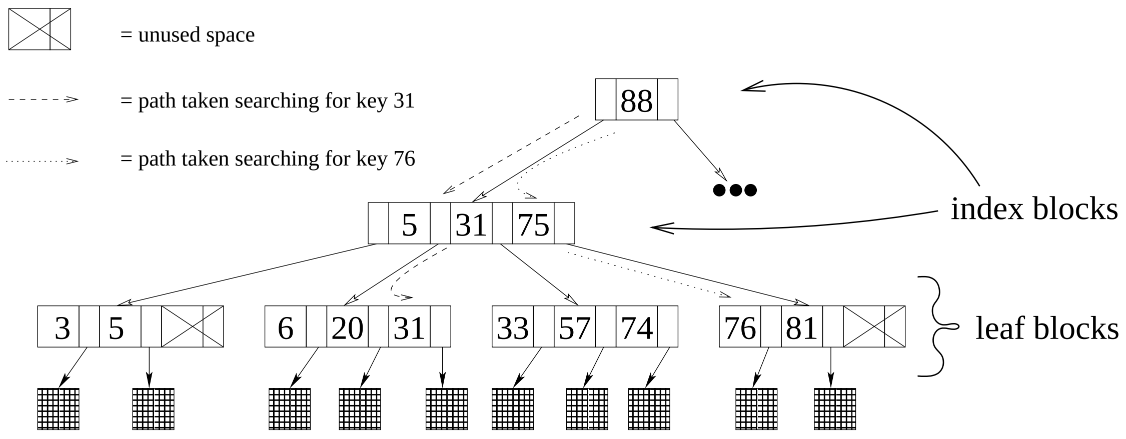
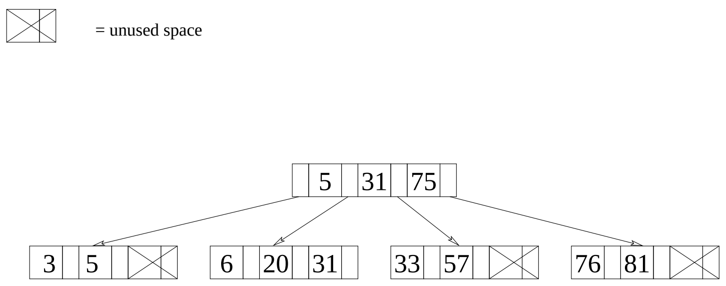

= unused space
= path taken searching for key 31
= path taken searching for key 76
88
5
31
75
3
5
6
20
31
33
57
74
76
81
index blocks
leaf blocks

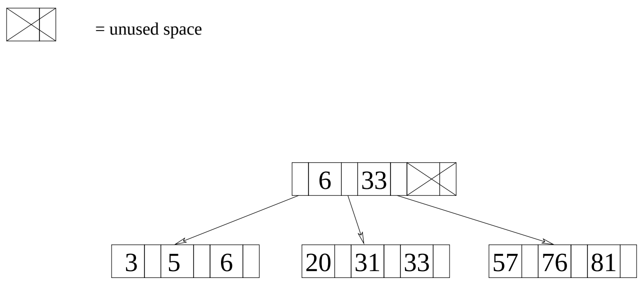
Figure 4.1: This is an example of a B-tree for $b = 3$. The data stored are (key, p) pairs, where p is a pointer to one of the blocks with a grid of data in it. Only the subtree to the left of the index 88 in the root node is shown.

each node into a memory block. This minimizes tree height while producing a one-to-one mapping from nodes to memory blocks. Figure 4.1 shows an example of a B-tree and how to search it. Searching a node to find the appropriate child pointer to follow can be implemented in constant time in hardware since the block-size is fixed.

The B-tree is obviously designed for block-oriented transfers and a memory allocation/deallocation scheme based on fixed-size memory blocks. It also supports efficient insertion of data [LD91]. The B-tree is also self-balancing, with the height of the B-tree being of order $\mathcal{O}(\log n)$. Unfortunately the B-tree has the disadvantage of permitting the nodes to be only half full resulting in wasted memory and a higher tree height when random insertion is used. The remedy is **compaction**.



(a) The original tree.



(b) The compacted tree.

Figure 4.2: Illustration of B-tree compaction.

Compacting a B-tree (see Figure 4.2) involves rearranging the distribution of data in the leaves and the indices in the internal nodes to make the nodes as full as possible. Since it possibly involves all nodes in the B-tree, it is not as efficient an operation as insertion or deletion which involves $\mathcal{O}(h)$ nodes, where h is the height of the B-tree. However, if b is large, the number of nodes in the B-tree is small.

4.1.2 Basic Description

For now assume we are dealing with a 2-D texture with default value d. We divide the texture up into $n \times n$ blocks, where n is chosen to be as large as possible and have the block of texels still fit into a memory block *i.e.* we choose a blocked representation. Only blocks that have a texel with a non-d value, *i.e.* **occupied** blocks, are kept; unoccupied or **void** blocks are "discarded". Each occupied block has a key k associated with it that can be computed from the (x, y)-coordinates of the origin of each block; we call the method of computing k the **index scheme**. The $(k, \text{pointer to block})$ pairs are stored in a B-tree. It is easy to see how this method can be extended to higher-dimensional textures.

As a confusion avoiding measure, henceforth we refer to B-tree index blocks and leaf blocks as **index blocks**, occupied blocks as **texture blocks**, and occupied and void blocks as **(texture) data blocks**.

Looking up a texel at position (x, y) involves the following:

1. Compute k from $(\lfloor \frac{x}{n} \rfloor, \lfloor \frac{y}{n} \rfloor)$.

2. Search the B-tree for key k. If k is not in the B-tree,

    a) Return $p = 0$. We could additionally return at constant cost a key-range $[k_1, k_2]$ where $k_1 <= k <= k_2$ and k_i corresponds to a void-block

$\forall k_i \in [k_1, k_2]$. k_1 and k_2 are obtained from the keys in the B-tree that bracket k in the ordering.

Else

b) Return the appropriate pointer p.

3. Return the texel. If $p = 0$,

a) Return d.

Else

b) Retrieve the block b pointed to by p

c) Compute the offset i into the block from $(x - \lfloor \frac{x}{n} \rfloor n, y - \lfloor \frac{y}{n} \rfloor n)$.

d) Return the texel $b[i]$.

Note that unlike the grid method, there is no scale factor stored to implement a form of "adaptive sampling". We also do not pad the texture blocks.

4.1.3 B-tree Parameters

Naturally it is best for the texel searches to be as efficient as possible. In order to accomplish this we need to tune the B-tree parameters in conjunction with other external parameters to optimize performance. These B-tree parameters are the block size and the index scheme.

Block size

The block size affects the height and size of the B-tree, the amount of memory needed to store the texture data, and the time taken to load a memory block into the cache. These effects in turn affect the latency of texel access, the texture compression ratio, the miss rate, and the bandwidth consumption.

As the block size increases, tree height decreases so the number of index blocks accessed on a texel lookup also decreases. However, the time taken to load a memory block into the cache increases. Also, more memory is required to store the texture data. For a fixed cache size, the number of cold misses decreases. The opposite occurs when the block size decreases.

Index scheme

The use of a B-tree requires an index scheme that imposes a strict one-dimensional order on each texture block based on the location of the block. The curves mentioned in the section on rasterization order are appropriate for 2-D textures, as are their higher dimensional analogues for higher dimensional textures. In fact, a sufficiently high resolution quantized approximation of any space-filling curve

$f : [0,1]^n \to [0,1]$ is also appropriate [Ma02, MM02]. The block arrangements described in Section 3.2.3 also qualify for 2-D textures.

We choose to test using raster order, Z-order, and Hilbert-order curves. The main concerns with the choice of index scheme are computational expense and locality preservation.

The computation of the key k for a texel at (x, y) adds to the access time so we want an index scheme that is efficient. Z-order is particularly efficient as it can be implemented in hardware to be zero cost (using bit interleaving), while Hilbert is more time consuming to compute as the bits of k need to be generated sequentially. Computational cost must be balanced with the need for a spatially coherent curve.

Consider a small set of consecutive texel requests and their corresponding keys. If the interval containing the keys is small, it restricts the number of paths down the B-tree involved in servicing the texel requests which translates into better temporal locality of the accesses to the index blocks of the B-tree. While we expect an index scheme with good spatial coherence to be apt for the task, the rasterization order and the orientation of the texture also need to be taken into account, as they interact with the index scheme to affect the width of the interval (see Figure 4.3).

Regardless of the index scheme chosen for ordering the texture blocks, in order to ensure parallel access to neighbouring texels as described in Section 3.2.4, the texels in each block need to be arranged in Z-order.

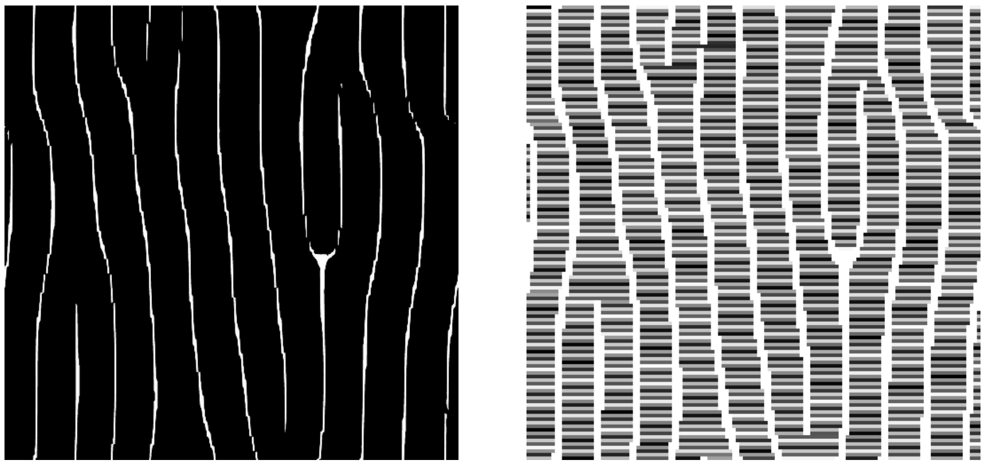

(a) The texture. (b) Texture showing block occupation.

Figure 4.3: In (b) the white areas are occupied blocks and the gray areas are the intervals between the keys in the B-tree. The index scheme used is raster order. Note that if we travel vertically down the texture a much larger interval is needed to contain a set of texel requests compared to the case where we travel horizontally across the texture.

4.2 Motivation

Both disadvantages to the uniform grid index method and advantages over the standard texture storage method motivate our implementation.

4.2.1 Uniform Grid Index Method

The use of a uniform grid to index the data blocks of a texture as described by Kraus and Ertl [KE02], possibly using a multi-level index, is similar to the use of page tables [Tan92] to index the occupied parts of a range of virtual memory. While

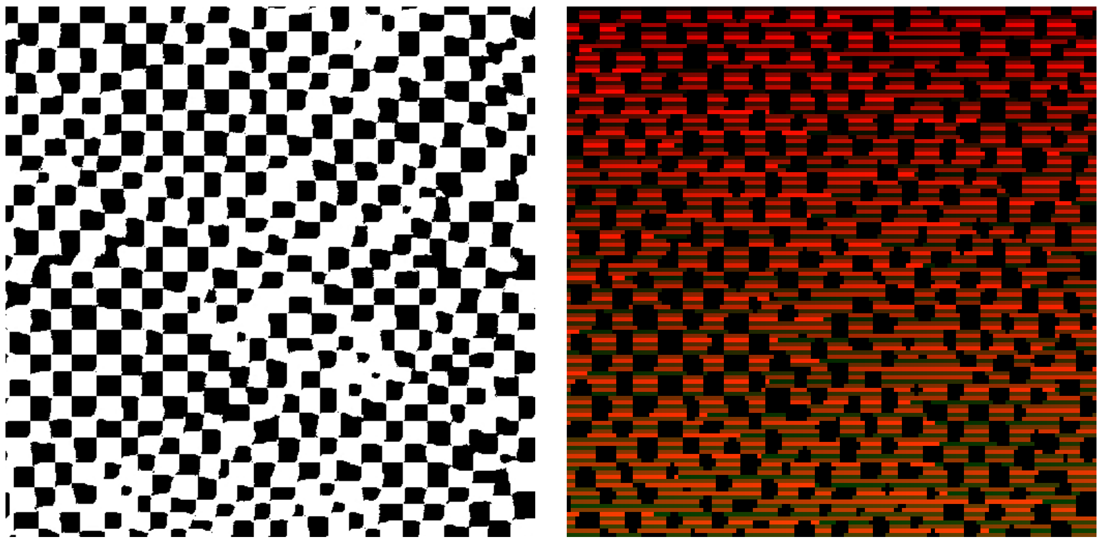


Figure 4.4: A texture and corresponding magnified index texture using the adaptive texture scheme described by Kraus and Ertl [KE02]. Blank parts of the index refer to the "empty" data block. Note that multi-level indexing for a reasonable block size will not reduce the space occupied by the index.

page tables work for virtual memory where the memory occupation pattern has a small number of contiguous segments, the occupation pattern for textures may not follow an analogous pattern. In fact, it is easy to create a sparse texture for which the use of more than one level of indexing would not save memory over using just one level of indexing (see Figure 4.4). The result is that the space taken up by the index does not necessarily scale with the texture occupancy.

Another concern is that the index texture can conflict with the data texture in the cache. The graphs in Figures 4.6, 4.7, 4.8, and 4.9 show that an extra level of associativity or a separate cache is needed to deal with the conflict misses induced by the index if the cache is small or the memory block size is large for densely occupied portions of a sparse texture. This necessity obviously extends to an extra level of associativity or separate cache being needed for each level of indexing.

Comparing the bandwidth consumption of the uniform grid index method to that of the standard dense texture storage method when loading blocks with nothing interesting in them consumes no bandwidth (Figures 4.10, 4.11, 4.12, 4.13), we see that the bandwidth consumption overhead involved in the uniform grid index method is high. Most of the overhead can be attributed to the texel padding needed to minimize seam artifacts, and the wasted bandwidth from the "edge effect" of the memory blocks not perfectly covering the area in use at a time.

The requisite dependent texturing in the uniform grid index method means high latency of access that cannot be covered by a prefetching architecture such as that proposed by Igehy *et al.* [IEP98].

These areas for improvement motivate our implementation of sparse textures and the supporting hardware.

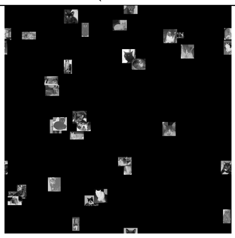
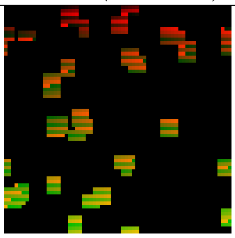
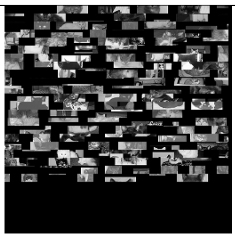
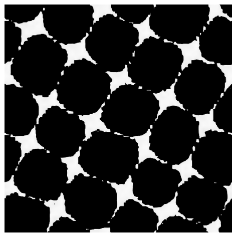
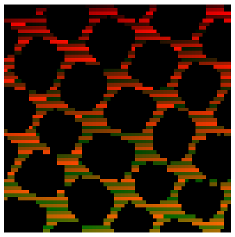
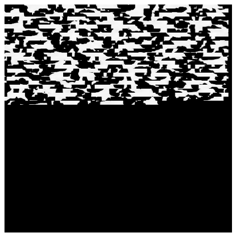
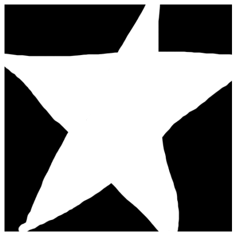
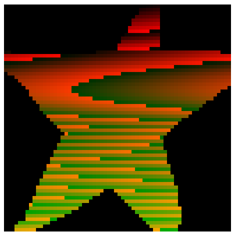
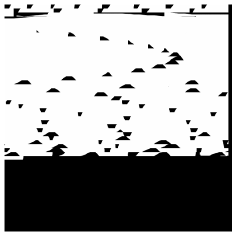
name	original(512×512)	index (64×64)	data	data resolution
cat				256×256
cellorig				512×512
star				512×512

Figure 4.5: The 512×512 texture is cut up into 8×8 tiles (resulting in a 64×64 index texture) that are padded to 9×9 and are processed in raster order and stored in the data texture in raster order. The scale factor for all the data blocks is $\alpha = 1$.

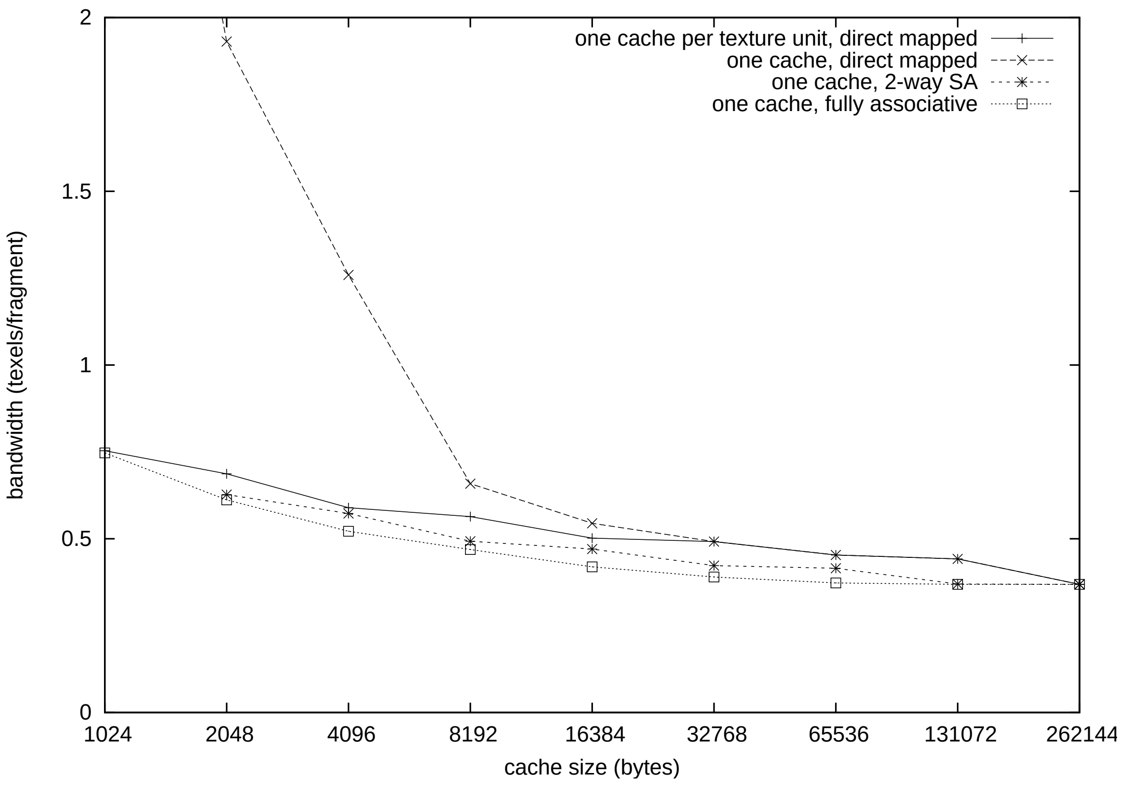


Figure 4.6: The graph of bandwidth consumption versus cache size using the adaptive texture map method of Kraus and Ertl [KE02] for the *cat* texture in Figure 4.5 for a memory block size of 64 bytes or 4×4 texels. All textures are represented in memory using the blocked representation, with the blocks arranged in Z-order in the texture. The rasterization order is Hilbert order; the combination of this rasterization order and the arrangement of the memory blocks in the texture was chosen to be shown because it gives the best performance. Other parameters: the scene is a single triangle zoomed to full screen (256×256) and rotated so that the texture is at a 90 degree angle, the sampling method is bilinear interpolation, the eviction policy is LRU, and the cache line size is matched to the memory block size.

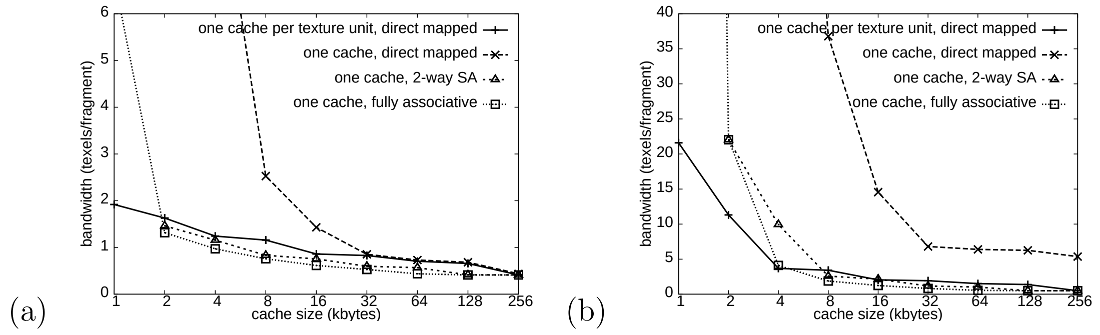


Figure 4.7: This is the same as in Figure 4.6 except the block size is (a) 256 bytes (8 × 8) and (b) 1024 bytes (16 × 16).

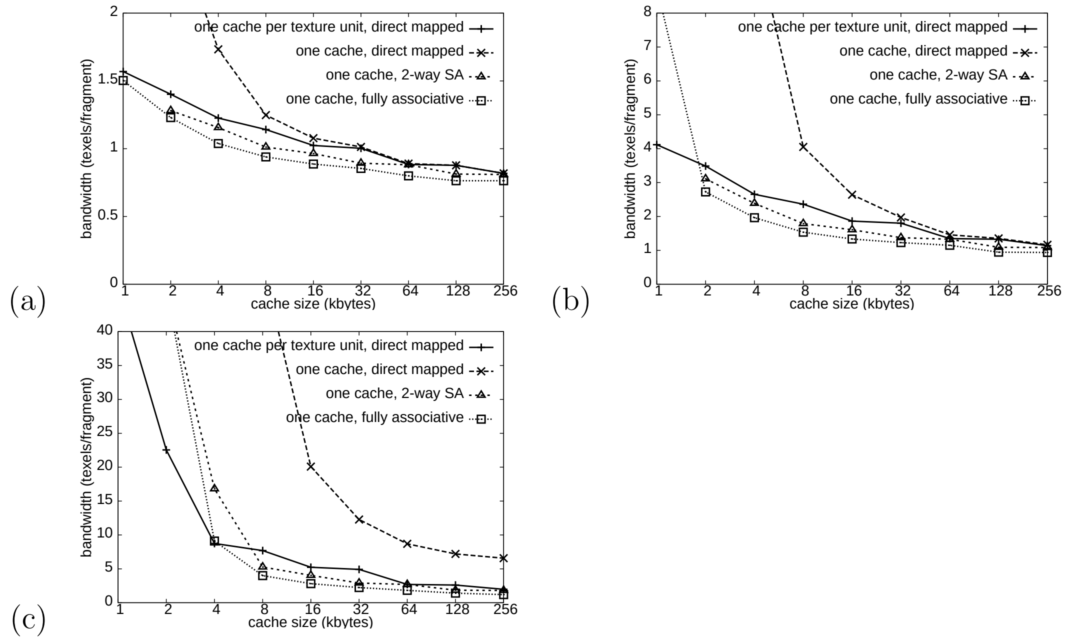


Figure 4.8: This is the same as in Figures 4.6 and 4.7 except the texture is *cellorig*. The block size for each graph is (a) 64 bytes (4 × 4) (b) 256 bytes (8 × 8) and (c) 1024 bytes (16 × 16).

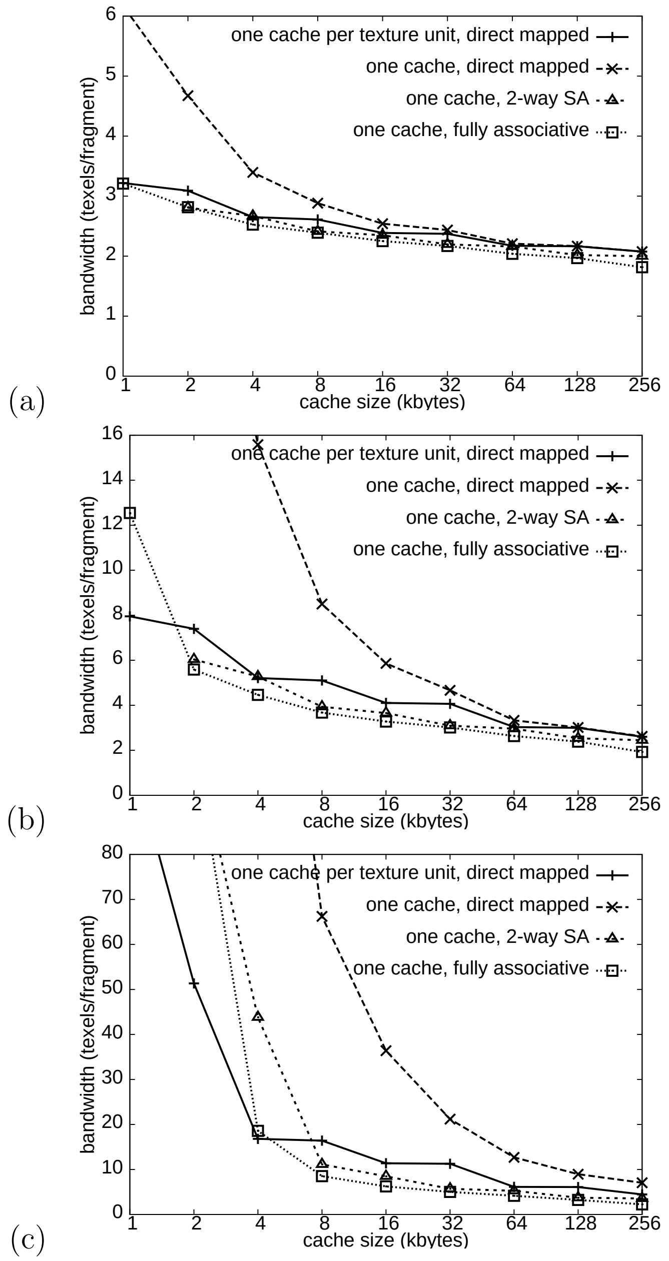
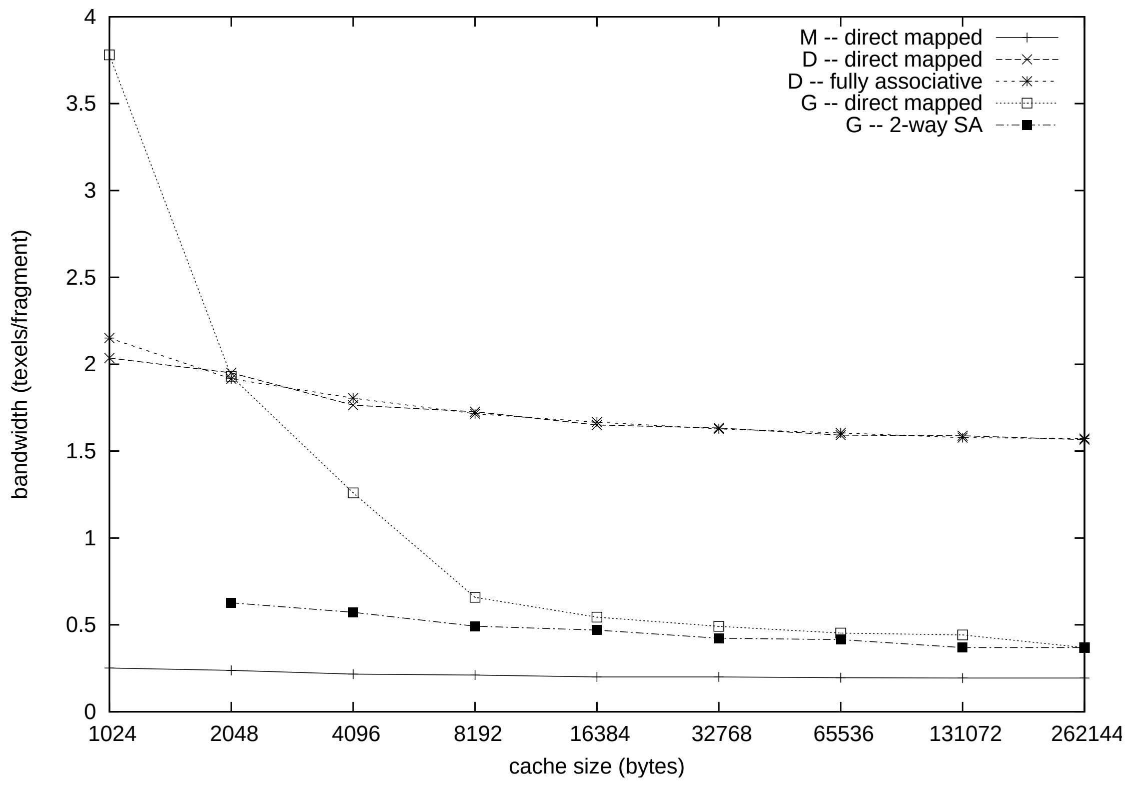
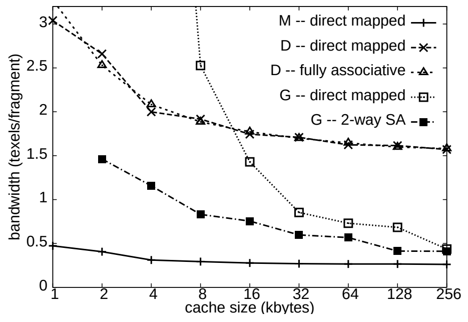


Figure 4.9: This is the same as in Figure 4.8 except the texture is *star*. The block size for each graph is (a) 64 bytes (4×4) (b) 256 bytes (8×8) and (c) 1024 bytes (16×16).



Label	Description
D	Conventional sampling from the original *cat* texture in Figure 4.5. The sampling method is bilinear interpolation with no MIP-mapping to match the Kraus and Ertl method. The other parameters used to generate the graph line are as in Figure 4.6.
M	Same as D except that blocks that only store the default value do not have the bandwidth they contribute counted.
G	Uses the uniform grid indexing method of Kraus and Ertl using the index and data textures for the *cat* texture in Figure 4.5.

Figure 4.10: This graph shows the bandwidth used by the various methods of storing the *cat* texture from Figure 4.5 for different types of caches. The descriptions of the various methods used are in the table. The memory block size is 64 bytes (4×4 texels).

(a) 

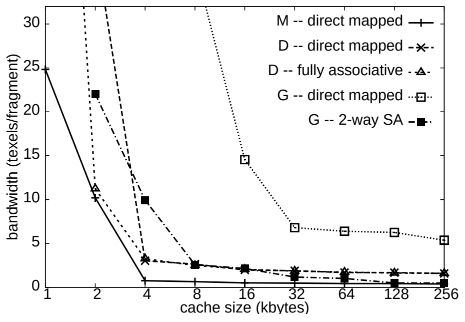
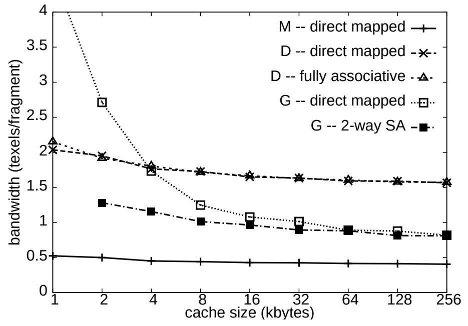
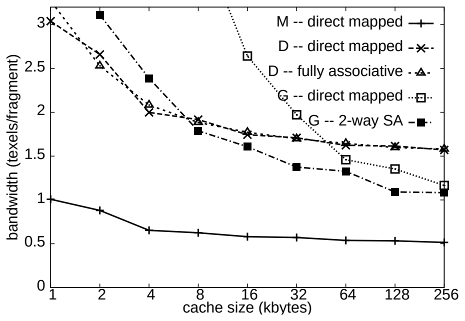
(b) 

Figure 4.11: This is the same as in Figure 4.10 except the block size is (a) 256 bytes (8 × 8) and (b) 1024 bytes (16 × 16).

(a) 

(b) 

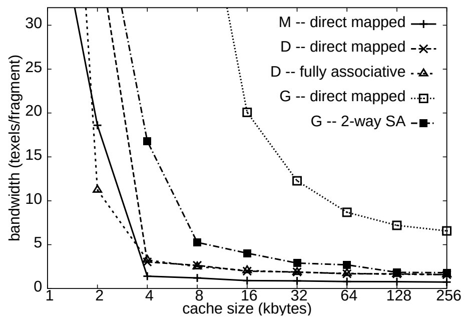
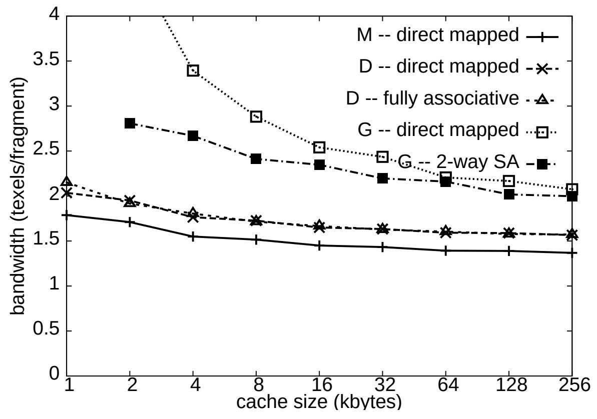
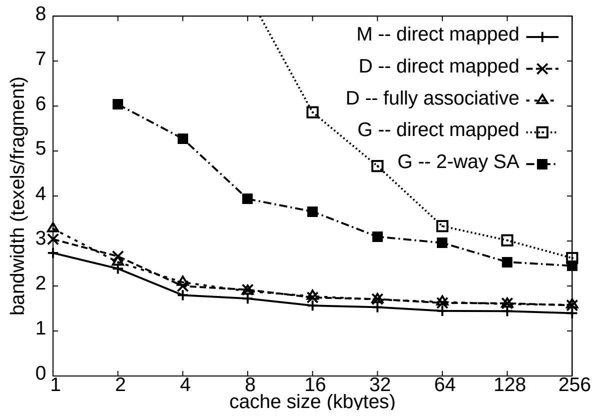
(c) 

Figure 4.12: This is the same as in figures 4.10 and 4.11 except the texture is *cellorig*. The block size for each graph is (a) 64 bytes (4 × 4) (b) 256 bytes (8 × 8) and (c) 1024 bytes (16 × 16).

(a)


(b)


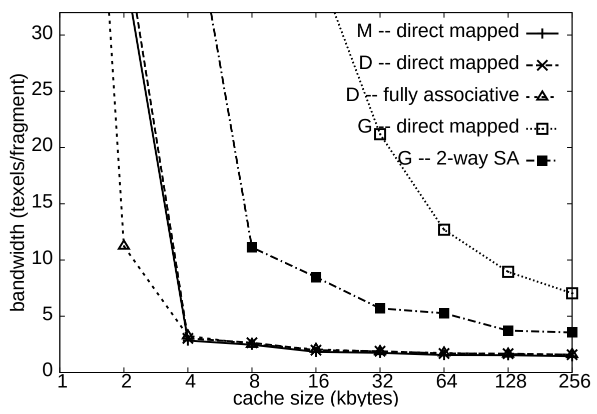
(c)


Figure 4.13: This is the same as in Figure 4.12 except the texture is *star*. The block size for each graph is (a) 64 bytes (4×4) (b) 256 bytes (8×8) and (c) 1024 bytes (16×16).

4.2.2 Advantages of the B-tree Index Method

Besides advantages over the uniform grid index method, there are more general benefits to the B-tree index method.

One advantage over the uniform grid index method is the B-tree index method does not require texture blocks to be padded. Seam artifacts are also not an issue. Another advantage of the B-tree method is the size of the B-tree is based on the number of $(key, pointer)$ pairs stored in it, not the texture size, so the index size is directly related to the amount of memory needed to store the texture data of the sparse texture, which is important for large high dimensional textures. Our method is also intended to work with existing MIP-mapping hardware. Finally, since the B-tree data structure supports on-the-fly editing if the memory allocation is done in fixed-size blocks, we get as a result on-the-fly texture editing.

An important general benefit arises from the B-tree being a block-oriented data structure with fixed-size blocks. Since all the blocks, both index and texture blocks, are the same size, memory allocation and deallocation can be done in such a way that there is no memory fragmentation. Memory fragmentation is a problem because most memory systems needed to deal with it are not compatible with a real-time system, and may have relatively high space and processor time costs. No memory fragmentation means that the memory systems necessary to deal with memory fragmentation are not needed, nor the associated space and processor time.

4.3 Cache Support

There are many cache design techniques used to improve performance by reducing the miss rate, the miss penalty, and the hit time. Many are mentioned in Section 2.2.4. Some of these techniques are complicated to implement. The relative impact of the various techniques is also hard to quantify without actual hardware implementations. Most do not directly address the problems posed by our method of storing and accessing the texture, but work in a more general fashion that would still work for our method. As a result, we focus our cache design efforts on coarse solutions that deal directly with the problems introduced by the B-tree indexing.

We focus on minimizing the extra latency and bandwidth consumption induced by the indexing. We also try to translate not storing void parts of the texture into reductions in bandwidth consumption and miss rate. The only hardware cache parameters we are concerned with are the size, the associativity, and the eviction policy.

4.3.1 Dealing With the Index

There are two main issues with the index: latency and extra bandwidth consumption.

Latency

As stated before, the use of an index to store only blocks we need to store is analogous to the use of page tables to index virtual memory, which have similar latency issues. The solution for page tables is to take advantage of the principle of locality and keep a small fully associative cache or translation lookaside buffer (TLB) to map virtual addresses to page frames, so one only needs to do the full page table lookup if the virtual address needed is not in the TLB [HP03]. In the case where the latency involved in the TLB lookup is unwanted, a solution is to use virtual addresses for both the cache index and tags instead of the physical address allowing the elimination of the TLB. Such caches are called **virtual caches** as opposed to **physical caches** [HP03].

Virtual caches are not widely adopted as there are issues arising from having multiple processes [HP03]. Fortunately, most of these issues are not relevant to the case of multiple textures. The only issue that applies is the case where one switches between different processes/textures, requiring a cache flush. The solution is simply to widen the index/tag to include a process/texture identifier. We use such a virtual cache with a texture identifier prepended to each block's key as the tag in our block-based caches.

Using a virtual cache solves other problems besides the latency problem. It unifies the treatment of void blocks and occupied blocks in the texture, at the disadvantage of having void blocks fill up cache entries. Most importantly we can apply the block permutation techniques described in Section 3.2.3 to reduce conflict misses by choosing the index scheme carefully.

Due to the spatial locality of texel accesses consecutive lookup paths down the B-tree tend to have similar prefixes which we can leverage to reduce both the average latency for a lookup and bandwidth consumption for index blocks. One way to use it is to store n paths per texture unit and search the paths backwards, then forwards for the appropriate block for a given key in parallel like in an n-way associative cache. Most of the time we should not have to search very far backwards whereas we would have to travel the whole height of the B-tree if we searched forwards.

Bandwidth Consumed by the Index Blocks

If index blocks are not cached, we must load $h + 1$ index blocks on a cache miss where h is the height of the B-tree. This is an unwanted scaling of the bandwidth requirements which motivate us to investigate the effect of caching B-tree index blocks in our tests.

Suppose the index blocks are stored in the same cache as the data for the texture. This is possible for virtual caches if we add a bit to the tag to distinguish between an index block and a texture data block. If the index blocks in use at the time are needed frequently, that means that a lot of B-tree lookups are needed so there must be a lot of misses for the texture data. In this scenario the index blocks that are commonly used, like the ones near the root of the tree, basically live in the cache and exacerbate the bad situation. If the index blocks in use at the time are not needed because the cache performance with respect to the actual texture blocks is good, we expect the index blocks in use to be evicted by the time they are needed again, thus loading them into the cache only served to evict texture blocks. It is with this reasoning that we expect a separate cache for the index blocks to be needed and to noticeably lower the miss rate.

4.3.2 Reducing Bandwidth Consumption and the Miss Rate Due to the Texture Data

Bandwidth savings come automatically from not reading (or storing) void blocks with the amount saved dependent on the texture. Any additional bandwidth usage would have to come from the effect of the index blocks which we discuss in the section on the index. The same is not necessarily so for the total miss rate.

Recall that we use the term **texture block** to refer to an occupied block in the texture. As discussed previously, to reduce latency we can use a virtual cache which treats void blocks and occupied blocks the same by indexing the cache based on the key. This means that while bandwidth consumption is reduced, the miss rate is not. Also, the void blocks take up valuable cache space despite not needing all of it. One solution to this is to have a separate cache for the void blocks that does not have unneeded storage space.

Instead of storing void-blocks individually, we can use a fully associative cache to store a set of key intervals whose blocks are void instead, gleaned from a lookup for a void-block in the B-tree. We call this sort of cache a **void-range cache**. Looking up a block in the void-range cache then involves checking whether the (modified) key falls in any of the intervals in the cache in place of matching a tag stored in the cache. By using a range $[k_1, k_2]$ instead of a single key k, we compress multiple blocks which could take up multiple cache entries into one entry. This should result in a smaller working set in terms of the number of cache entries.

One thing to note is that if we use key intervals $[k_1, k_2]$, the index scheme in combination with the occupancy pattern of the texture affects the shape of the individual intervals in texture space. This, in interacting with the rasterization order and the texture orientation, could affect the working set size. For example, in Figure 4.3, travelling vertically through the texture would result in a larger working set size than travelling horizontally through it.

Chapter 5

Test Setup

In this chapter we describe the test setup and the input data we use. We also explain why particular data are chosen.

5.1 The Simulation Process

The experimental setup is based on trace driven programs arranged in a pipeline, shown in Figure 5.1. The individual components of the pipeline work as follows:

Generate texel requests: This stage takes as input a scene description and parameters on how to render it *e.g.* window size, rasterization order, geometry, shaders, and produces the texel requests that result from rendering the given scene.

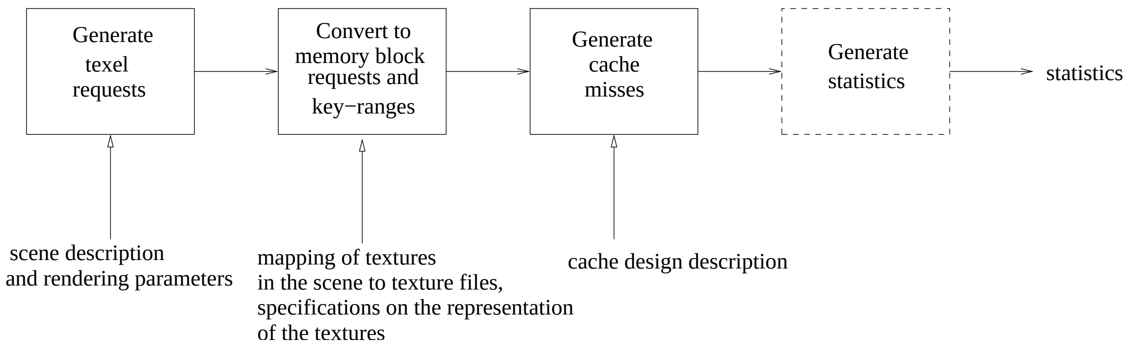

Generate texel requests
Convert to memory block requests and key−ranges
Generate cache misses
Generate statistics
statistics
scene description and rendering parameters
mapping of textures in the scene to texture files, specifications on the representation of the textures
cache design description

Figure 5.1: The inputs and data flow in the simulation pipeline used for the tests.

Convert texel requests to memory block requests and key-ranges: This stage uses a mapping from scene textures to sparse texture files and specifications on the representation of the sparse textures, including memory block size, to translate the texel requests to the necessary memory block requests and key-ranges. The key-ranges are produced to enable the simulation of range-caches and virtual caches.

Generate cache misses: This part of the pipeline takes as configuration input cache design specifications, simulates the cache(s) on the output from the previous stage of the pipeline, and produces the cache misses that are generated. The cache block size used is identical to the memory block size used in the previous stage.

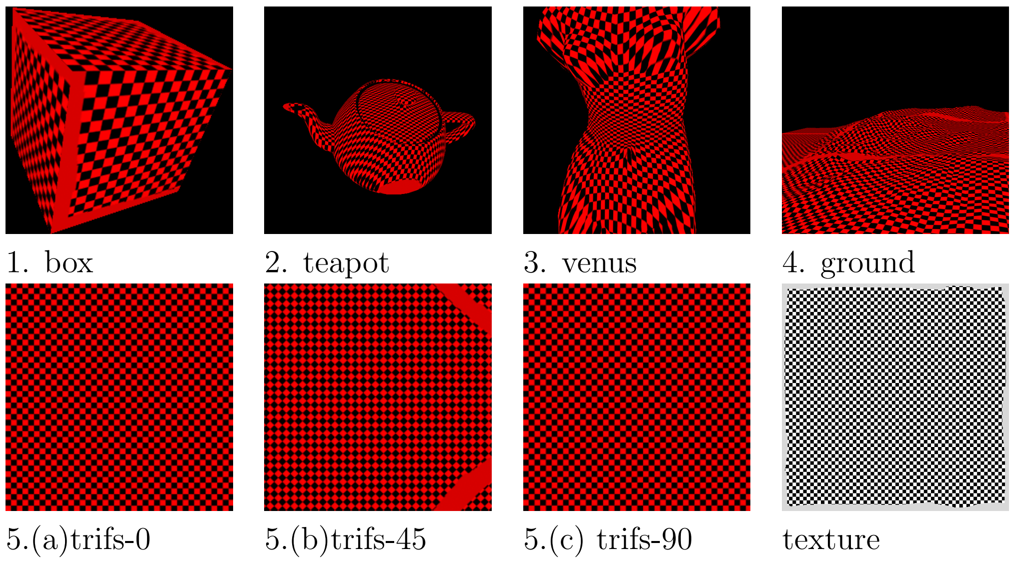


Figure 5.2: Scenes used for generating texel requests, rendered with the bordered checkerboard pattern at the bottom right.

Generate statistics: Several programs are used in this stage to generate a variety of types of output. We plot bandwidth versus cache-size and miss-rate versus cache-size graphs using data generated from the previous stage in the pipeline. Unlike other parts of the pipeline, a statistic generating program may take the output of many iterations of the previous parts of the pipeline, typically while varying a single variable, to produce its output. This deviation from a typical pipeline is indicated by the dashed line for the corresponding box.

5.2 Test Scenes

The test scenes (Figure 5.2) each consist of a simple object with a 2-D texture image mapped onto its surface. The scenes were chosen to give a variety of triangle sizes and MIP-map level changes. The texture coordinates were chosen so that there is no tiling except in the cases of “ground” and “box”, and there is as little texture magnification as possible. This is to limit the working set size reducing effect of tiling and texture magnification, which has already been explored in previous work [IEH99].

The image resolution of each scene was selected so that most of the texel accesses were to MIP-map levels of resolution 512×512 and 256×256. Lower resolution levels would be densely occupied, and higher resolution levels would require a larger image resolution, vastly increasing the amount of data needed to be generated, processed, and stored.

Here is a short description of each scene:

Scene 1: box

image resolution: 256×256

triangle size: large

LOD: gradual changes in the level-of-detail

Scene 2: teapot

image resolution: 512×512

triangle size: small

LOD: sharp changes in the level-of-detail around the silhouette edges

Scene 3: venus

image resolution: 512×512

triangle size: moderate

LOD: sharp changes in the level-of-detail around the curves

Scene 4: ground

image resolution: 1024×1024

triangle size: small and moderate

LOD: large range in level-of-detail, discontinuities in level-of-detail around mountain silhouettes

Scenes 5a,b,c: trifs-0, trifs-45, trifs-90: This is a set of theoretical test scenes consisting of a single zoomed-in triangle chosen to eliminate the working set size limiting effect of geometry. There is also only one constant level-of-detail. The suffix of the name of a scene indicates the rotation in degrees of the triangle counter-clockwise about the viewing axis; the various rotations allow for testing how texture orientation affects performance.

image resolution: 256×256

triangle size: one triangle that fills the screen

LOD: one constant level-of-detail

5.3 The Textures

The set of textures used is shown in Figure 5.3. The textures chosen give a variety of texture occupation patterns and occupation densities.

5.4 Other Parameters

The use of a B-tree to index the $n \times n$ blocks of the texture induces a blocked representation of the texture. The selection of the index scheme used to index the blocks then corresponds to the selection of a way to order the set of both occupied and unoccupied blocks of the texture. The permutation of the texture data blocks selected affects several things:

key-intervals: As noted in the section on our implementation of sparse texture, the shape of the parts of the texture that are represented by a single key-

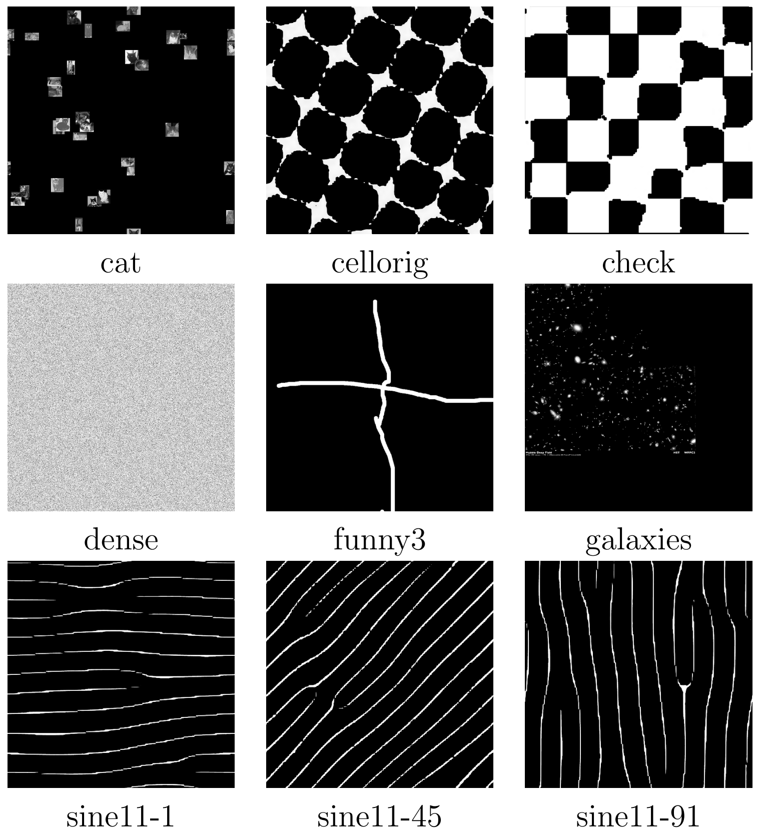


Figure 5.3: The test textures.

interval is affected by the index scheme, which could affect the working set size for a void-range cache. The shape of key-intervals when the index scheme is raster order tends towards long and thin. The shape of the key-intervals for Hilbert and Z-order tends to be less anisotropic, and closer to a square (Figure 5.4).

conflict miss avoidance: Since we use virtual caches, with the key for each block

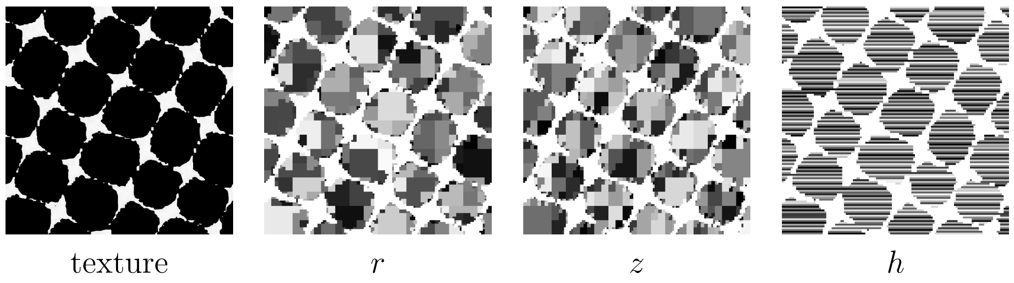


Figure 5.4: The texture *cellorig* and how the void key-intervals are distributed in it for various index schemes. The white areas are occupied blocks, the gray areas are the void key-intervals.

as the virtual address for the block, the B-tree index scheme used affects where in the cache a block is placed for a non-fully associative cache. Arranging the blocks in raster order results in conflict misses between blocks in the same column. Using Z-order ensures that any four adjacent blocks will not map to the same cache-index (for a cache that holds more than four entries). Hilbert order does not ensure that any four adjacent blocks will not conflict, but neither does it have anything as bad as all blocks in the same column conflicting.

cost of computation: The effects of the factors above must be balanced against the cost associated with the scheme. Z-order can be implemented to be zero-cost in hardware by interleaving the bits of the x and y coordinates. Hilbert order keys must be generated bit-by-bit sequentially.

We denote the raster order, Z-order, and Hilbert order index schemes by r, z, and h respectively. They cover a range of behaviours for all the factors listed.

Here are descriptions of the other test parameters:

block size: We test using block sizes of 64, 256, and 1024 bytes a 4-byte texel, and 32-bit keys and pointers. These correspond to 4 texels $\times$ 4 texels, 8 texels $\times$ 8 texels, and 16 texels $\times$ 16 texels respectively.

rasterization order: The rasterization orders we test are raster order, Z-order and Hilbert order. We denote them by the symbols R, Z, and H respectively. Hilbert order is tested because of its high spatial locality. Raster order is tested as a conventional case and because it is at the other end of the scale of spatial locality. Z-order is tested because it is between the other two in the scale of spatial locality, and is simpler than Hilbert-order rasterization.

sampling method: We use trilinear sampling from a MIP-map pyramid in most of the tests because it is an anti-aliasing feature common across most modern graphics hardware, and the MIP-mapping is needed for coherent memory accesses in order to ensure good cache behaviour. We use table FELINE sampling to test the effects of less coherent texture accesses due to the use of an anisotropic filtering method. Some tests use bilinear sampling from the nearest MIP-map level to restrict the sampling to one level of the MIP-map.

shader: We use the function `test_turbulence` described in Section 2.1.3 to represent the effects of a programmable shader in our tests. The parameter controlling the degree of turbulence is ω.

shader derivatives: Whether or not to use shader derivatives when computing the level of detail for MIP-mapping is an option.

cache design: We vary the cache design as described in Section 4.3. If the cache design allows for a range of cache associativities, we run tests with 1, 2, 4, 8-way set-associative, and fully associative caches. We also vary the eviction policy between LRU, FIFO and optimal.

5.5 Metrics

We use the bandwidth consumption curve as the primary measure of cache performance. Note that the bandwidth consumption curve is the miss rate curve scaled by the block size.

It is hard to quantify the average memory access time without a hardware implementation. The hit time can possibly vary depending on whether there is a separate void cache and how a separate void cache is implemented. The miss penalty is also variable as no texture data needs to be downloaded if the miss involves a void part of the texture. The use of an index cache also contributes to variation in the miss penalty. Because of these complications, we use the average number of index blocks accessed to lookup a texel as our measure of the extra latency due to the indexing.

Chapter 6

Results

In order for the B-tree indexing method to be a viable alternative to other sparse texture compression methods and the standard dense texture storage method, it must demonstrate good texture compression, lower bandwidth consumption, and low additional latency. This section provides the results and arguments that show that our method satisfies all the requirements.

6.1 Summary of Unimportant Results

We found the following in our tests:

- The anisotropic nature of rasterization order R and index scheme r resulted in poor performance.

- Rasterization orders H and Z produced similar results.
- Index schemes h and z produced similar results.
- The combination of rasterization order H or Z and index scheme z produced negligible conflict misses between neighbouring blocks. When index scheme h was used instead, we got somewhat more conflict misses at small cache sizes,
- The H/Z, h/z combination was also tolerant of miss rate increasing factors like a turbulence shader and table FELINE sampling when shader derivatives are used.
- Using FIFO was comparable to using LRU.

Due to the above we will use rasterization order H, index scheme z, a fully associative cache, and LRU eviction policy for demonstrating most of the results.

We also confirmed previous work showing that the rasterization order used has little effect on the miss rate when the triangles are small [HG97]. In the case of rasterization orders H and Z, this resulted in the results improving as triangle size increased, with *teapot* scene giving the worst miss rate/bandwidth consumption curves, and the *trifs* scenes giving the best ones. We will usually use these two scenes to demonstrate the results.

Unless otherwise specified, the graphs shown are for block size 256.

6.2 Texture Compression

The tables in Figures 6.1 and 6.2 give the texture compression ratio with and without the index overhead for various textures and block sizes, while the table in Figure 6.3 gives the index overhead as a proportion of the amount of memory used to store the occupied blocks.

Both Figures 6.1 and 6.2 demonstrate the memory needed to store the texture roughly scaling with the texture occupancy. Figure 6.3 shows that the index overhead is unacceptable for a block size of 64 bytes. Figure 6.2 shows that the extra memory needed due to the blocking, or the “edge effect”, is high for a block size of 1024 bytes.

Texture	block size = n bytes		
	$n = 64$	$n = 256$	$n = 1024$
cat	0.15	0.15	0.22
cellorig	0.31	0.35	0.47
check	0.75	0.79	0.96
dense	1.17	1.03	1.01
funny3	0.07	0.08	0.12
galaxies	0.22	0.32	0.46
sine11-1	0.23	0.30	0.49
sine11-45	0.32	0.42	0.70
sine11-91	0.23	0.31	0.50

Figure 6.1: The compression ratio of the B-tree indexing method for the 512×512 level of the MIP-map.

Texture	block size = n bytes		
	$n = 64$	$n = 256$	$n = 1024$
cat	0.13	0.15	0.21
cellorig	0.27	0.34	0.47
check	0.64	0.76	0.95
dense	1	1	1
funny3	0.064	0.082	0.12
galaxies	0.19	0.3	0.46
sine11-1	0.2	0.29	0.48
sine11-45	0.28	0.4	0.7
sine11-91	0.2	0.3	0.5

Figure 6.2: The compression ratio of the B-tree indexing method ignoring the index overhead for the 512×512 level of the MIP-map.

6.3 Texture Bandwidth

The graphs in Figure 6.4 show that as expected the bandwidth consumption due to the texture data scales with the texture occupancy.

We intuitively expect a separate cache devoted to the void parts of the texture to decrease bandwidth consumption due to the texture blocks. However, as depicted in Figure 6.5, in reality there is hardly any bandwidth reduction when a separate void cache is used.

The reductions in bandwidth are greatest for scene *teapot*, and least for scene *trifs-90.* This is explained by the way the occupied blocks tend to cluster together in the texture so that a working set is more likely to contain both void blocks and texture blocks when the spatial locality is poor, which we do not get when

Texture	r_u for block size = n bytes		
	$n = 64$	$n = 256$	$n = 1024$
cat	0.17	0.034	0.014
cellorig	0.17	0.035	0.01
check	0.17	0.034	0.0092
dense	0.17	0.034	0.0098
funny3	0.17	0.036	0.0081
galaxies	0.17	0.035	0.011
sine11-1	0.17	0.035	0.01
sine11-45	0.17	0.035	0.0098
sine11-91	0.17	0.035	0.0099

Figure 6.3: The memory overhead involved in the B-tree index for the 512×512 level of the MIP-map. Expressed in terms of $r_u = \frac{\text{number of index blocks}}{\text{number of texture data blocks}}$.

rasterizing in H or Z order. The non-cold miss rate is also so low as to make any reductions in bandwidth necessarily small.

The miss rate curves for a separate void cache (Figure 6.6) are flat enough so that enlarging the cache has little effect on the miss rate. Figure 6.7 demonstrates that using a range cache is effective at lowering the miss rate due to the void parts of the texture, even when the range cache has very few entries. The miss rates that result are low enough that unless the texture is very sparsely occupied (*e.g. cat*), they are small compared to the miss rate due to the occupied parts of the texture.

From the above, we have that the bandwidth consumption due to the texture data scales with the texture occupancy. If additionally we use a separate void range cache, we also have that the miss rate scales with the texture occupancy.

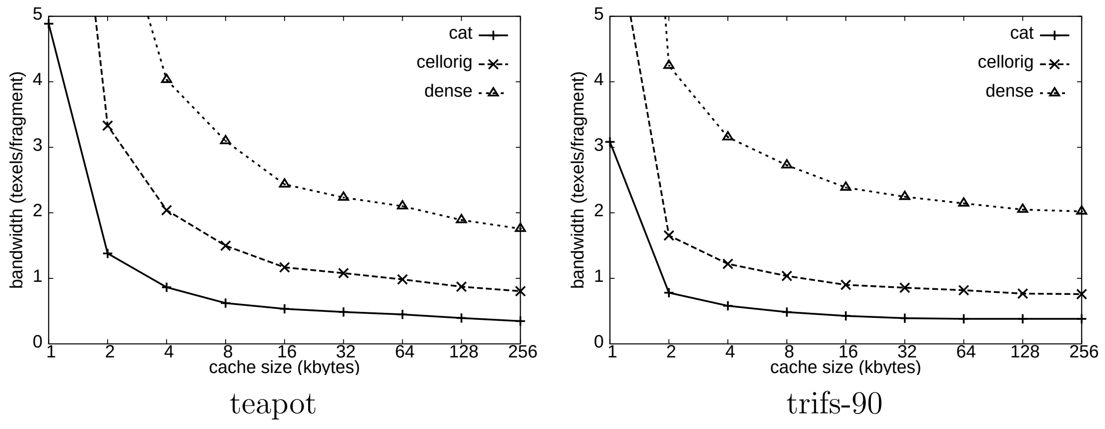


Figure 6.4: Bandwidth consumption due to texture blocks for various textures and scenes *teapot* and *trifs-90*. The cache is a fully associative cache that stores both texture and void blocks.

6.4 The Index

Figure 6.8 demonstrates that the bandwidth consumption for the index blocks scales with the texture occupancy. We can also see from the figure that it is somewhat sensitive to spatial locality, as the steepness of the curve for scene *teapot* is much greater than that for scene *trifs-90*.

Figure 6.9 shows that for sufficiently large caches, a larger block size results in less index bandwidth consumption overhead. Unfortunately the Figure 6.10 shows that the reductions are more than made up for by the extra bandwidth due to the edge effect in the texture. Clearly, increasing the block size to 1024 bytes does not decrease the overall bandwidth consumption. Though the same can be

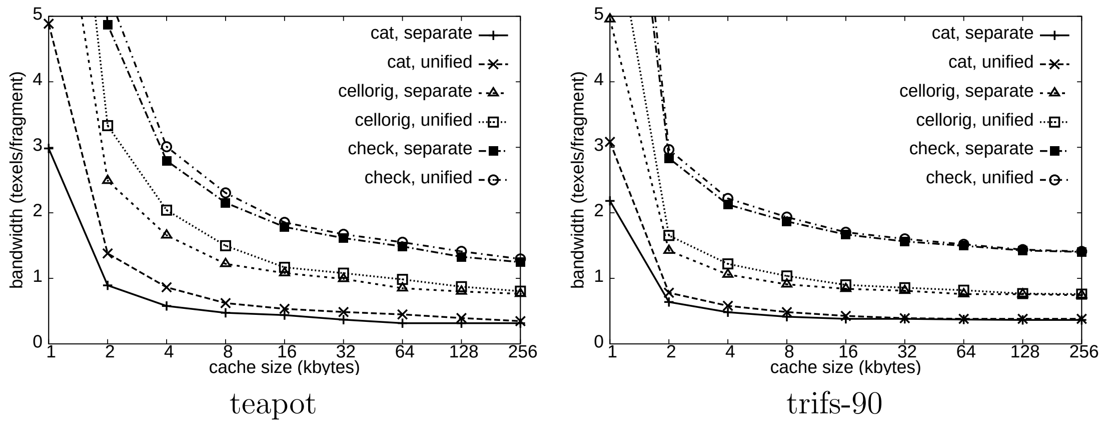


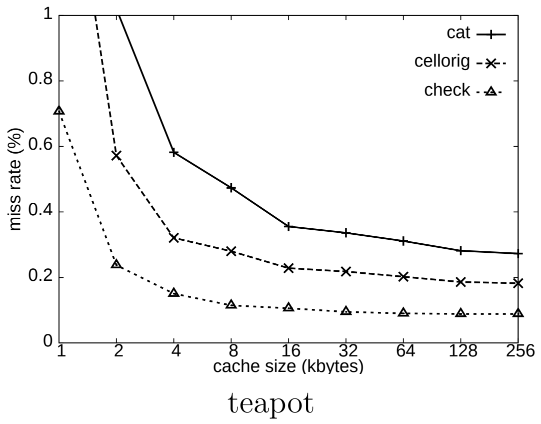
Figure 6.5: Comparison of texture block bandwidth consumption when using a unified cache and when using a separate void cache. The cache size is the size of the cache that stores the texture blocks.

said of increasing the block size to 256 bytes, the increase in overall bandwidth consumption is modest in this case.

Lacking a hardware implementation, we use the average number of index blocks accessed per fragment as the measure of latency. Figure 6.11 gives this for several scenes and textures. Note that the latency scales with the texture occupancy, reflecting the height of the B-tree increasing with texture occupancy (see figure 6.12). The number when the block size is 64 is high, indicating that we should avoid using such a small block size if latency is an issue.

In our tests, the block size that best balances bandwidth consumption and latency of access is 256 bytes.

The effect of using path-caches to store the paths down the B-tree for backwards



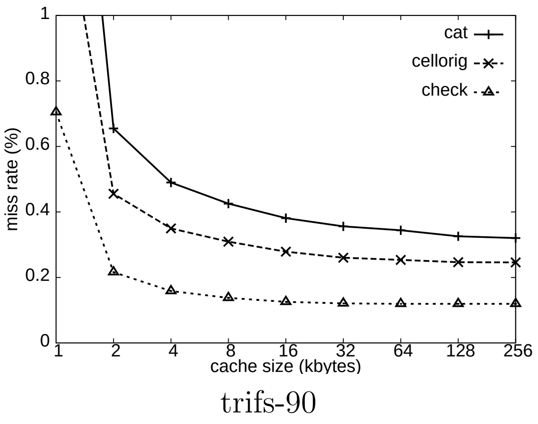
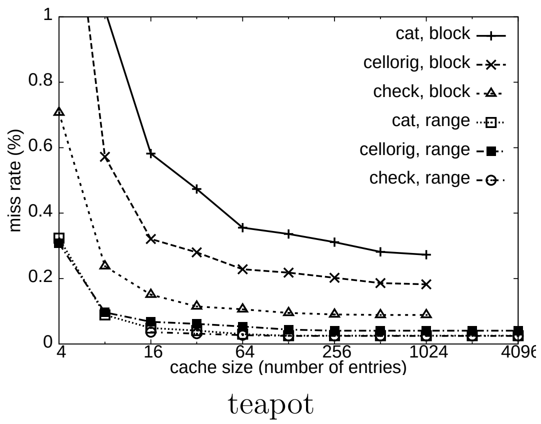


Figure 6.6: Miss rates for the void block-based cache for various textures.



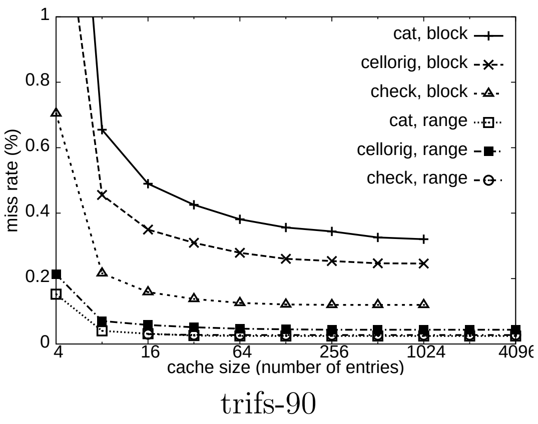


Figure 6.7: Comparison of miss rates for a void block cache and a void range cache.

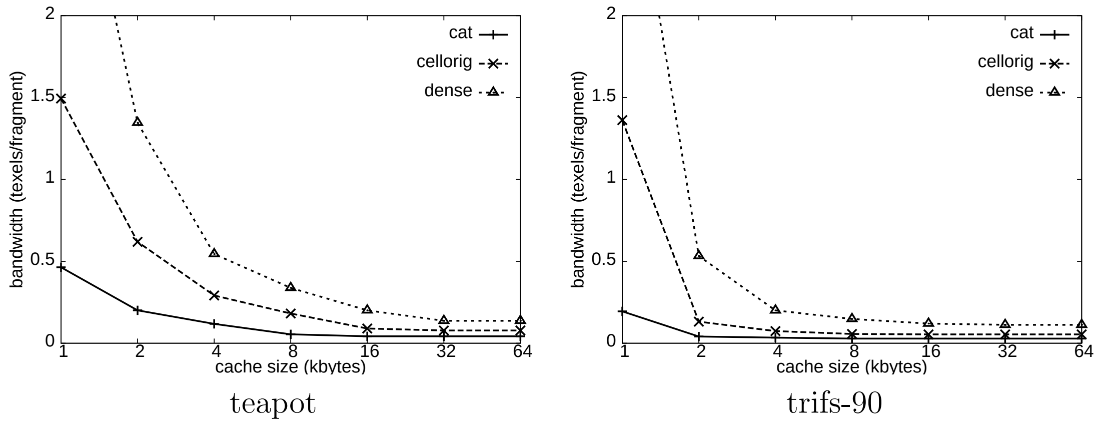


Figure 6.8: Index bandwidth as a function of the index cache size for several textures. We use a texture block cache of size 16K and a void range cache of size 8192 entries.

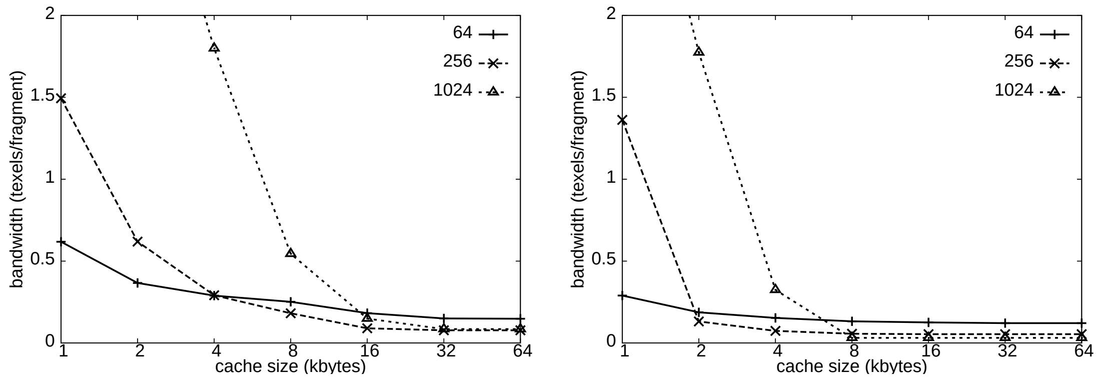


Figure 6.9: Bandwidth consumption for the index blocks for various block sizes as the index cache size varies. The texture is *cellorig*. We use a texture block cache of size 16K and a void range cache of size 8192 entries.

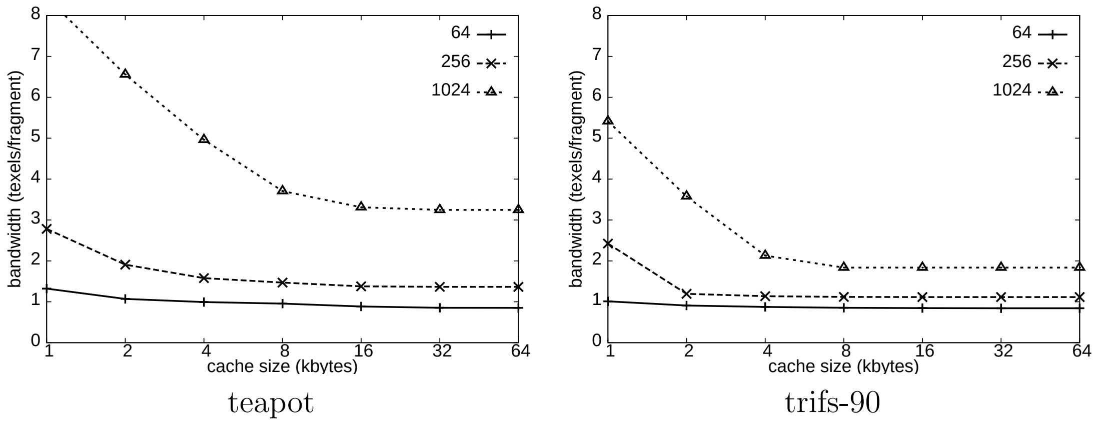


Figure 6.10: Total bandwidth consumption for various block sizes as the index cache size varies. The texture is *cellorig*. We use a texture block cache of size 16K and a void range cache of size 8192 entries.

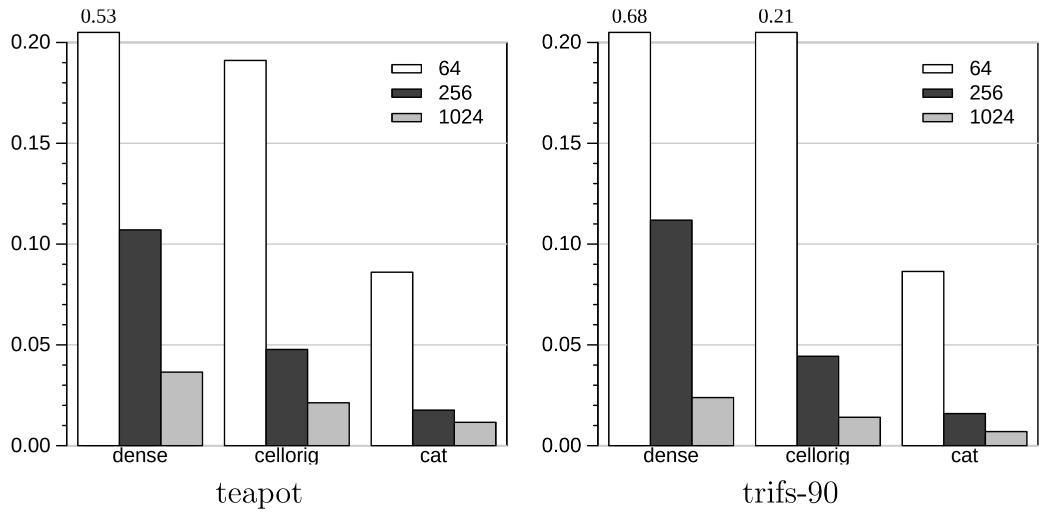


Figure 6.11: The average number of index blocks accessed per fragment for various block sizes, textures, and scenes *teapot* and *trifs-90*. We use a texture block cache of size 16K and a void range cache of size 8192 entries.

Texture	B-tree height for block size $= n$ bytes		
	$n = 64$	$n = 256$	$n = 1024$
cat	4	2	2
cellorig	5	3	2
check	5	3	2
dense	5	3	2
funny3	4	2	1
galaxies	5	3	2
sine11-1	5	3	2
sine11-45	5	3	2
sine11-91	5	3	2

Figure 6.12: This table shows the effect of block size on B-tree height for the various textures and block sizes.

searching is shown in Figure 6.13. While the reduction is small when the texture is very sparse, the use of path-caches is noticeable when the texture is denser.

6.5 Discussion

Bandwidth consumption and latency of access constrain current graphics hardware performance. Here we discuss and compare measures of these two factors for our B-tree indexing method, Kraus and Ertl's uniform grid indexing method, and the standard dense texture storage method.

As can be seen in Figures 6.4 and 6.8, the bandwidth consumption scaling with the region of interest and the low index bandwidth consumption overhead combine to give bandwidth consumption that is much lower than that for the standard dense

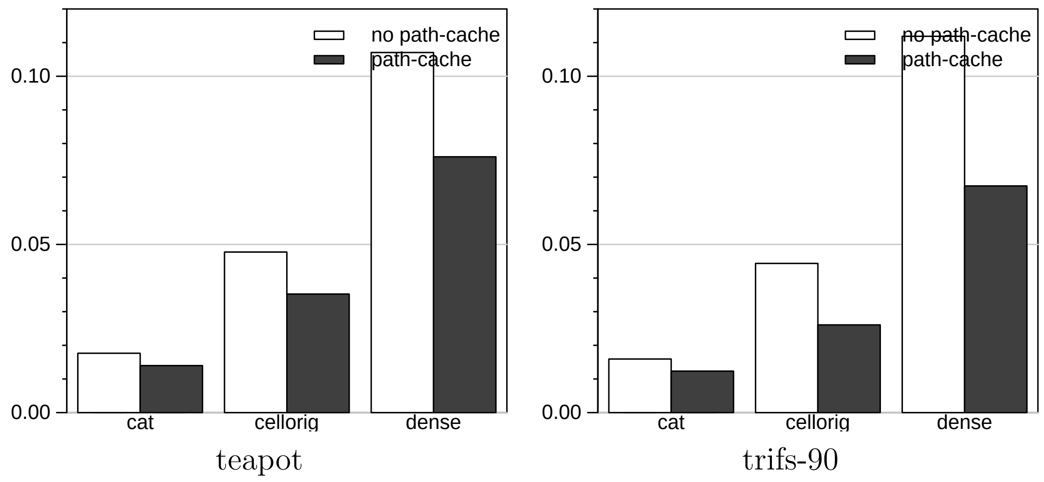
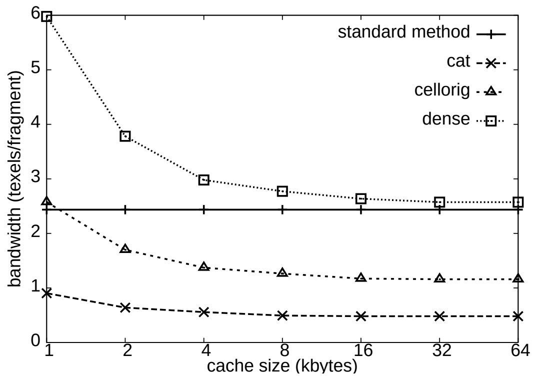


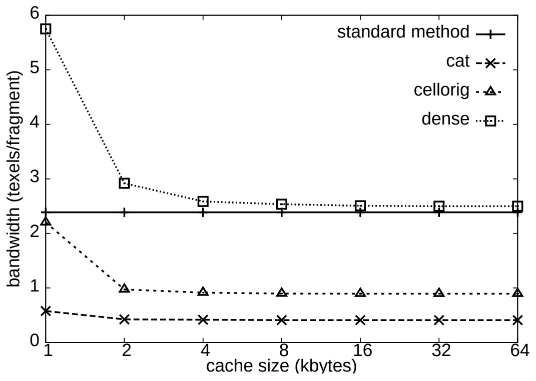
Figure 6.13: Effect of having a path cache in use on latency. We use a texture block cache of size 16K and a void range cache of size 8192 entries.

texture method (Figure 6.14) unless the texture is densely occupied. Even when the texture is dense, the index bandwidth consumption overhead is not intolerable.

Now compare the index bandwidth overhead for our method (Figure 6.8) with the bandwidth above the theoretical limit for the uniform grid index method (Figure 6.15). They are not directly comparable as the uniform grid index method has an advantage from sampling bilinearly from one texture image instead of trilinearly probing into a MIP-map pyramid so less data is involved. There are also other factors affecting performance. Despite this, the padding of the texture tiles combined with sensitivity to the edge effect results in much higher bandwidth consumption overhead above the theoretical limit than our method. In fact, we do not obtain significant savings in bandwidth consumption for a sparse texture unless the cache



teapot



trifs-90

Figure 6.14: The total bandwidth consumption for the B-tree method compared to the standard method for several textures as the index cache size varies. For the B-tree method, we use a texture block cache of size 16K and a void range cache of size 8192 entries.

is very large (see the graph for texture *cellorig* and block size 256 in Figure 6.15).

A better measure of performance than the miss rate or bandwidth consumption is the average memory access time, given by:

$$\text{average memory access time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

However, the average memory access time is hard to quantify without a hardware implementation. The miss penalty is variable because we do not always need to download a memory block and because of the indexing. The hit time may also be variable depending on the type and size of void cache we use.

As an approximation, we can divide the overhead affecting the average memory access time into two parts: the additional hit time due to accessing index blocks in the cache, and the additional time needed to download the index blocks into the cache. The latter corresponds to the extra bandwidth due to the index, which we have shown is relatively low and usually overcompensated for by not having to download void parts of the texture.

The former is given by

$$\text{average number of index blocks accessed per texel access} \times \text{hit time}_{\text{index cache}}$$

. Since there are eight texel accesses per trilinear probe, Figure 6.13 gives the average number of index blocks accessed per texel access scaled by eight. Note that since we only access the index on a texture data miss, the average number of index blocks accessed per texel access is correlated with the total miss rate, which depends on the texture occupancy if a void range cache is used. For the worst case of texture *dense* and no path-cache, this translates into approximately $0.11 \div 8 \approxeq 0.014$ index blocks accessed per texel access. This corresponds to the relatively low 1.4% hit time overhead over the standard method if the hit time for the index cache is the same as the hit time for the texture cache.

The average memory access time for the uniform grid index method is large in comparison. The dependent texturing results in an extra texel access for each sample from the actual texture data, which comes with a doubling of the hit time, not accounting for the time needed for the shader to compute where to sample in the texture data. Moreover, we have shown via the bandwidth consumption curves, which are just the miss rate curves scaled by the number of texels in a memory block, that the miss rate for the uniform grid index method is higher than that for the B-tree index method.

In discussing the average memory access time we should note that because we use virtual caches, we only go to the index on a miss, so the latency due to the indexing in our method is hideable by a prefetching texture cache architecture such as the one proposed by Igehy *et al.* [IEP98]. Since the uniform grid index method uses dependent texturing, the latency due to the index lookup there is not hideable.

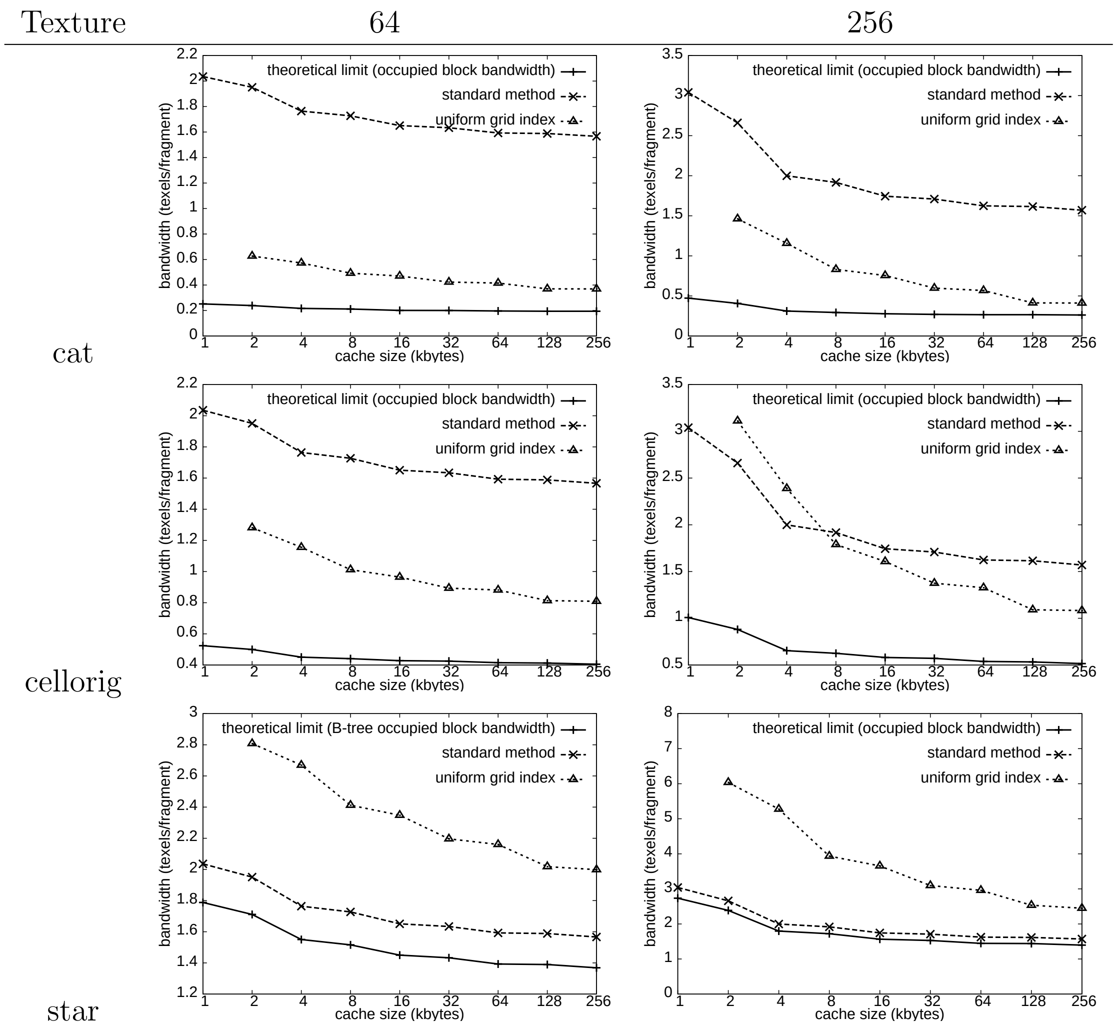


Figure 6.15: Bandwidth consumption for scene *trifs-90*, various textures and block sizes. A direct mapped cache is used for the theoretical limit and the standard method, and a 2-way set-associative cache used for the uniform grid index method.

Chapter 7

Conclusions

This document presents a hardware compatible implementation of sparse textures based on B-tree indexing and explores cache designs for it. We demonstrate that it is possible, with the appropriate cache design and other parameters, to have the bandwidth consumption and miss rate due to the texture data alone scale with the texture occupancy. We also show that the additional bandwidth consumption and hideable latency due to the B-tree indices is low. Furthermore, the caches necessary for this can be quite small. These results are highly dependent on good spatial locality in the texel accesses, which is possible when rasterizing in Hilbert order and indexing the texture blocks in Z-order.

Hardware manufactures are currently looking at implementing a virtual memory system for GPUs. Such a system would require an indexing scheme very similar to that explored here. Even for dense textures, a B-tree index simplifies memory management and avoids memory fragmentation. Thus there is motivation for further future work related to our method.

7.1 Summary of Results

The tables in Section 6.2 show that the amount of memory occupied by a texture scales with the texture occupancy. Also, the space overhead due to the index can be less than 4% of the size of the texture data stored if the block size is at least 256 bytes.

The tests in Section A.5 and the graphs in Section 6.3 show that the bandwidth consumption and miss rate due to the texture data alone can scale with the texture occupancy. Section A.5.2 shows that a separate void cache is ineffective at reducing the bandwidth consumption when there is good spatial locality.

The tests in Section A.7 and the graphs in Section 6.4 show that the bandwidth consumption and latency due to the index are low and scale with the texture occupancy. They also show that the best block size balancing the bandwidth consumption and latency of access in our tests is 256 bytes. Section A.7.4 demonstrates the latency reducing effect of using path-caches.

7.2 Applications

The applications of sparse textures are not restricted to the simple pasting of images onto polygons, but can be applied anywhere normal textures can be applied. As defined in this document, sparse textures do not have any form of "adaptive sampling", so their memory saving advantages are not widely applicable to different data. They are primarily directly useful when the region of interest sparsely occupies the texture space, as is often the case with 3D volumes and higher dimensional data. Specific examples where sparse textures are useful include:

- alpha masks
- decals
- map of locations of objects
- thresholded height fields, polynomial texture maps, deep shadow maps
- star fields
- 3D surfaces
- 3D surface textures

Storing adaptive texture maps in sparse textures, as described in the introduction, would get around the sparse data limitation.

7.3 Future Work

There are many avenues left unexplored, particularly in implementation and texture compression.

Our tests are based on software simulations involving 2D sparse textures. It would be simple to extend the software implementation to three or higher dimensions. More cache tests would need to be run as the way a 3D texture is accessed is different from the way a 2D texture is accessed. Some ways in which a 2D texture could be accessed are not tested for in our tests, in particular the accessing of more than two MIP-map levels in one texture for each fragment. A hardware implementation of the B-tree sparse texture and the supporting cache structures would also be useful to compare hardware costs.

In order to be more generally useful, an implementation of an adaptive texture map scheme is needed. One way to do this is to incorporate sparse textures into a compression scheme like the wavelet decomposition/reconstruction described in the introduction. Another way to do this is extend the B-tree sparse texture to have adaptive sampling built in. This would be possible with the addition of some scale bits, *i.e.* instead of just $(\text{key}, \text{pointer})$ pairs the B-tree stores $(\text{key}, \text{pointer}, \alpha)$ triplets, where α is a scale factor. Sparse textures could be used in a texture synthesis scheme such as one described by Wei [Wei01]. In any case, the texture access patterns in a more general texture compression scheme could be sufficiently unusual to necessitate a specific supporting cache design.

Appendix A

Appendix

A.1 Basic Tests for Dense Textures

We start our tests by seeing what kind of miss rate curves the different rasterization orders give in a conventional setup *i.e.* ignore the effect of the index blocks, then see how rotating the scene or zooming in and out changes the miss rate curves.

A.1.1 Test: Basic Curve Shapes

The first test establishes a baseline relationship between the rasterization order and the shape of the miss rate curve for a non-sparse texture. The test input is as follows:

scene: trifs-0

texture: dense

rasterization order: R, Z, H

sampling method: trilinear

block size: 64 – the smallest size is used to reduce aliasing artifacts from the discreteness of the blocks

cache design:

- **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).

  associativity: full

  eviction policy: LRU, optimal

index scheme: r (irrelevant for a fully associative cache)

The results are in Figures A.1 and A.2.

Discussion

The graphs in Figure A.1 show that there is a fundamental difference between the shapes of the miss rate curve for the Z and H-order rasterizations and the shape of the miss rate curve for the conventional R-order.

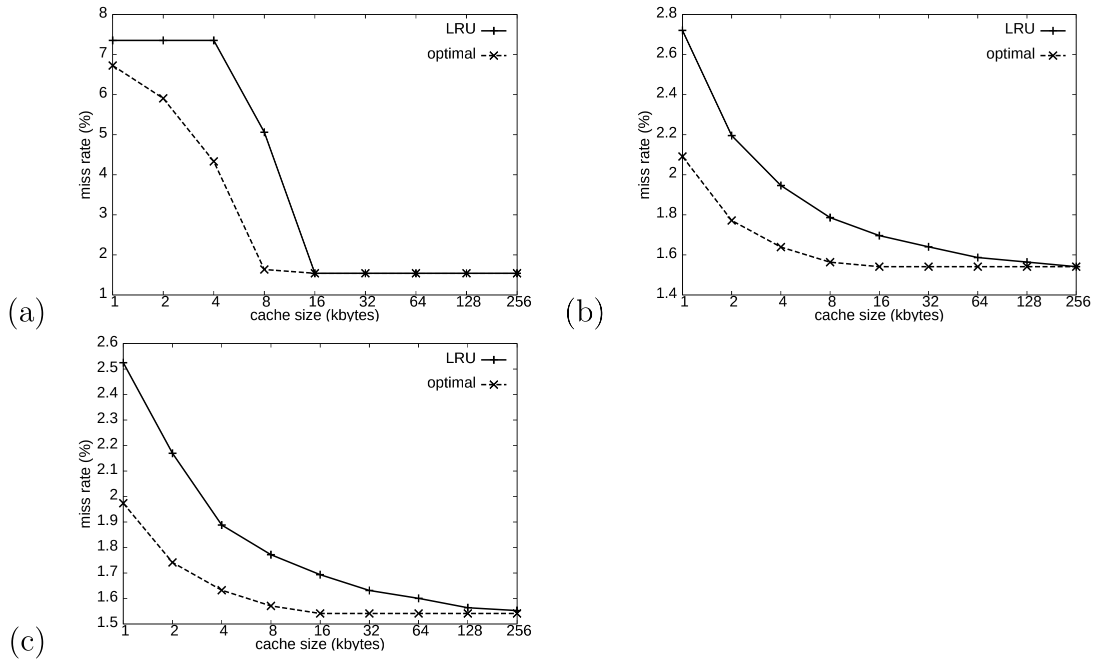


Figure A.1: Graphs of the miss rate *vs.* cache size for LRU and optimal eviction policies. The rasterization orders used for each graph are (a) R (b) Z and (c) H.

There is a clearly defined step in the curve corresponding to a distinct working set size in the case of R order. Note that the number obtained for R order agrees with the analysis in that for the given case the working set must contain enough texture to rasterize a scanline [HG97].

There is no well defined working set for either Z or H order. This can be explained by the shape of the Z-order and H-order curves, which are hierarchically self-similar in 2×2 blocks, so at each level there is a working set roughly corresponding to the texture data needed for a 2×2, 4×4, 8×8, *etc.* block of pixels.

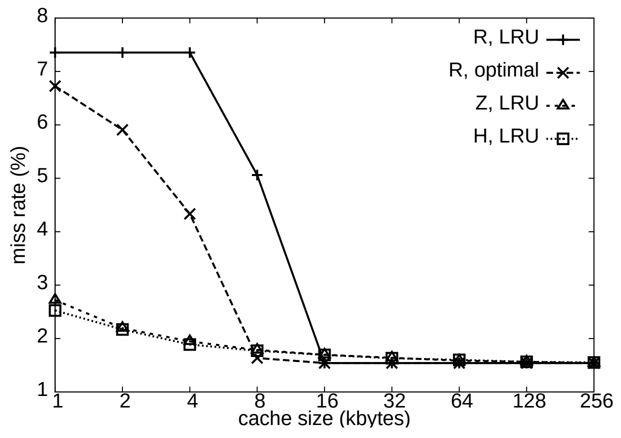
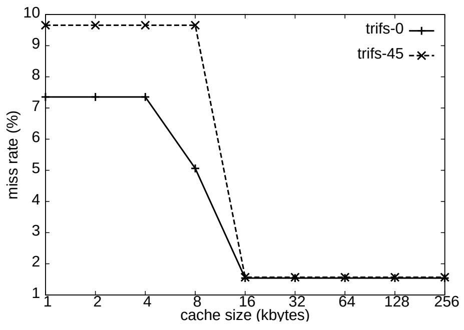
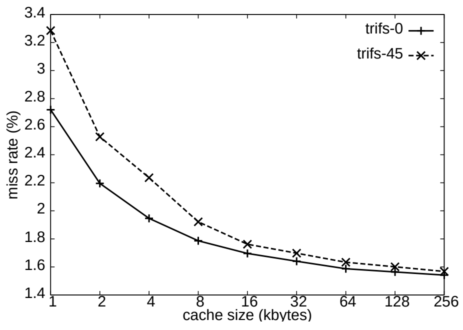


Figure A.2: Comparison of the miss rate curves for rasterizing in R, Z, and H order.

This, coupled with the cache sizes being powers of 2, results in a smooth decline for their miss rate curves. Additionally, the miss rate curve for Z-order is always higher than the corresponding miss rate curve for H-order, which is explained by the poorer spatial locality of Z-order.

As expected, H and Z orders have much lower miss rates than R order at most cache sizes due to their greater spatial locality (Figure A.2). In fact, the LRU eviction policy curves for H and Z are lower than the optimal eviction policy curve for R. For very large cache sizes R order performs slightly better than the other two. This is because when rasterizing using Z or H order, the texture blocks used to render some of the pixels on the edge of a $2^n \times 2^n$ block of pixels belong to a working set that is larger than the working set for rasterizing in R-order, and

(a)


(b)


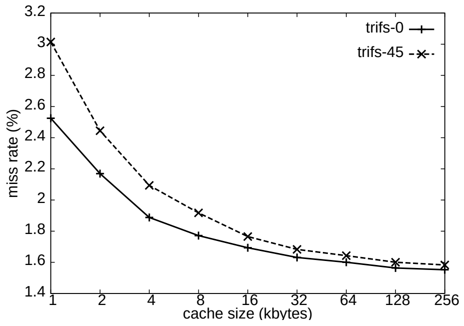
(c)


Figure A.3: The effect of changing the orientation of the viewpoint. The graphs are for a cache eviction policy of LRU and rasterization orders (a) R, (b) Z, and (c) H.

contribute misses other than cold misses to the miss rate for cache sizes that are smaller than that working set size.

A.1.2 Test: Effect of Texture Orientation

The second test is to see the effect of rotating the scene. The test parameters are:

scene: trifs-0, trifs-45

texture: dense

rasterization order: R, Z, H

sampling method: trilinear

block size: 64

cache design:

- **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).
  - **associativity:** full
  - **eviction policy:** LRU, optimal

index scheme: r (irrelevant for a fully associative cache)

The results are in Figures A.3.

Discussion

The miss rate for scene trifs-45 is greater than the corresponding miss rate for scene trifs-0. This is translated into a steeper curve for Z and H order, and a higher and steeper step for R order. Since the rotation of the scene is about the view axis, the number of texture blocks needed to render the scene is little changed between scenes. We attribute the increase in miss rate instead to the increase in the number of texture blocks needed to render a scanline and a $2^n \times 2^n$ block of pixels, due to the texture block boundaries no longer being aligned with the edges of the rasterization curves.

A.1.3 Test: Effect of Changing $\lambda - \lfloor\lambda\rfloor$

The third test deals with the effect of changing the fractional part of the value λ by slightly zooming in and out of the scene. This was shown to dramatically affect the texture memory bandwidth consumption in typical test scenes [IEH99]. The test parameters are:

scenes: trifs-0, trifs-0′, trifs-0″. The latter two are slightly zoomed in or out versions of the first scene. Their corresponding values for λ are 9.313409, 9.991481, and 9.106958 respectively.

texture: dense

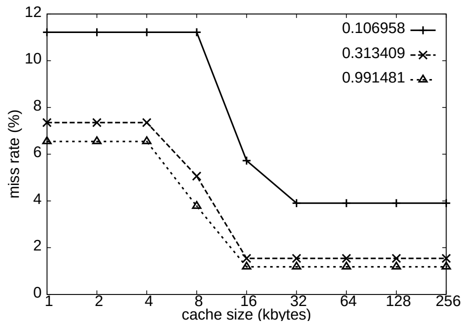
rasterization order: R, Z, H

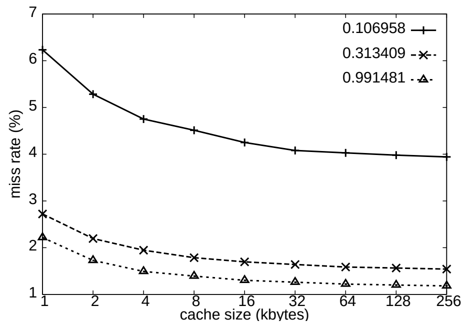
sampling method: trilinear

block size: 64

cache design:

cache 1: One cache for only texture blocks (T blocks) and void blocks (V blocks).

(a)


(b)


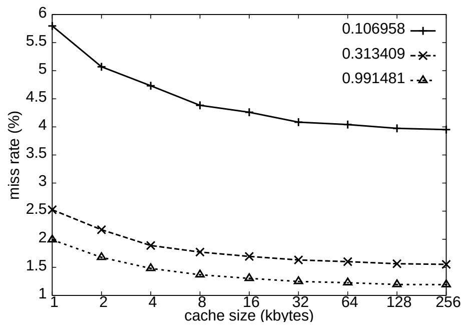
(c)


Figure A.4: The effect of changing $\lambda - \lfloor \lambda \rfloor$ on the miss rate curve for rasterizing in: (a) R order (b) Z order (c) H order.

associativity: full

eviction policy: LRU, optimal

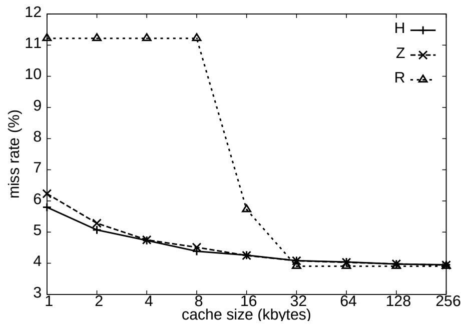
index scheme: r (irrelevant for a fully associative cache)

The results are in Figures A.4 and A.5.

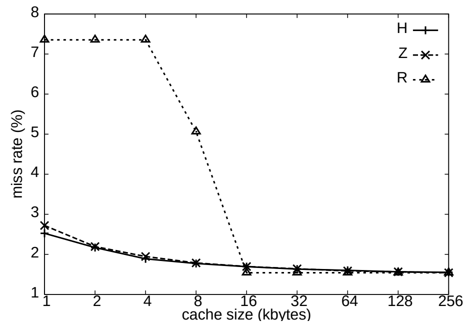
Discussion

This has a greater effect on the curves than rotation. For raster order R, as the fractional part of λ gets close to 0, the working set doubles from when the fractional part of λ is close to 1. The cold miss rate, indicated by the miss rate for very

(a)



(b)



(c)

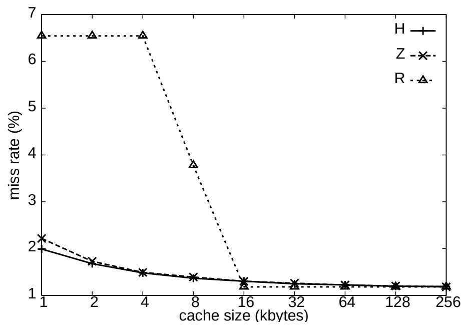


Figure A.5: Graphs showing the curves for all rasterization orders for (a) $\lambda - \lfloor\lambda\rfloor = 0.106958$ (b) $\lambda - \lfloor\lambda\rfloor = 0.313409$ and (c) $\lambda - \lfloor\lambda\rfloor = 0.991481$.

large cache sizes, greatly increases for all rasterization orders. The steepness of the curves for Z and H also increases. These changes are explained by the increased texture block-to-fragment ratio, which increase the number of texture blocks involved in rendering the scene, hence the increase in the number of cold misses. It also increases the number of texture blocks needed to cover a scanline or a $2^n \times 2^n$ block of pixels, which similarly to rotating the scene, results in a larger working set for R order rasterization and a steeper curve for Z and H order. However, the conclusions for the initial test still hold for all the values of lambda; the curves still have the same basic shape, Z and H are still pretty close, with Z higher than H, and both are well below R.

A.1.4 Summary

In conclusion, if there are no other factors breaking up the scene for better spatial locality, it is desirable to use either Z or H order rasterization because of the lower miss rates, flatter miss rate curve shape, and less additive sensitivity to changes in the block-to-fragment-ratio.

A.2 Basic Tests for Void Range-Cache

Our next set of tests explores the effects of storing a key-interval instead of a block in the cache on the basic miss rate curve. The scenario is based on caching the void key-intervals for a sparse texture and ignoring the effects of the B-tree index. More specifically, we focus on the effects of the interaction of the index scheme chosen with the rasterization order and texture orientation on the miss rate curve.

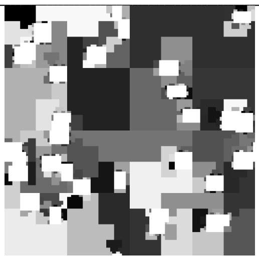
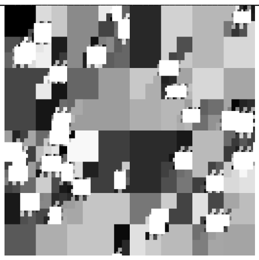
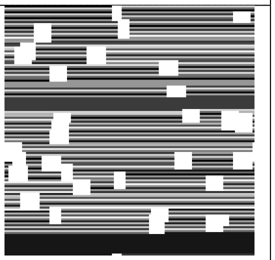
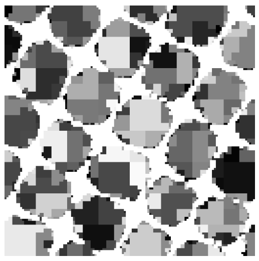
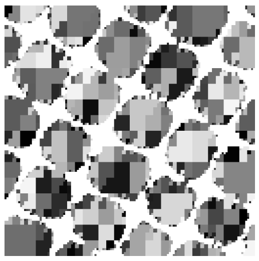
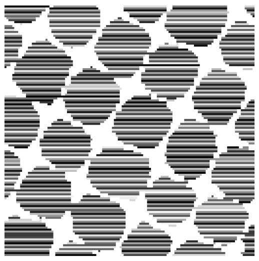
A.2.1 The Index Scheme and the Layout of the Key Intervals

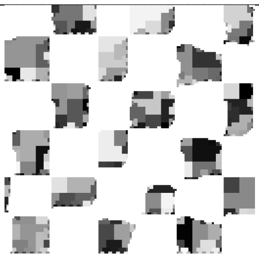
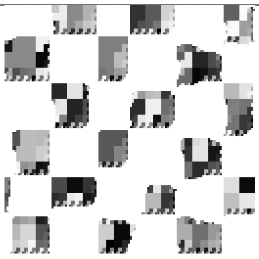
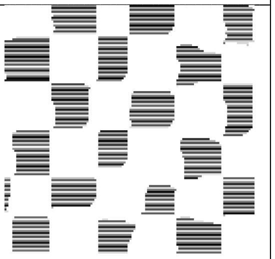
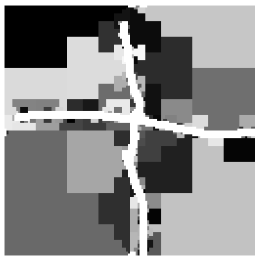
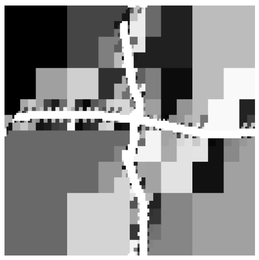
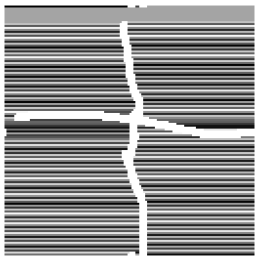
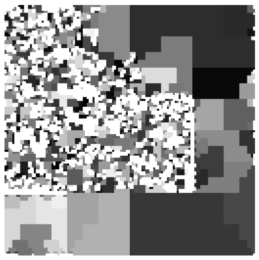
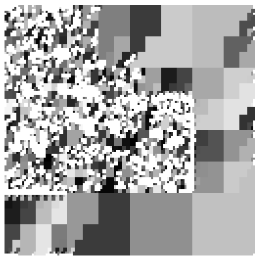
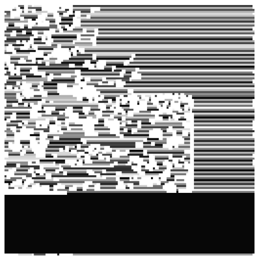
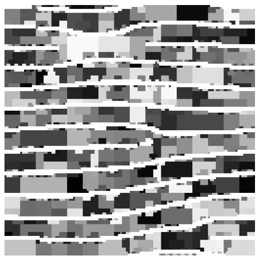
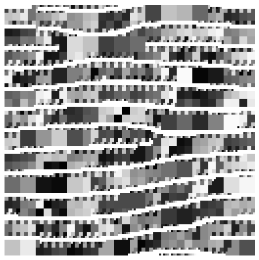
In order to inform our analyses in subsequent tests we visually show how the void key-ranges are distributed in various textures depending on the index scheme.

Table A.1 shows the MIP-map level with resolution 512×512 for various sparse textures. These images also show the block occupation and key distribution of the textures. White is an occupied block and shades of gray denote key-ranges.

The graphs in Table A.2 show the number of void key-ranges and the distribution of the lengths of them in the given textures.

Table A.1: Pictures showing how key-ranges are shaped in a sparse texture depending on the index scheme chosen. The texture is blocked into 4×4 texel blocks.

Texture Name	Index Scheme		
	h	z	r
cat			
cellorig			

Texture Name	Index Scheme		
	h	z	r
check			
funny3			
galaxies-bw			
sine11-1			

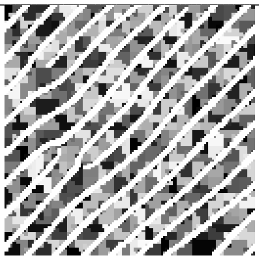
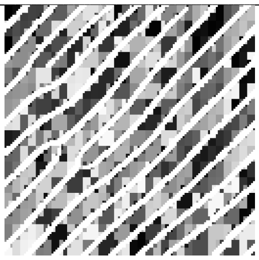
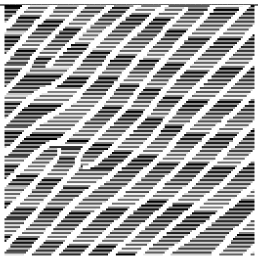
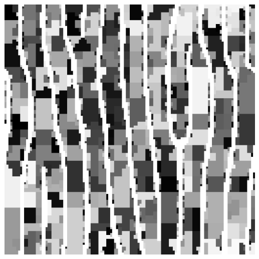
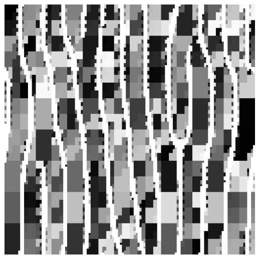
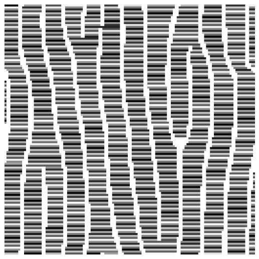
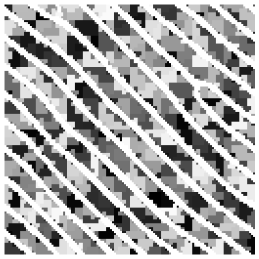
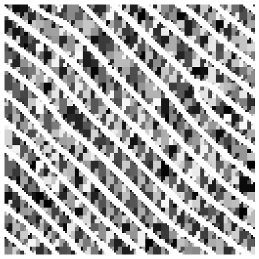
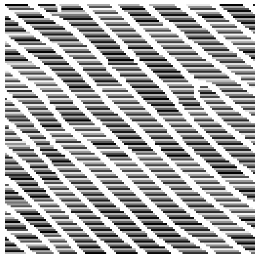
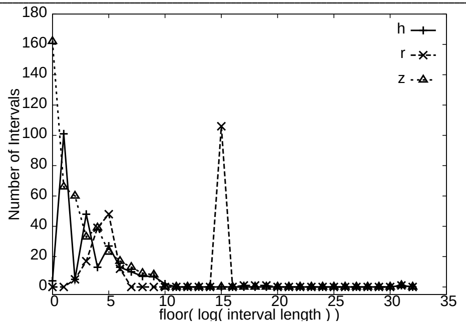
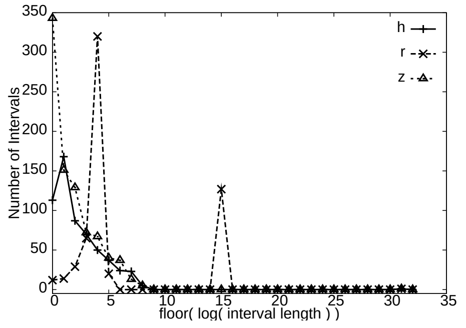
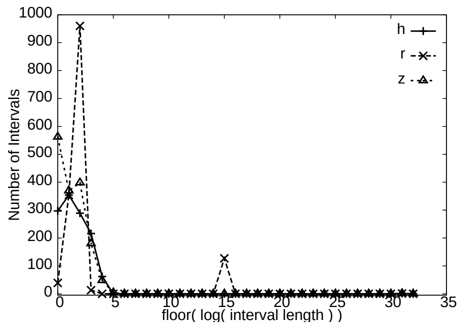
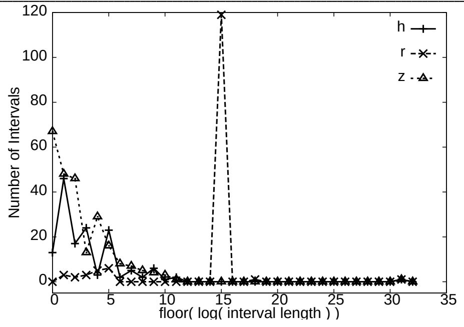
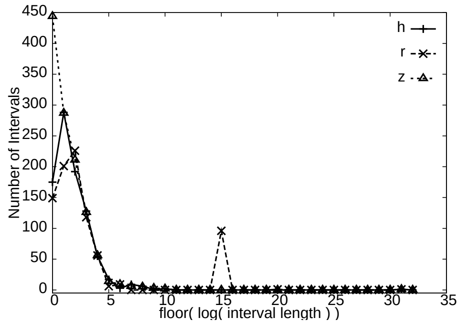
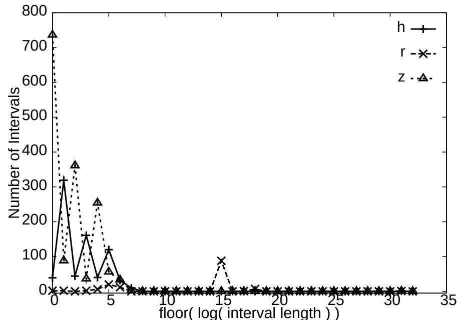
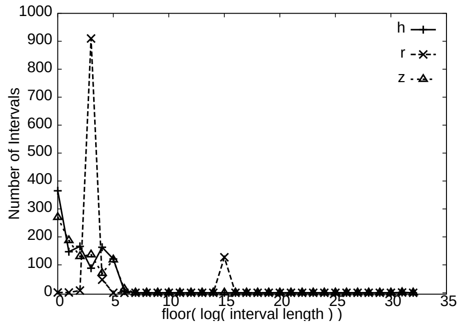
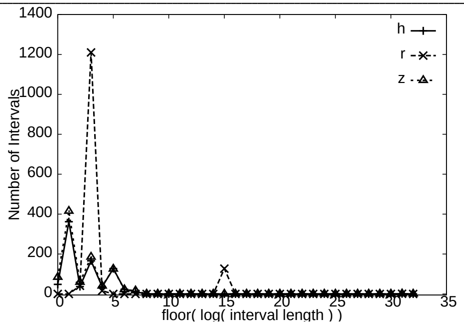
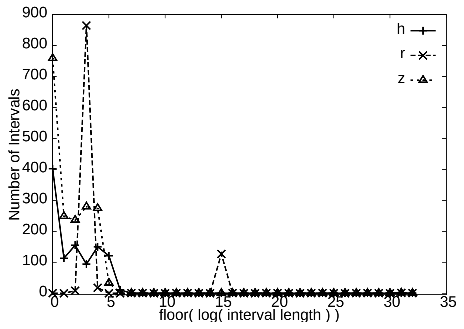
Texture Name	Index Scheme		
	h	z	r
sine11-45			
sine11-91			
sine11-135			

Table A.2: List of graphs showing the distribution of interval lengths for index schemes r, z, and h in a sparse texture.

Texture	Graph
cat	
cellorig	
check	

Texture	Graph
funny3	
galaxies-bw	
sine11-1	
sine11-45	

Texture	Graph
sine11-91	
sine11-135	

Discussion

Looking at the pictures in Table A.1, we see that index scheme r only works well when there are large open horizontal spaces, as in the textures *cat*, *sine11-1*, and the bottom of *galaxies-bw*. The presence of any features that extend vertically result in many shorter 1-block high intervals; this means that the orientation of the features in the texture is also an issue for good compression. The graphs in Table A.2 show that index scheme r produces void key-intervals that pack well into the texture, as there are few short key-intervals.

The pictures and graphs in the two tables show that index schemes z and h produce similar results. They both pack the void parts of a texture with roughly square "blocks", whose size is bounded by the smallest dimension of the space the "block" is in. The graphs of number of intervals versus interval length for z and h have a rough decay shape, indicating that there are many shorter intervals and a few larger ones. From the pictures for textures *sine11-1*, *sine11-91*, *sine11-45*, and *sine11-135*, we see that feature orientation in the texture does not affect the key-intervals produced to the extent that they affect the key-intervals for r-order, though there are some noticeable visual differences for z-order. Also, unlike index schemes r and h, index scheme z does not pack well, requiring many unit-length intervals clustered around the occupied blocks.

A.2.2 Rasterization Order

We first run a test to determine the relationship between rasterization order and index scheme in how they affect the miss rate curve. The test parameters are:

scenes: trifs-0

textures: cat, check, cellorig, funny3, sine11-1, sine11-45, sine11-91

rasterization order: R, Z, H

index schemes: r, z, h

sampling: bilinear sampling from the nearest MIP-map level – the statistical distribution of the key intervals is different for each MIP-map level, so we restrict the sampling to one MIP-map level to simplify the analysis

block size: 64

cache design:

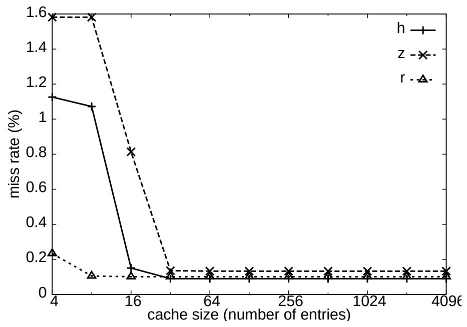
- **cache 1:** A range-cache used to store key-intervals for void-ranges ($[k_1, k_2]$).
  - **eviction policy:** LRU
- **cache 2:** A cache used to store texture blocks that never generates a miss.

The results for the texture *cellorig* are in Figure A.6. The graphs for the other textures are similarly shaped.

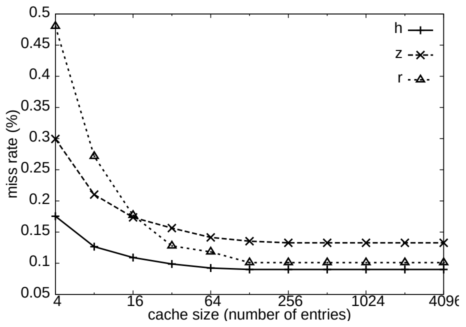
Discussion

Identically to the case of dense textures, the curves for rasterization order R are different from the curves for rasterization orders Z and H, with R having a distinct working set size while Z and H have a smooth decline as cache size increases. These shapes are not affected by the index scheme. The exception to this is rasterization order R with index scheme r; this curve appears very flat because the long thin key-intervals being aligned with the direction of rasterization results in extremely

(a)



(b)



(c)

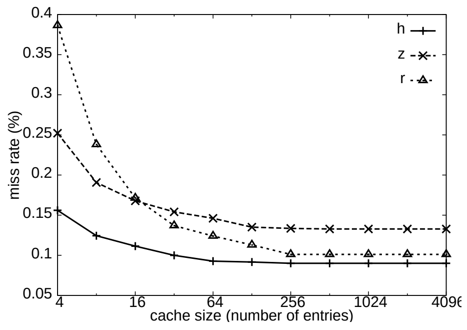


Figure A.6: Graphs of the miss rate curve for the texture *cellorig* for rasterization orders (a) R, (b) Z, and (c) H. The curves for the different index schemes are placed in the same graph.

low miss rates.

The curves do not look as theoretically "perfect" as the curves obtained in previous tests. This is explained by the irregular shapes and sizes of the void key-intervals and their irregular placement in the texture. Before we note anything else we run some tests to see how rotation affects the results.

A.2.3 Effects of Rotation

The next test is to rotate the scene to see how the orientation of the texture relative to the rasterization order affects the miss rate curve. The test parameters are:

scenes: trifs-0, trifs-45, trifs-90

textures: cat, check, cellorig, funny3, sine11-1, sine11-45, sine11-91

rasterization order: R, Z, H

index schemes: r, z, h

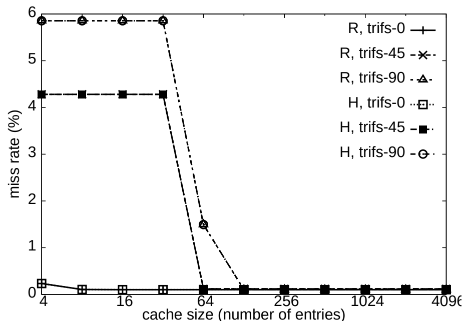
sampling: bilinear sampling from the nearest MIP-map level – the statistical distribution of the key intervals is different for each MIP-map level, so we restrict the sampling to one MIP-map level to simplify the analysis

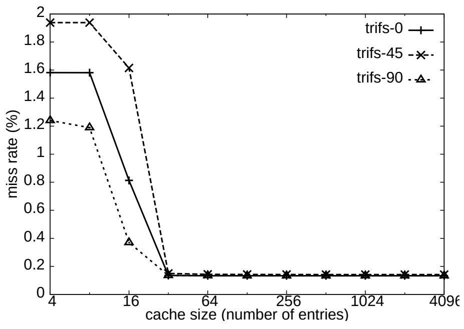
block size: 64

cache design:

> **cache 1:** A range-cache used to store key-intervals for void-ranges ($[k_1, k_2]$).
>
> > **eviction policy:** LRU
>
> **cache 2:** A cache used to store texture blocks that never generates a miss.

Some results for texture *cellorig* are in Figures A.7 to A.9.

(a)


(b)


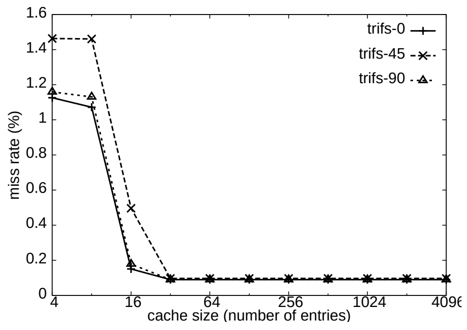
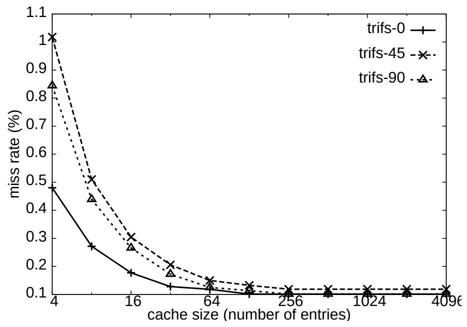
(c)


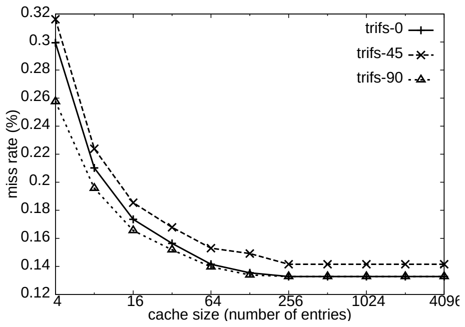
Figure A.7: Miss rate curves for various scenes for texture *cellorig* with rasterization order R and index schemes (a) r, (b) z, and (c) h.

Discussion

We find that with a couple of exceptions, rotation of the scene produced the same effects in this test as in the corresponding test for dense textures *i.e.* the miss rate curve for scenes *trifs-0* and *trifs-90* is about the same, while that for *trifs-45* is somewhat higher and has a larger working set. We can divide the explanation into the following cases:

- In the case where the index scheme is z or h the miss rate curve behaviour under texture rotation is expected due to the block-like shapes of the void key-intervals, reducing the situation to a variation of the dense texture case.

(a) 

(b) 

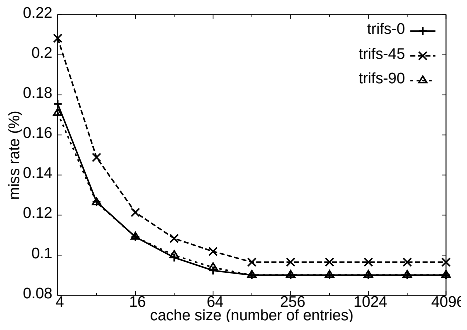
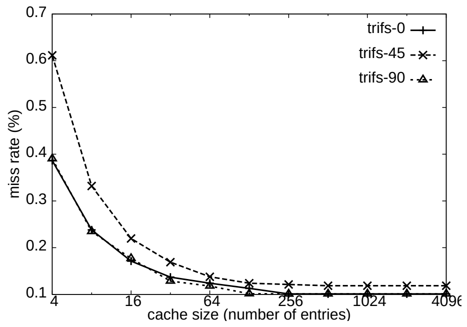
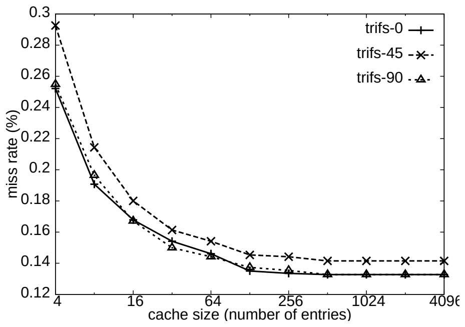
(c) 

Figure A.8: Miss rate curves for various scenes for texture *cellorig* with rasterization order Z and index schemes (a) r, (b) z, and (c) h.

Note that the increase in the cold miss rate for scene trifs-45 for the given graphs for *cellorig* is due to features popping in and out of the scene as it rotates; scene trifs-45 still has the steepest non-cold miss rate curve of the three scenes.

- For the case where the index scheme is r and the rasterization order is Z or H, the explanation is depicted in Figure A.13. Basically the number of void key-intervals necessary to cover a square block of pixels peaks at the 45-degree angle.

(a) 

(b) 

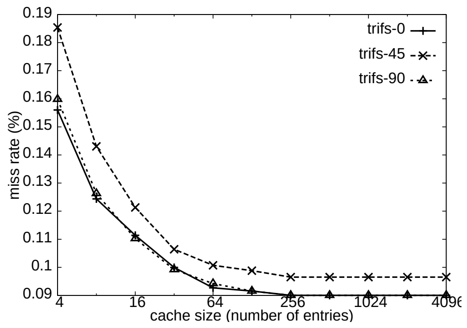
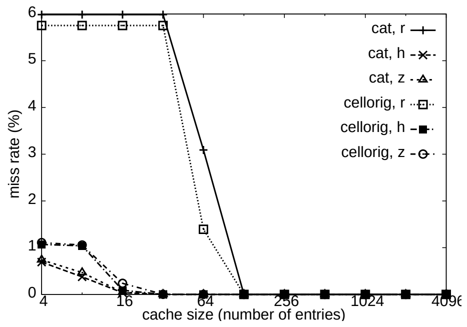
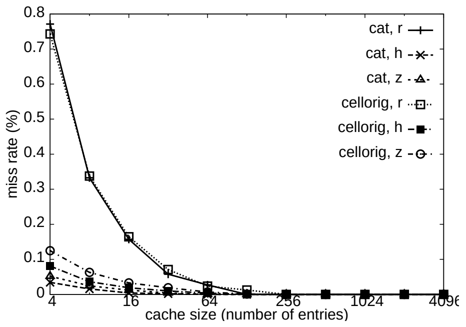
(c) 

Figure A.9: Miss rate curves for various scenes for texture *cellorig* with rasterization order H and index schemes (a) r, (b) z, and (c) h.

- The first exception to the miss rate peaking near the 45-degree angle is when the textures have very few void key-intervals showing such as *funny3*, so that the number of void key-intervals is so low and the texture so irregular that "sampling artifacts" become an issue.

- The second exception to the rule and remaining case of rasterization order R, index scheme r, has the working set size and miss rate peaking at the 90-degree angle. Also, unlike the other cases there is a marked performance disparity between the 0-degree angle curve and the 90-degree angle curve. Similarly to the second case, this is accounted for in Figure A.14.

(a)


(b)


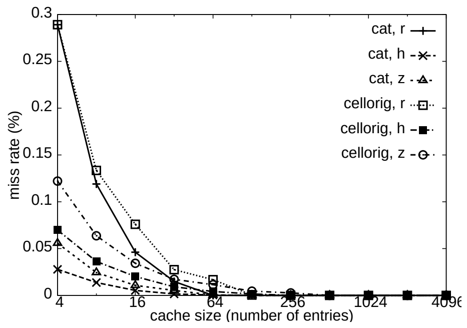
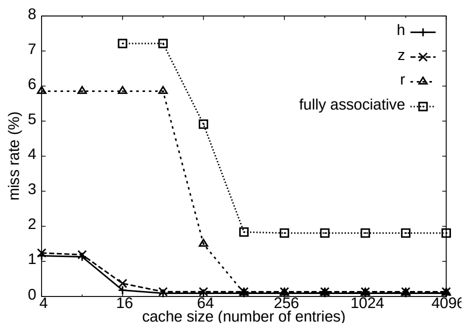
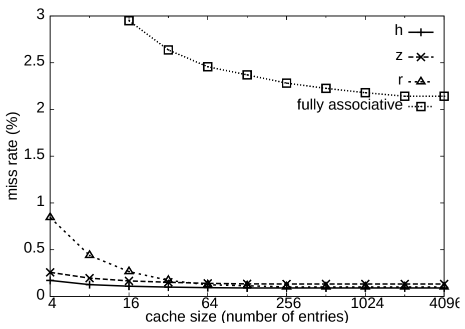
(c)


Figure A.10: Non-cold miss rate curves for scene *trifs-90* and textures *cellorig* and *cat* and various index schemes. The rasterization order is (a) R, (b) Z, and (c) H.

Regardless of texture orientation, if we consider the curves obtained when the directions of rasterization order R and index scheme r are aligned to be the tail end of a step curve, the basic curve shape for each rasterization order stays the same independently of the texture or index scheme chosen.

We also find that using a void range cache instead of a void block-based cache results in comparatively low miss rates and working set sizes for the sparser textures (see Figure A.11) . The exception is the combination of rasterization order R and index scheme r, which only produces comparatively low miss rates and working set sizes if the void key-intervals and the direction of rasterization are aligned.

(a)


(b)


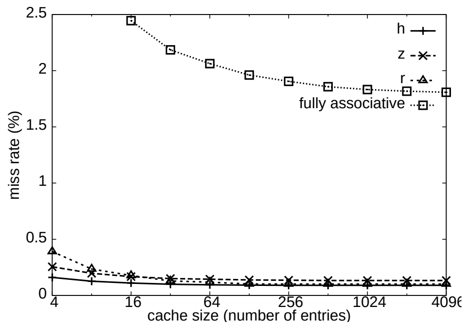
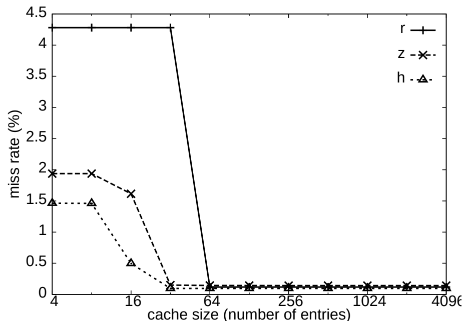
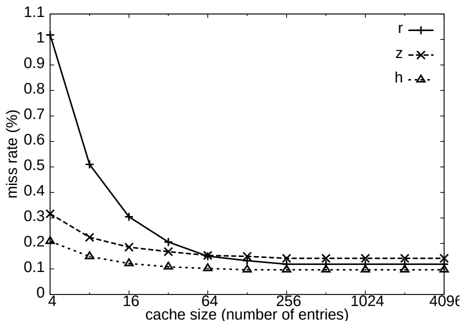
(c)


Figure A.11: Graphs comparing the miss rate for the different index schemes when using a range-cache to the miss rate when using a fully associative cache for rasterization orders (a) R, (b) Z, and (c) H. The scene is trifs-90, and the texture *cellorig*.

However, the reduction in working set size and flattening of the rate of growth as the cache size shrinks does not necessarily increase as the texture gets sparser. Figure A.10, which shows the non-cold miss rate curves for texture *cellorig* and the sparser texture *cat* in scene *trifs-90*, show that when index scheme r is used, the working set size for R order and the rate of growth as the cache size decreases for Z and H orders stay much the same between the two textures. This is not the case with index schemes z and h, which have the non-cold miss rate curves get flatter as the texture gets sparser. The explanation for this is in the pictures in Table A.1.

(a)


(b)


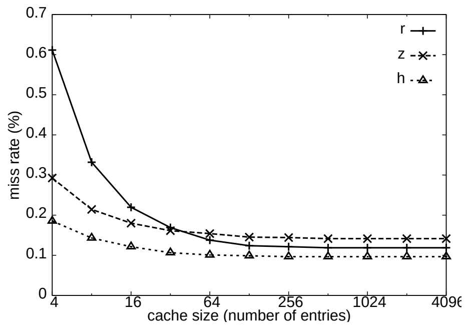
(c)


Figure A.12: Graphs comparing the miss rate for the different index schemes for rasterization order (a) R, (b) Z, and (c) H. The scene is trifs-45, and the texture *cellorig*.

We get fewer larger key-intervals for the sparser texture when index schemes z or h are used, while we still get many 1-block high void key-intervals for both textures independent of sparsity when index scheme r is used.

As for the effects of the index scheme when the texture and scene is fixed, they can be described in terms of the miss rate at small cache sizes and large cache sizes (see Figure A.12). The miss rate for the largest cache sizes is determined by the cold miss rate, which means that for these cache sizes index scheme z produces the highest miss rate for a given rasterization order due to poor packing of the void key-intervals. The miss rate at the smallest cache sizes for rasterization orders Z and H

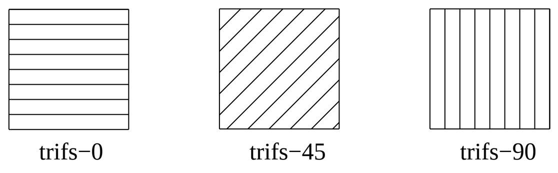


Figure A.13: Simplified graphics representing how void key-intervals for index scheme r cover a square block of pixels for the various scenes.

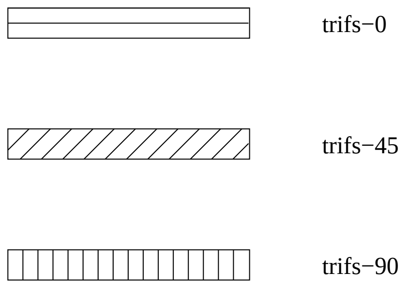


Figure A.14: Simplified graphics representing how void key-intervals for index scheme r cover a row of pixels for the various scenes.

increase as we go from index scheme h, to z, to r, corresponding to the increasing number of void key-intervals on average needed to cover a $2^n \times 2^n$ block of pixels, though the differences are so slight as to be insignificant. For rasterization order R the ordering is similar for similar reasons except that the texture orientation and the texture itself will move the placing of r in the list around, though for the worst case where the scene is trifs-90 r will generally produce a larger working set.

A.3 Shader Tests

The tests in this section deal with the effects of unusual patterns of texel lookups due to a programmable shader and anisotropic sampling.

A.3.1 Test: Effect of Turbulence On A Conventional Setup

We start off with testing the effect of turbulence on dense textures. The test parameters are:

scenes: trifs-90

textures: dense

rasterization order: R, Z, H

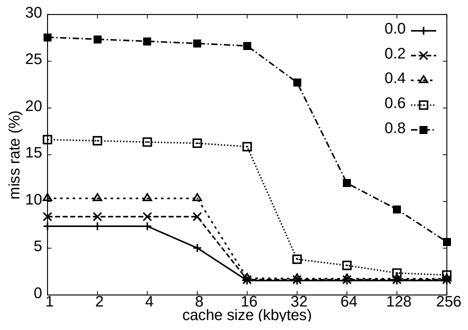
index schemes: r (irrelevant for a fully associative cache)

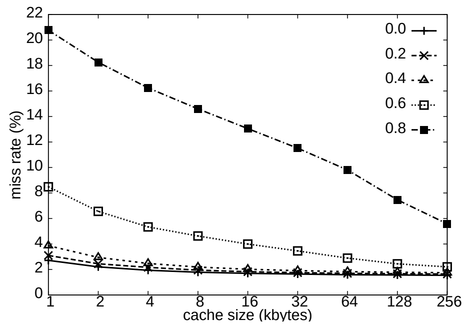
sampling: trilinear

shader parameters:

> ω**:** 0.0, 0.2, 0.4, 0.8
>
> **shader derivatives:** no

block size: 64

(a)


(b)


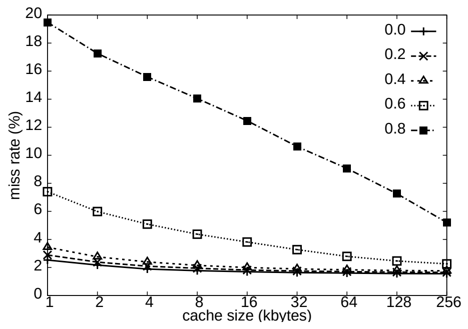
(c)


Figure A.15: Miss rate curves when not using shader derivatives for rasterization orders (a) R, (b) Z, and (c) H.

cache design:

cache 1: One cache for only texture blocks (T blocks) and void blocks (V blocks).

associativity: full

eviction policy: LRU, optimal

The results for block size 64 are in Figure A.15.

Discussion

Generally, we get abysmally bad miss rates for higher degrees of turbulence. Note that we get the same basic shapes of a decay curve for H and Z orders and a step curve for R order regardless of the degree of turbulence, but we also get an increase in cold miss rate for all rasterization orders, an increase in slope for rasterization orders Z and H, and an increase in step height and working set for rasterization order R. The cold miss rate can be explained by looking at Figure 2.13. Since the MIP-map level used to access the texture is constant (flat triangle parallel to view-plane), as ω increases more of the texture is used and hence more texture blocks are used. However, it is unclear how much of the increase in miss rate is due to the turbulence and how much is due to sampling from the wrong MIP-map level.

A.3.2 Test: Effect of Using Shader Derivatives

We next run the same tests but this time the λ_{LOD} is determined by the actual derivatives of the shader function.

scenes: trifs-90

textures: dense

rasterization order: R, Z, H

index schemes: r (irrelevant for a fully associative cache)

sampling: trilinear

shader parameters:

- ω**:** 0.0, 0.2, 0.4, 0.8

- **shader derivatives:** yes

block size: 64

cache design:

- **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).

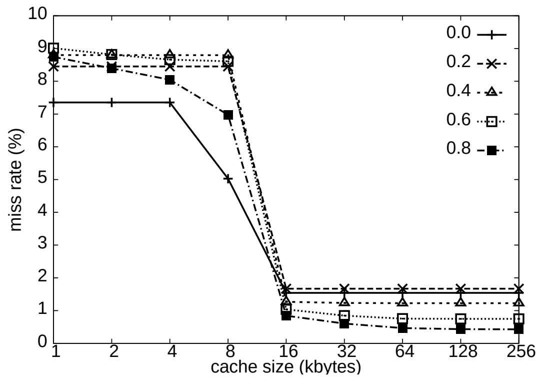
  - **associativity:** full

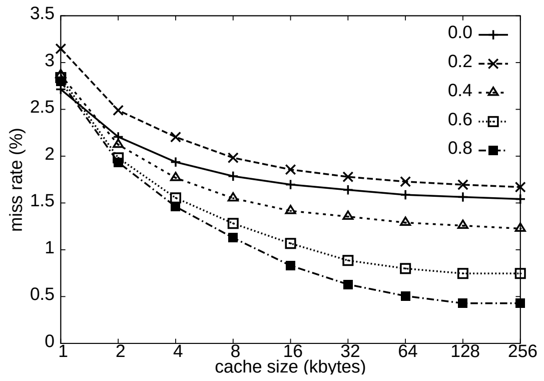
  - **eviction policy:** LRU, optimal

The results for block size 64 are in Figure A.16.

Discussion

The primary thing to note is that the miss rate curves here are very close together for all rasterization orders, indicating that the main reason for the bad cache behaviour

(a)


(b)


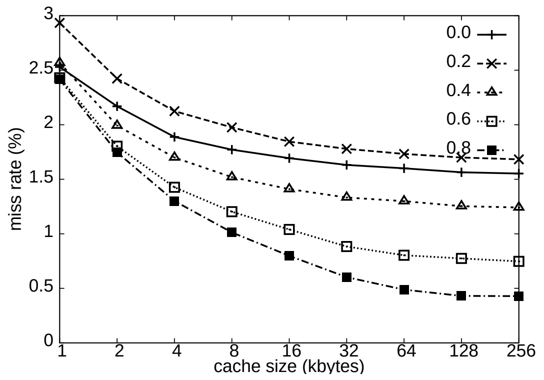
(c)


Figure A.16: Miss rate curves when using shader derivatives for rasterization orders (a) R, (b) Z, and (c) H.

in the previous case is using the wrong MIP-map level for sampling instead of the increase in discontinuities and distortion.

As ω increases the cold miss rate, indicated by the miss rate at large cache sizes, initially increases slightly, then largely decreases, well below the bound established in Figure A.4. Some of the decrease we can attribute to fewer blocks being needed at the large scale due to the average MIP-map level going down resulting in partial "tiling", but some of it can also be attributed to changes in $\lambda - \lfloor \lambda \rfloor$. For rasterization orders Z and H, this translates into a lower miss rate for most cache sizes as ω

goes above 0.2, as the effect of having fewer blocks involved outweighs the effect of having a steeper slope. Note that the steepness of the slopes is close to the variation due to the fractional value of λ_{LOD} changing (see Figures A.4 and A.5).

A.3.3 Test: Block Size

So far we have seen that using shader derivatives when sampling is sufficient to ensure that there is enough locality in the texel accesses to deal with a high degree of turbulence for a block size of 4 texels $\times$ 4 texels. However, using that block size results in a high cold miss rate, and as detailed in the chapter on the test setup a small block size results in a high index overhead for the B-tree sparse texture implementation for both index memory overhead and tree height.

We run tests to see if there is sufficient locality to take advantage of the inherent prefetching in a larger block size, so that the miss rate can be reduced without incurring too much additional bandwidth usage. The test parameters are:

scenes: trifs-90

textures: dense

rasterization order: R, Z, H

index schemes: r (irrelevant for a fully associative cache)

sampling: trilinear

shader parameters:

- **ω:** 0.0, 0.2, 0.4, 0.8
- **shader derivatives:** yes

block size: 64, 256, 1024

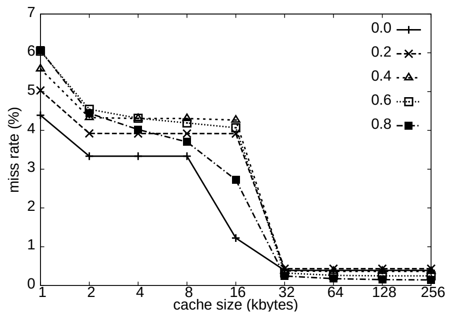
cache design:

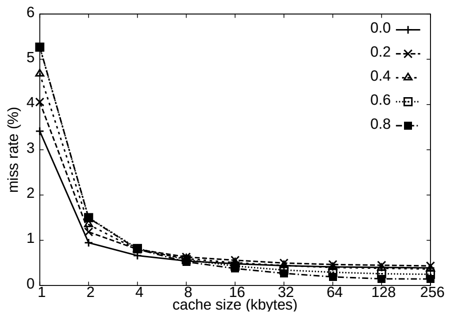
- **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).
  - **associativity:** full
  - **eviction policy:** LRU, optimal

Graphs showing curves for various values of ω while the rasterization order and block size are fixed are in Figures A.16, A.17, and A.18. Some results showing both miss rate and bandwidth consumption with curves for different block sizes in the same graph are in Figures A.19 to A.24.

Discussion

From the miss rate graphs in Figures A.17, and A.18, we see that similarly to the test for block size 64, the miss rate curves for the different degrees of turbulence are

(a)


(b)


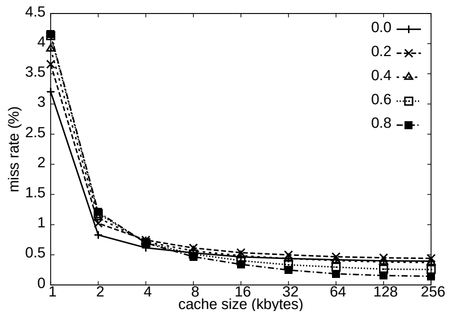
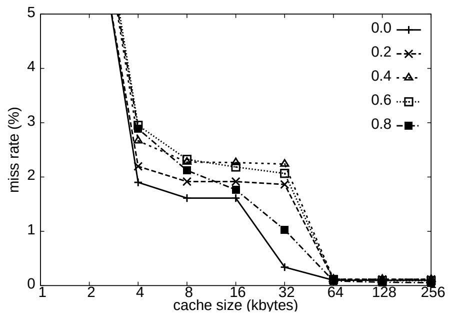
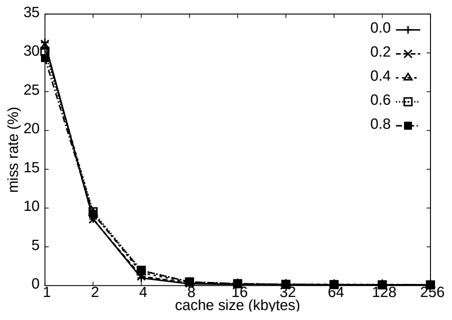
(c)


Figure A.17: Miss rate curves for varying values of ω for block size 256 and rasterization orders (a) R, (b) Z, and (c) H.

(a)


(b)


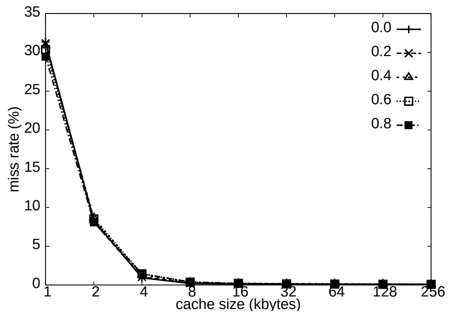
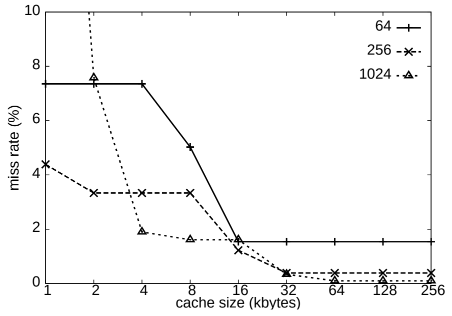
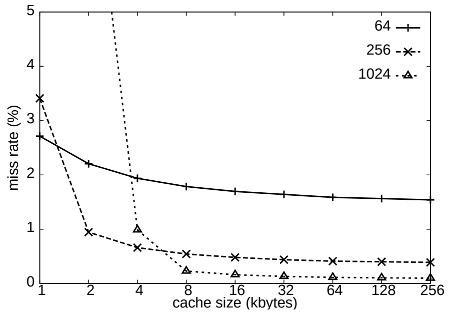
(c)


Figure A.18: Miss rate curves for varying values of ω for block size 1024 and rasterization orders (a) R, (b) Z, and (c) H.

(a)


(b)


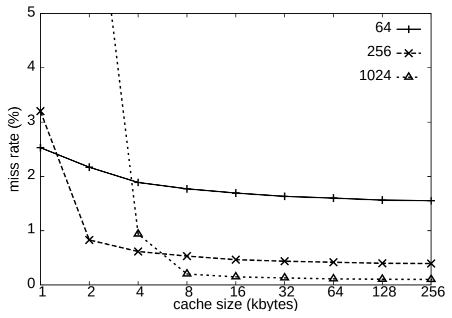
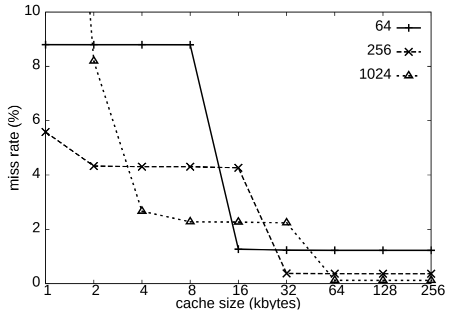
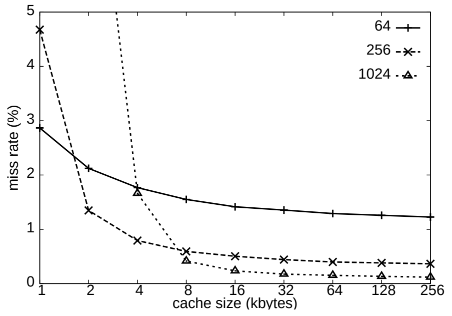
(c)


Figure A.19: The miss rate for varying block sizes for $\omega = 0.0$ for rasterization orders (a) R, (b) Z, and (c) H.

(a)


(b)


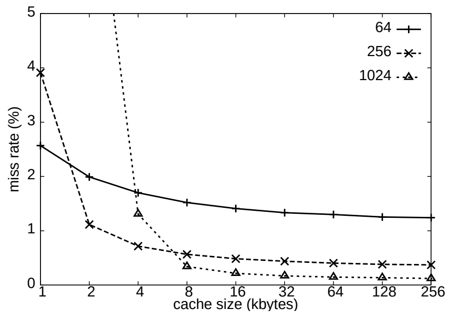
(c)


Figure A.20: The miss rate for varying block sizes for $\omega = 0.4$ for rasterization orders (a) R, (b) Z, and (c) H.

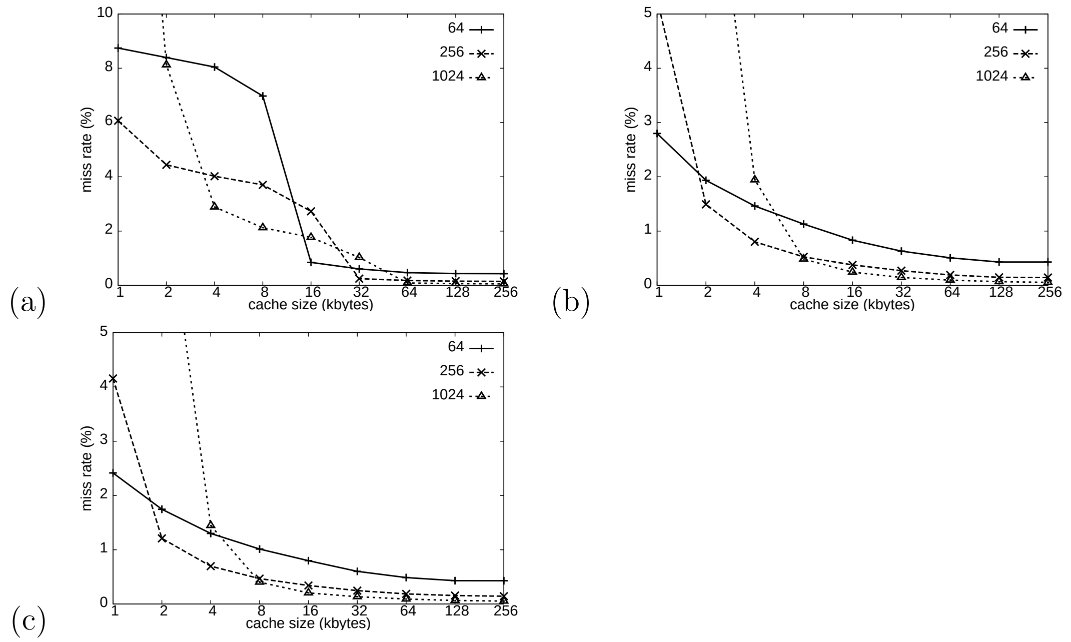


Figure A.21: The miss rate for varying block sizes for $\omega = 0.8$ for rasterization orders (a) R, (b) Z, and (c) H.

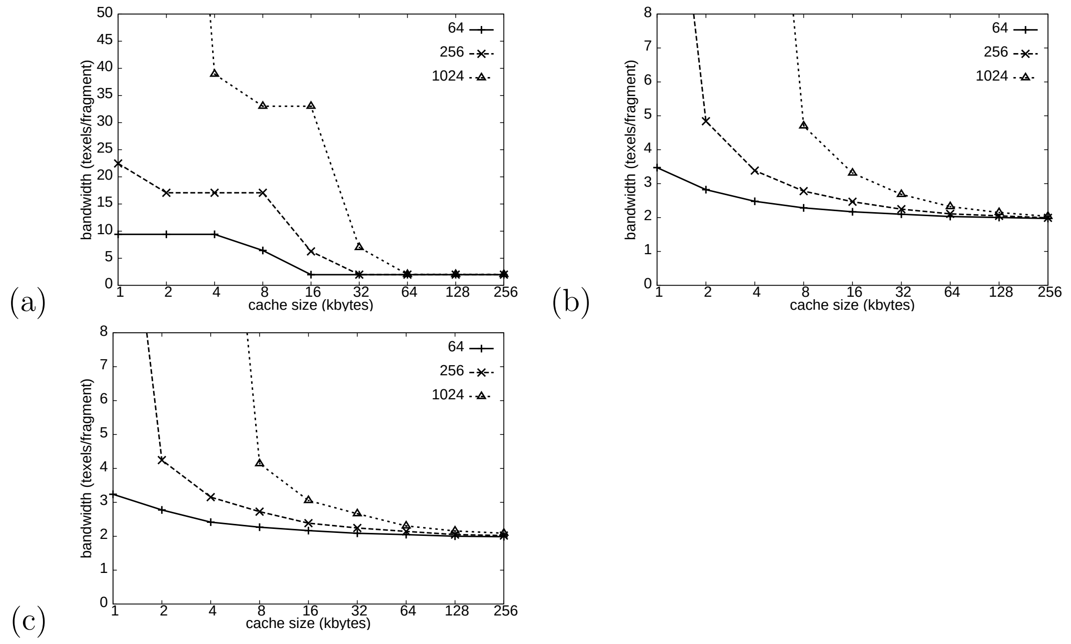


Figure A.22: The bandwidth consumption for varying block sizes for $\omega = 0.0$ for rasterization orders (a) R, (b) Z, and (c) H.

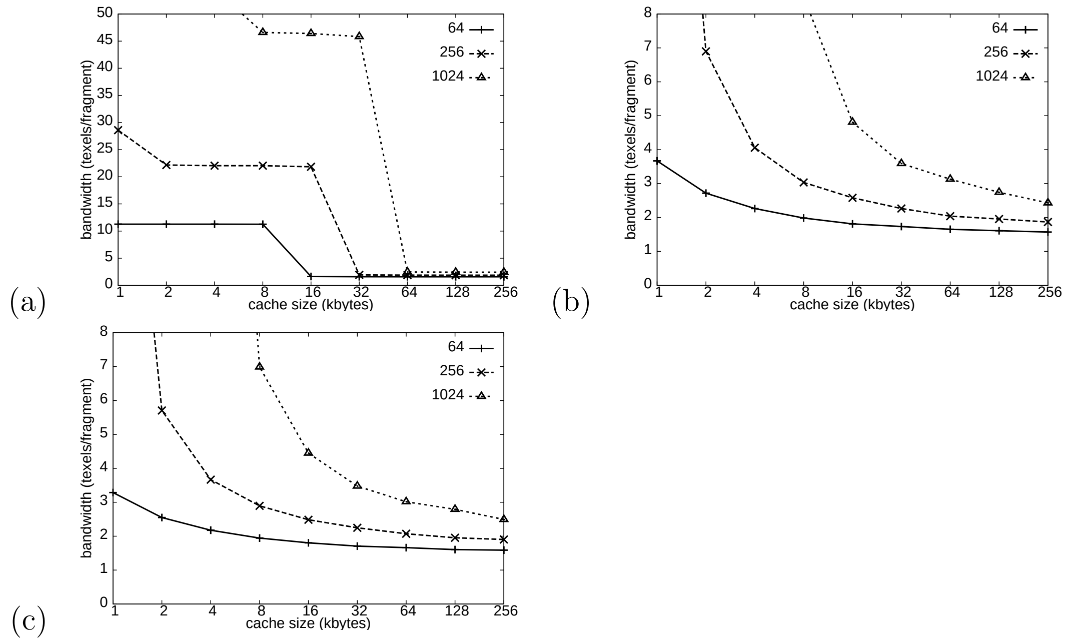


Figure A.23: The bandwidth consumption for varying block sizes for $\omega = 0.4$ for rasterization orders (a) R, (b) Z, and (c) H.

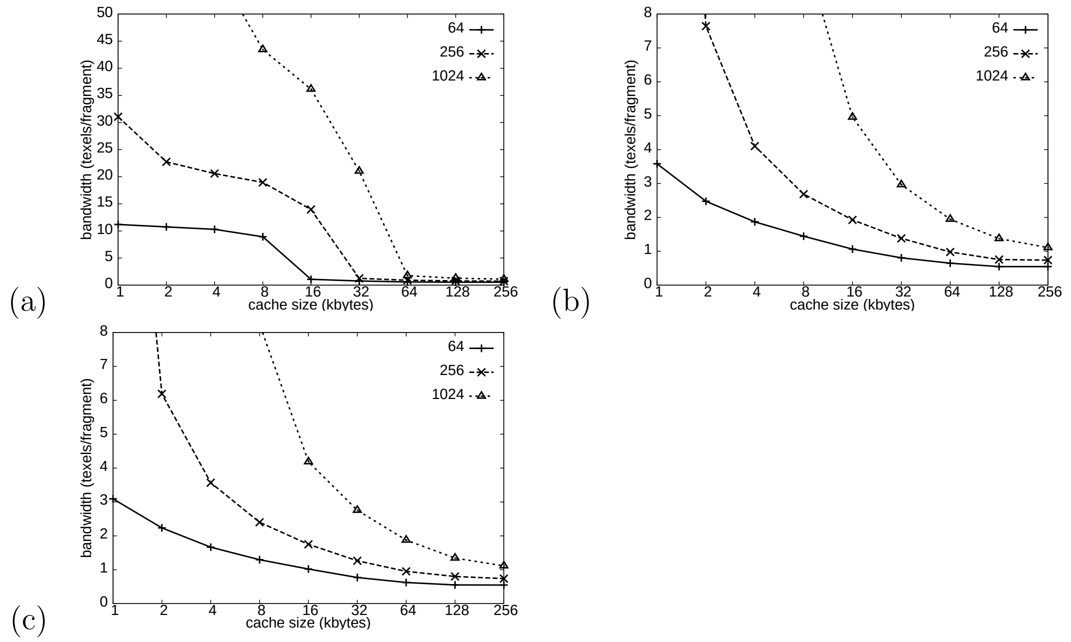


Figure A.24: The bandwidth consumption for varying block sizes for $\omega = 0.8$ for rasterization orders (a) R, (b) Z, and (c) H.

close together, indicating that there is enough spatial locality so that turbulence does not significantly affect the miss rate for those block sizes.

As expected, increasing the block size decreases the cold miss rate and the miss rate at very large cache sizes (Figures A.19 to A.21). However, looking at the miss rate curves for $\omega = 0.0$ more closely, we see something occur around cache sizes 2K and 8K for block sizes 256 and 1024 respectively. For rasterization order R, we get the end of a step-down. For rasterization orders Z and H, the miss rate curve suddenly becomes much steeper as the cache size drops below the threshold. The bandwidth consumption curves (Figures A.22 to A.24) are similarly affected as they are the miss rate curves scaled by the corresponding block size. The given thresholds also mark the cache size below which the advantages of a lower cold miss rate from the next larger block size is outweighed by the increase in curve steepness for rasterization orders Z and H.

This behaviour is explained by the given thresholds (2K, 8K) being the cache size needed to hold 8 texture blocks. This is the maximum number of blocks needed by a trilinear probe. However few trilinear probes require 8 blocks; many more use 4, and the vast majority of the probes use only 2. This accounts for the curves for block size 1024 and $\omega = 0.0$, where the increase in miss rate as the cache size goes from 4 blocks to 2 blocks is much greater than the increase as the cache size goes from 8 to 4 blocks.

The bandwidth consumption curves are in Figures A.22 to A.24. As expected from more memory being needed to cover the texture used, the working set size (we ignore the first step-down) for rasterization order R doubles with each increase in block size. The step height for the curve also increases dramatically. When the rasterization order is Z or H, going from block size 64 to 256 results in modest increases in bandwidth consumption when the cache size is at least 2K; the increase from going from block size 256 to 1024 is large in comparison. The additional bandwidth overhead from going to block size 1024 also increases quite a bit as ω increases.

To summarize, there is sufficient locality so that the miss rate curves belonging to larger block sizes are not significantly affected by turbulence. However, when using trilinear sampling, it is necessary for the cache size to be large enough to hold 8 texture blocks to ensure that increasing the block size improves the miss rate for rasterization orders Z and H. In order to do the same for rasterization order R the cache must be large enough to contain the working set size, which in our test cases roughly doubles for each increase in block size. There is also sufficient locality to use a block size of 256 to reduce the miss rate with minor increases in the bandwidth consumption. There is insufficient locality to take advantage of the greater inherent prefetching in a block size of 1024.

A.3.4 Test: Effect of Turbulence on Void Range-Cache

Now we see the effect of turbulence on the miss rate for a void range-cache. We use the following test parameters:

scenes: trifs-90 – we only use the scene of the set of three that produces worst case behaviour for rasterization order R

textures: cat, funny3, cellorig, sine11-91

rasterization order: R, Z, H

index schemes: r, z, h

sampling: trilinear

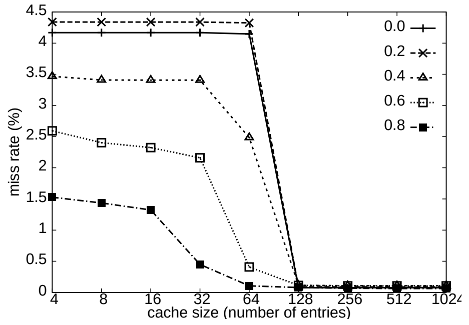
shader parameters:

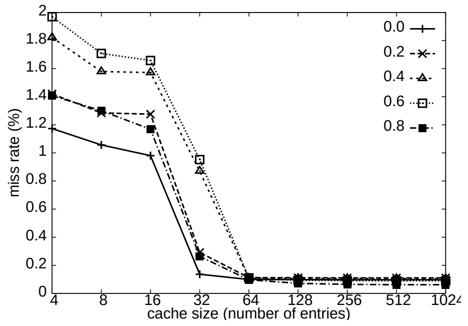
- **ω:** 0.0, 0.2, 0.4, 0.6, 0.8
- **shader derivatives:** yes

block size: 64

cache design:

- **cache 1:** A range-cache used to store key-intervals for void-ranges ($[k_1, k_2]$).
  - **eviction policy:** LRU

(a)


(b)


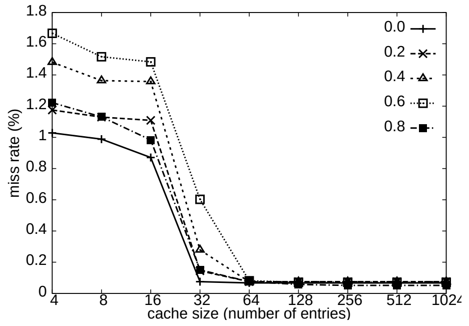
(c)


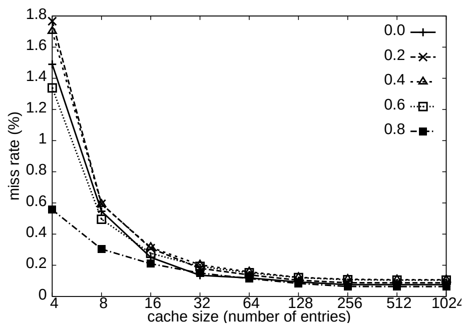
Figure A.25: Miss rate curves for texture *cellorig*, block size 64, and rasterization order R for index schemes (a) r, (b) z, and (c) h.

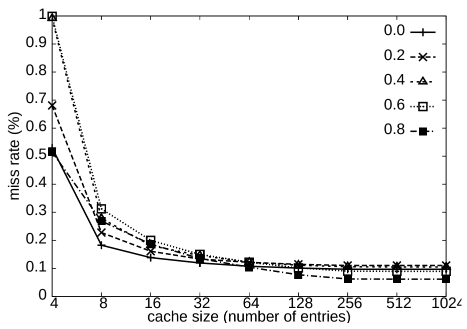
cache 2: A cache used to store texture blocks that never generates a miss.

The results for texture *cellorig* are shown in Figures A.25 to A.27. Similar results were obtained for the other textures.

Discussion

The miss rate curves for the various degrees of turbulence for rasterization orders Z and H are pretty close together except at small cache sizes where the curves for $\omega = 0.6$ and $\omega = 0.8$ are somewhat steeper than the others, showing that they are only significantly affected by turbulence at the small scale (we ignore the parts

(a)


(b)


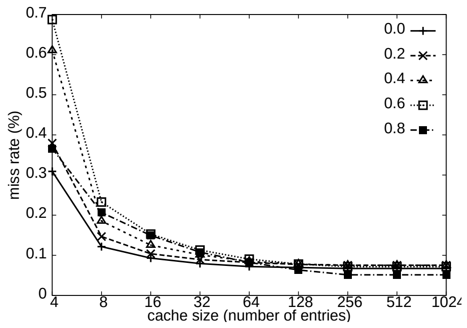
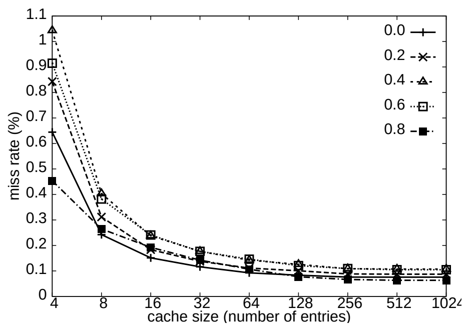
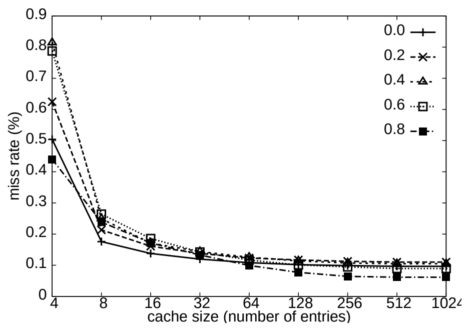
(c)


Figure A.26: Miss rate curves for texture *cellorig*, block size 64, and rasterization order Z for index schemes (a) r, (b) z, and (c) h.

of the curve with 8 or fewer cache entries). Part of the steepness increase can be attributed to the poorer spatial locality and changes in $\lambda - \lfloor\lambda\rfloor$, but it should be noted that as turbulence increases the MIP-map level resolutions accessed decreases, changing the shapes and number of void key-intervals involved in rendering in a way dependent on the texture. The dependency of the changes on the texture results in variances in the change in steepness of the curve and the number of cold misses as ω increases.

Turbulence has a much greater effect on the miss rate curves for rasterization order R. The effect of turbulence on the working set size and step height using

(a)


(b)


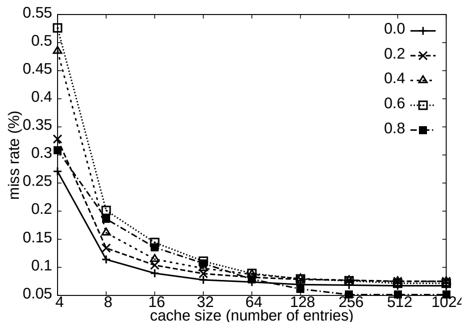
(c)


Figure A.27: Miss rate curves for texture *cellorig*, block size 64, and rasterization order H for index schemes (a) r, (b) z, and (c) h.

index schemes z and h varies with the texture, but when index scheme r is used we get a large decrease in the step height and the working set size with each increase in turbulence, even for small cache sizes.

This is explained by looking at the graph in Figure A.28. It is evident that for $\omega = 0$, the miss rate and working set size is strongly influenced by texture orientation, indicating that it is the primary factor in determining locality in this case. We also see that for $\omega = 0.6$ and $\omega = 0.8$, changing the texture orientation does not alter the curve as much as changing the turbulence does. Note also that the curves are sandwiched between the curve for (scene=trifs-0, $\omega = 0$) where

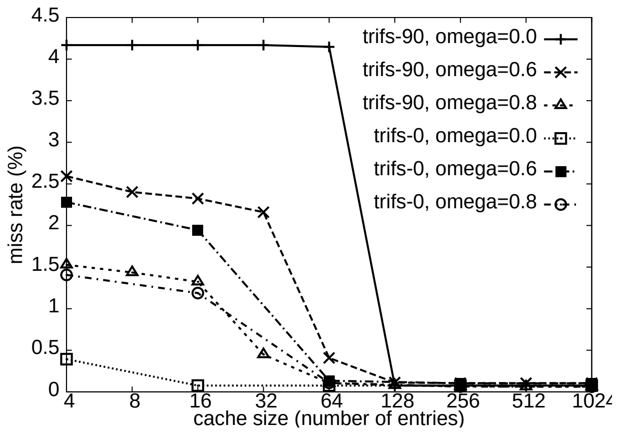


Figure A.28: Graph showing the miss rate curves for various values of ω for the scenes trifs-0 and trifs-90 using rasterization order R and index scheme r.

the key-intervals are aligned with the direction of rasterization and the curve for (scene=trifs-90, $\omega = 0$) where they are not. We conclude that the turbulence improves locality of reference for scene trifs-90 where the key-intervals are already aligned in the worst possible direction, and degrades it for scene trifs-0 where they are already aligned in the best possible direction.

In summary, turbulence does not much affect the performance of a void range-cache except in the case when rasterization order R and index scheme r is used, where due to the extreme anistropy of both the rasterization order and the void key-intervals the perturbations in the alignment between the two due to turbulence strongly affect the locality of the key-interval requests.

A.3.5 Test: Effect of FELINE Sampling on a Block-Based Setup

Here we see the effect of using table FELINE sampling on the miss rate curves. The test parameters are as follows:

scenes: trifs-90

textures: dense

rasterization order: R, H – we do not test Z because it is so similar to H.

index schemes: r (irrelevant for a fully associative cache)

sampling: table FELINE

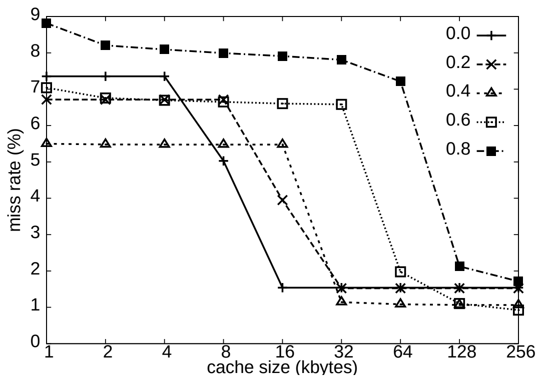
shader parameters:

> **ω:** 0.0, 0.2, 0.4, 0.6, 0.8
>
> **shader derivatives:** yes

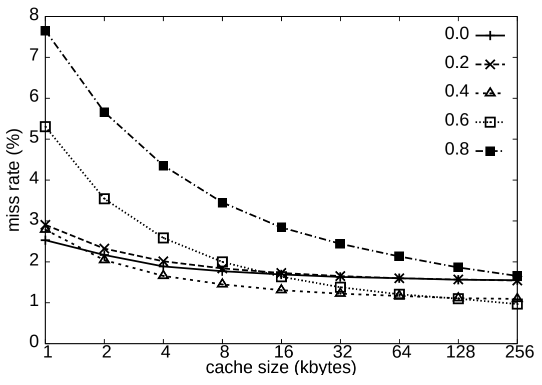
block size: 64, 256, 1024

cache design:

> **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).



(a) R



(b) H

Figure A.29: Miss rate curves when using table FELINE sampling, block size 64, LRU eviction policy, and various values of ω for rasterization orders R and H.

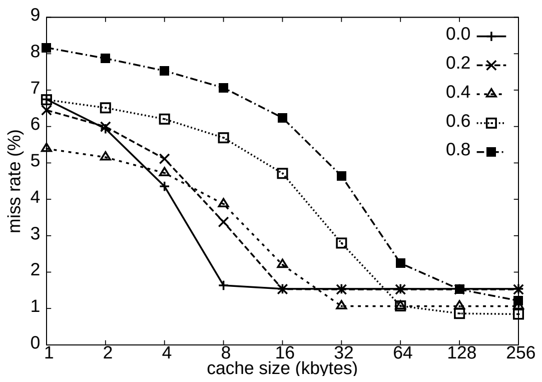
associativity: full

eviction policy: LRU, optimal

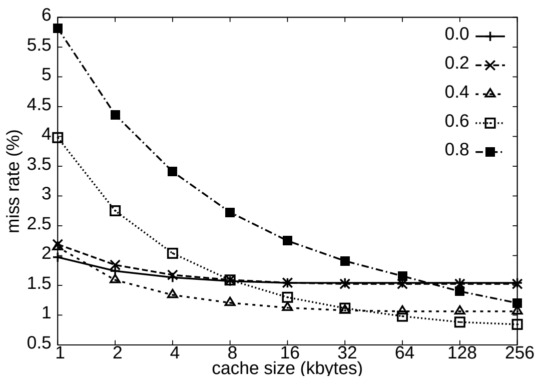
Some results for block sizes 64 and 256 are in Figures A.29 to A.32. We got similar results for block size 1024.

Discussion

We first note that the changes in the cold miss rate for the various values of ω is within the range expected due to fractional changes in the value of λ used to select the MIP-map level to probe. Subsequent discussion will ignore the cold miss rate and focus on the working set size for rasterization order R and the steepness of the



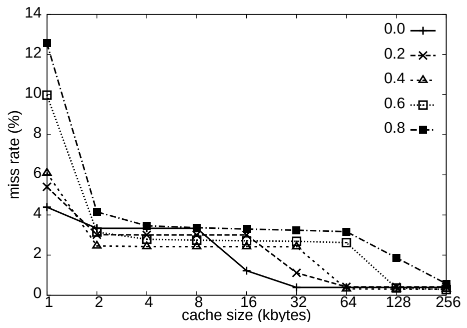
(a) R



(b) H

Figure A.30: Miss rate curves when using table FELINE sampling, block size 64, optimal eviction policy, and various values of ω for rasterization orders R and H.

(a)



(b)

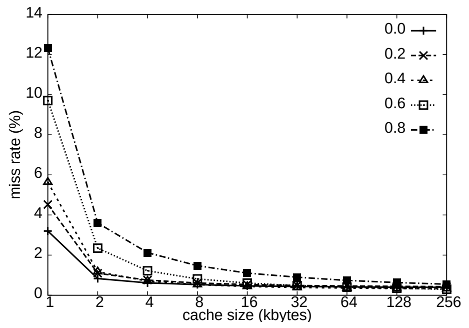


Figure A.31: Miss rate curves when using table FELINE sampling, block size 256, LRU eviction policy, and various values of ω for rasterization orders (a) R and (b) H.

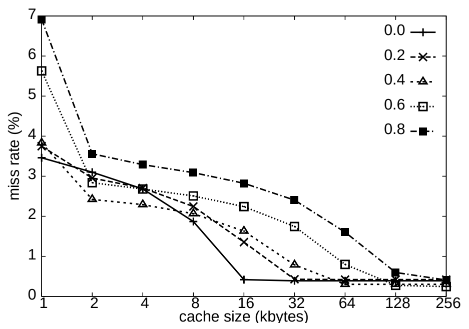
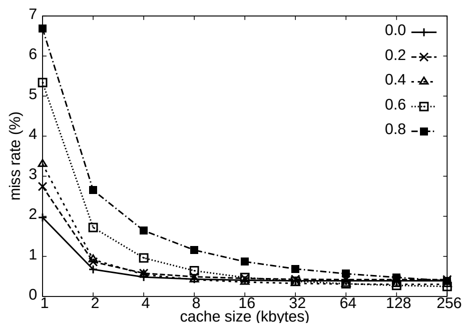
(a)  (b) 

Figure A.32: Miss rate curves when using table FELINE sampling, block size 256, optimal eviction policy, and various values of ω for rasterization orders (a) R and (b) H.

curves for rasterization order H.

We find that for rasterization order R, using table FELINE sampling instead of trilinear sampling results in the working set size doubling with each increase in turbulence, even when the eviction policy is optimal. This is expected as both the path through the texture corresponding to a scanline and the lines we probe along for each texture sample get longer on average as the turbulence increases particularly for the higher values of ω. Since the latter are not usually aligned with the direction of rasterization this means that many more texture blocks are needed to cover a scanline using this sampling method than when using trilinear sampling.

Comparing the graph in Figure A.29(a) to the graph in Figure A.16(a), we see that the miss rate curves for table FELINE sampling have smaller step heights than the corresponding miss rate curves for trilinear sampling, despite having larger working set sizes. This can be explained by the meandering off a straight line caused by the table FELINE sampling actually improving the spatial locality of the texel requests at the small scales.

In the case where we use rasterization order H, there is a steady increase in the steepness of the miss rate curves as the value of ω increases; this increase is slight for values up to $\omega = 0.4$ but is relatively large for higher values.

A.3.6 Test: Effect of FELINE Sampling on Void Range-Cache

Now we see the effect of table FELINE sampling on the miss rate for a void range-cache. The test parameters are the same as the previous test except for the following parameters:

scenes: trifs-90

textures: cat, funny3, cellorig, sine11-91

rasterization order: R, H

index schemes: r, z, h

sampling: table FELINE

shader parameters:

ω: 0.0, 0.2, 0.4, 0.6, 0.8

shader derivatives: yes

block size: 64

cache design:

cache 1: One cache for only texture blocks (T blocks) and void blocks (V blocks).

associativity: full

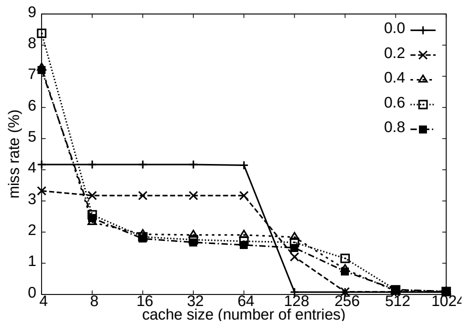
eviction policy: LRU, optimal

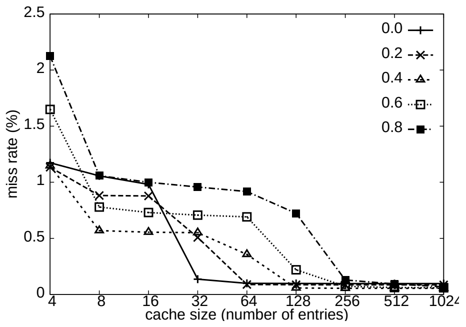
The results for texture *cellorig* are given in Figures A.33 to A.34. Similar results were obtained for the other textures.

Discussion

When rasterization order R is used the working set size increases as ω increases for all the index schemes tested.

The results for rasterization order H (Figure A.34) show that using FELINE sampling produces similar results to when no FELINE sampling is used. The curves for the smallest two or three values of ω tested are close together, and the rest get noticeably steeper as ω increases. Index schemes z and h produced similar miss

(a)


(b)


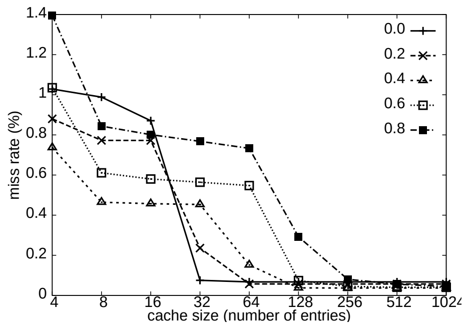
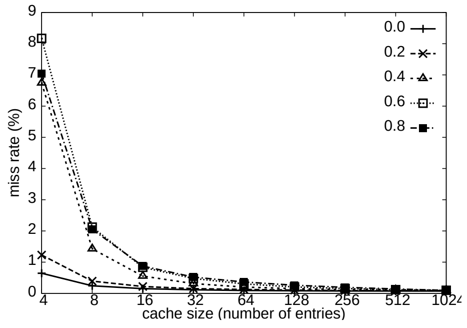
(c)


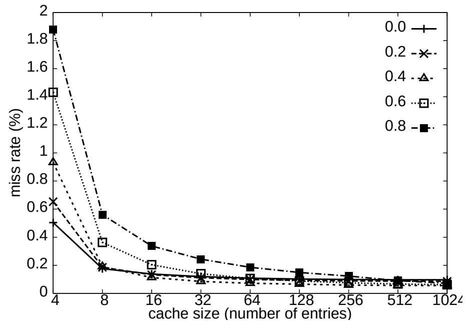
Figure A.33: Miss rate curves when using table FELINE sampling, block size 64, a void range-cache, and rasterization order R as turbulence increases for index schemes (a) r, (b) z, and (c) h. The texture used is *cellorig*, the scene is *trifs-90.*

rates, and index scheme r produced miss rates much higher than those obtained when using the other two index schemes.

A.3.7 Summary

The use of shader derivatives is both necessary and sufficient to ensure that turbulence does not greatly increase the miss rate/bandwidth consumption when trilinear sampling is used. When table FELINE sampling is used, rasterization order R produces an expanding working set size as ω increases while rasterization order H

(a)


(b)


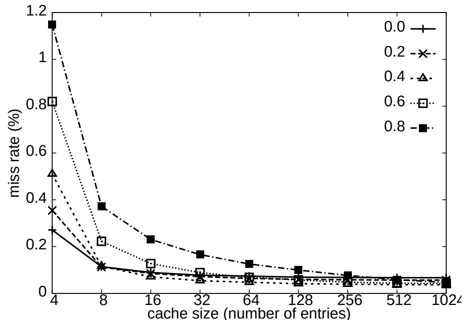
(c)


Figure A.34: Miss rate curves when using table FELINE sampling, block size 64, a void range-cache, and rasterization order H as turbulence increases for index schemes (a) r, (b) z, and (c) h. The texture used is *cellorig*, the scene is *trifs-90*.

produces increases in curve steepness which are large only for high ω values.

We get similar results, albeit with much smaller increases in miss rate and curve steepness, for a void range-cache. Additionally, we find that the texture orientation sensitivity of the rasterization order R and index scheme r combination results in turbulence significantly affecting the locality in key-space of the texel accesses.

A.4 Block-Based Cache Tests

This set of tests establishes a baseline for the miss rate curves for each test scene and the effects of the cache parameters and their interaction with other parameters in a block-based cache.

A.4.1 Test: Miss Rate Curve Shape

Again, we start off with a conventional setup to establish the effect of rasterization order on the miss rate curves for just the texture data *i.e.* we ignore the index. The test parameters are:

scenes: trifs-90, teapot, box, ground, venus

textures: dense

rasterization order: R, H – we do not test Z because it is so similar to H.

index schemes: r (irrelevant for a fully associative cache)

sampling: trilinear

block size: 256

cache design:

cache 1: One cache for only texture blocks (T blocks) and void blocks (V blocks).

associativity: full

eviction policy: LRU

The results are in Figures A.35.

Discussion

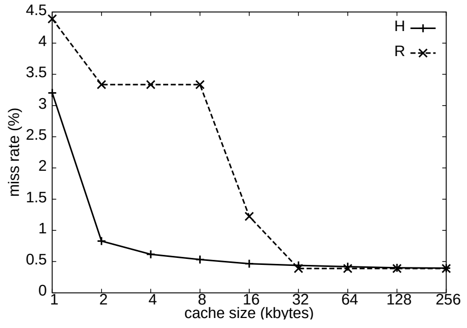
Aside from the scenes box and trifs-90, the general shape of the miss rate curves is primarily determined by the geometry and not the raster order. This confirms previous work [HG97] showing that changing the rasterization order only makes a difference to the locality of the texel requests and hence the miss rate when the triangles are large. Triangle merging when the triangles are small could be used to change this and improve locality if necessary.

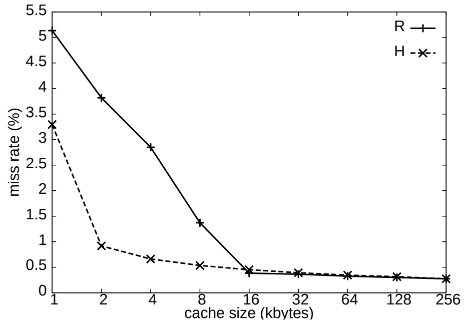
A.4.2 Test: Cache Associativity

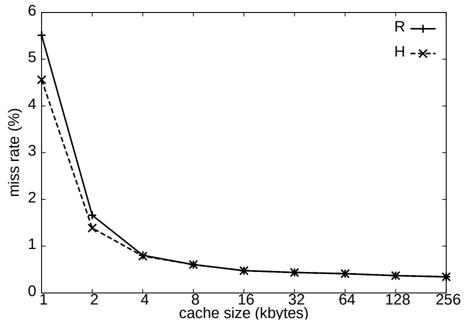
The next test is to check the best associativity needed to avoid conflict misses for the test scenes. The test parameters are:

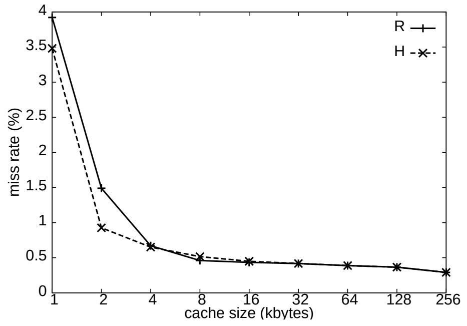
scenes: trifs-90, trifs-0, trifs-45, teapot, box, ground

textures: dense

(a)


(b)


(c)


(d)


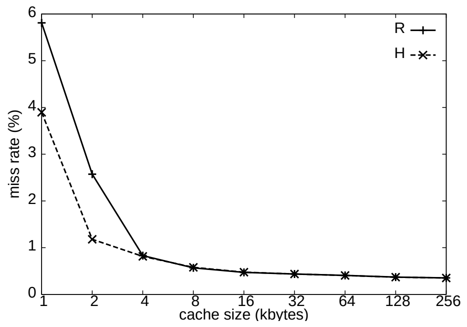
(e)


Figure A.35: Miss rate curves for scenes (a) trifs90 (b) box (c) teapot (d) kground (e) venus.

rasterization order: R, Z, H

index schemes: r, z, h

sampling: trilinear

block size: 256

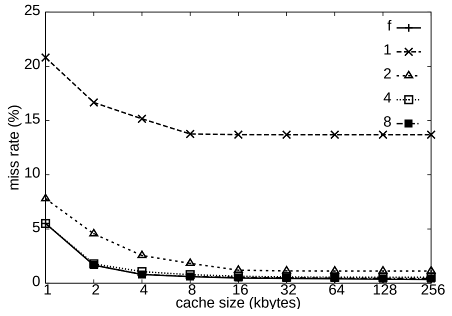
cache design:

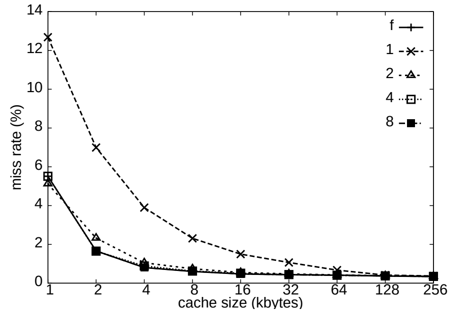
> **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).
>
> **associativity:** full, 1, 2, 4, 8
>
> **eviction policy:** LRU

Some results are in Figures A.36 to A.47. The results for rasterization order Z are not shown for most scenes as they are very similar to the results for rasterization order H. The only difference is that Z order gives slightly higher miss rates at small cache sizes *e.g.* compare Figures A.42 and A.44.

Discussion

The results in the graphs mostly agree with previous work showing that a 2-way set associative cache combined with index scheme z and some form of tiled raster-

(a) 

(b) 

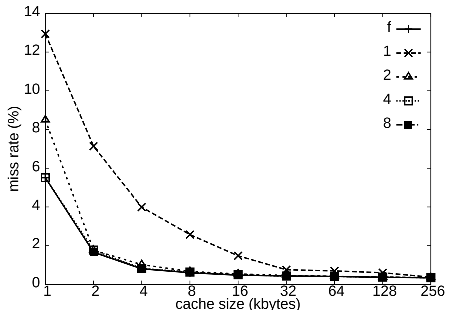
(c) 

Figure A.36: Miss rate curve for various cache associativities for scene *teapot* and rasterization order R for index schemes (a) r (b) h and (c) z.

ization is sufficient to deal with most possible conflict misses in our test cases. [1] Additionally, the abrupt changes in MIP-map levels accessed in scenes like *kground* and *teapot* do not seem to affect the miss rate noticeably compared to scenes that do not have such abrupt changes.

However the graph in Figure A.45(b), for scene *trifs-90* using rasterization order R and index scheme z, shows that it is not just the working set reducing properties of tiling in rasterization that reduces conflict misses as stated in [HG97], as the

[1]Previous work actually suggests using 6D-raster or 6D-Z-order with the supertiles the size of the cache [HG97]. It is clear that using index scheme z is similar except that there is tiling at all levels. Similarly, we metatile at all levels instead of just one level of 8×8 pixel blocks when rasterizing (while using 8×8 texel blocks for storing the texture).

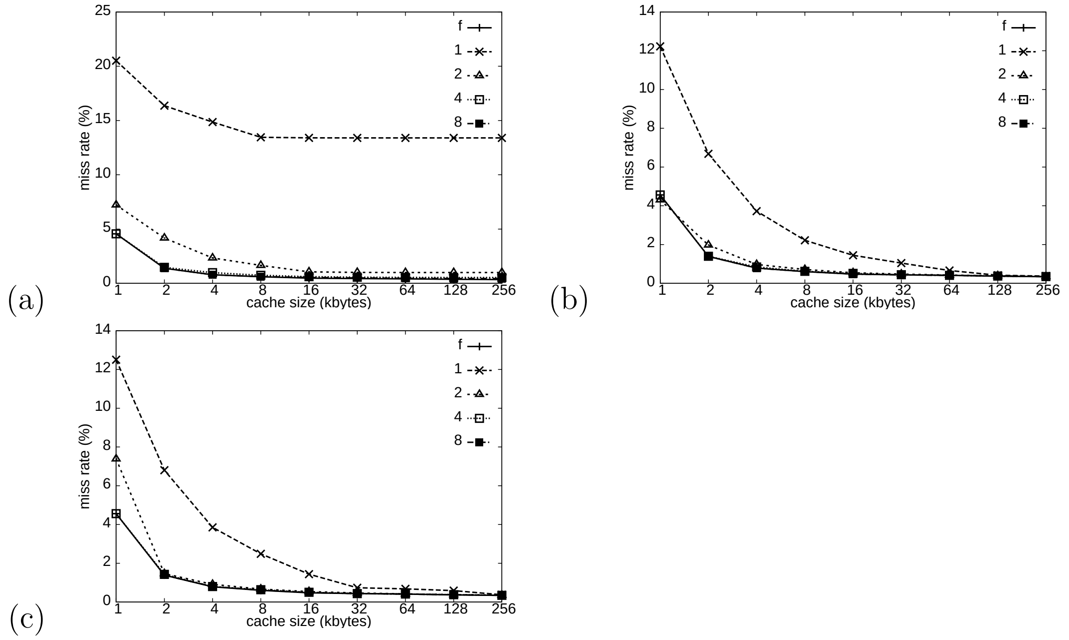


Figure A.37: Miss rate curve for various cache associativities for scene *teapot* and rasterization order H for index schemes (a) r (b) h and (c) z.

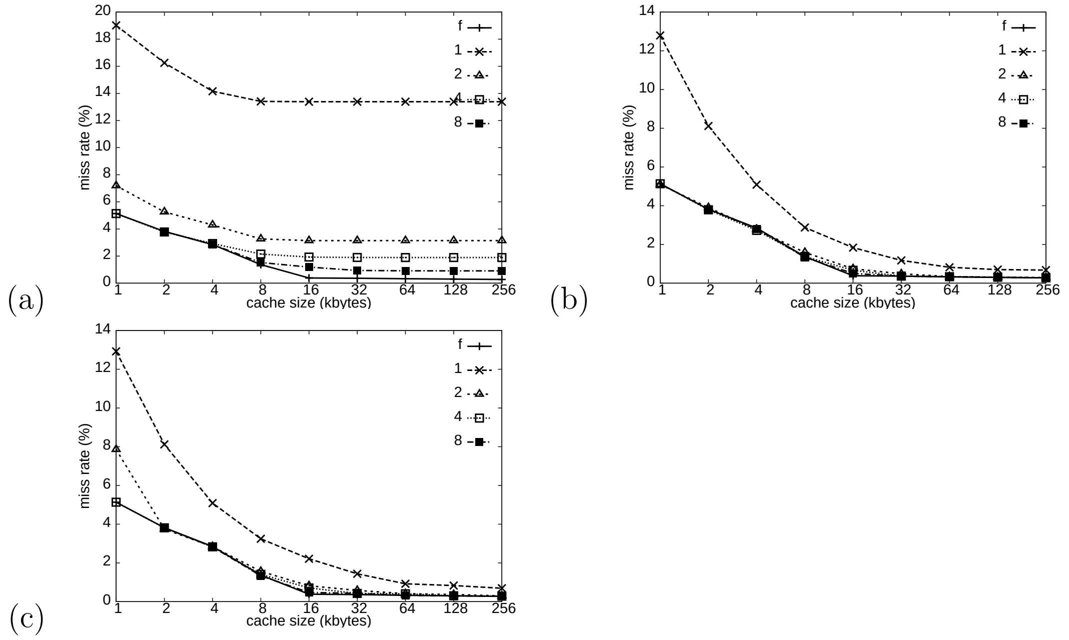


Figure A.38: Miss rate curve for various cache associativities for scene *box* and rasterization order R for index schemes (a) r (b) h and (c) z.

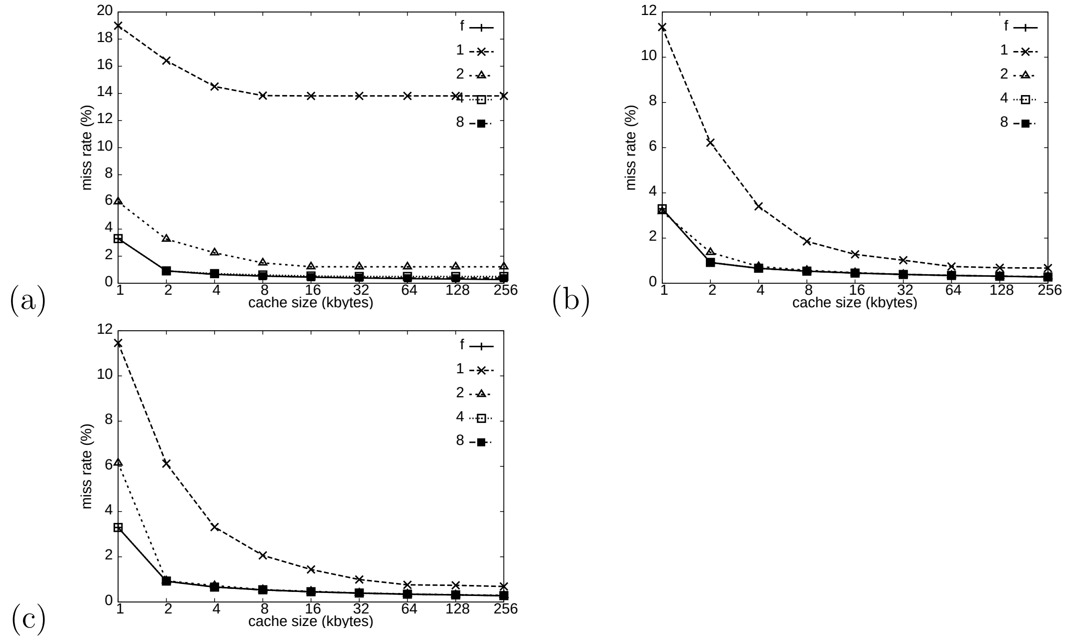


Figure A.39: Miss rate curve for various cache associativities for scene *box* and rasterization order H for index schemes (a) r (b) h and (c) z.

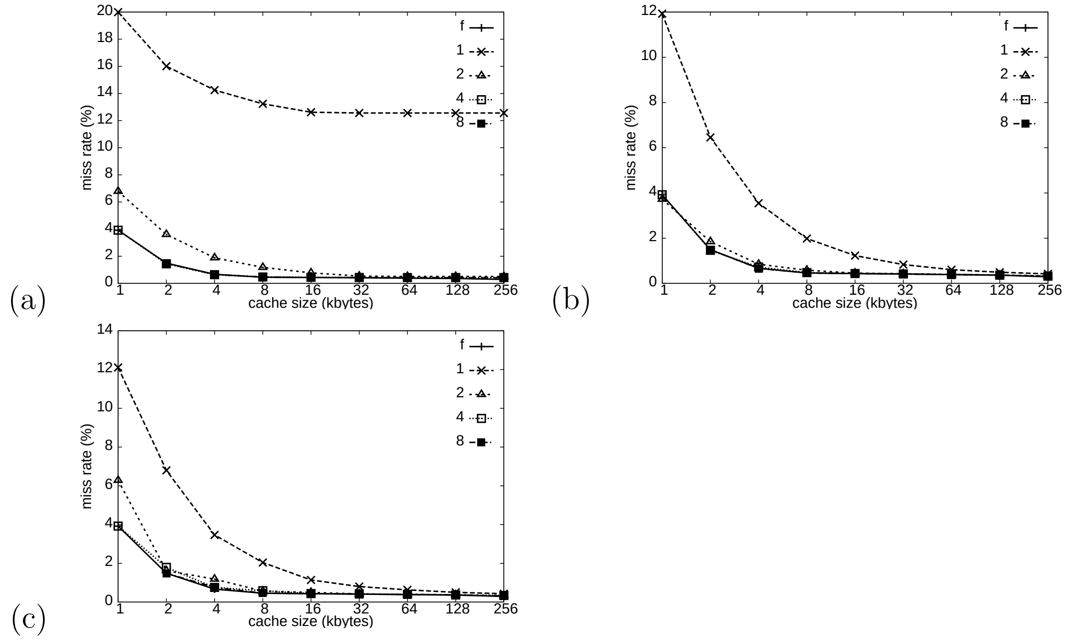
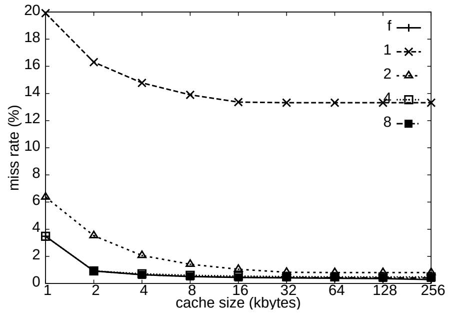
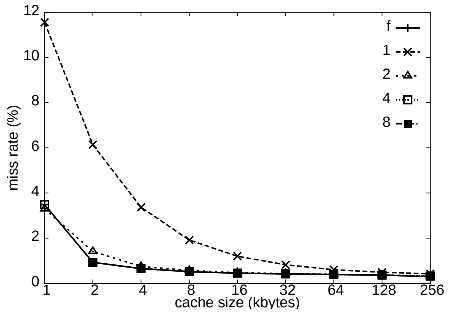


Figure A.40: Miss rate curve for various cache associativities for scene *kground* and rasterization order R for index schemes (a) r (b) h and (c) z.

(a)


(b)


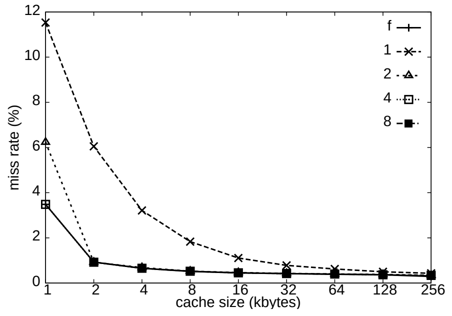
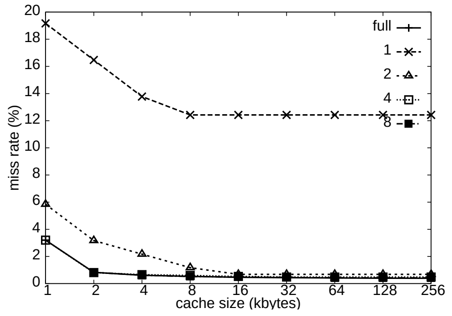
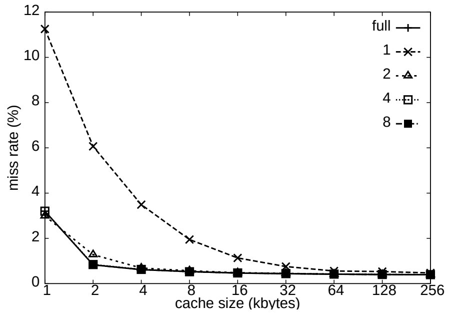
(c)


Figure A.41: Miss rate curve for various cache associativities for scene *kground* and rasterization order H for index schemes (a) r (b) h and (c) z.

(a)


(b)


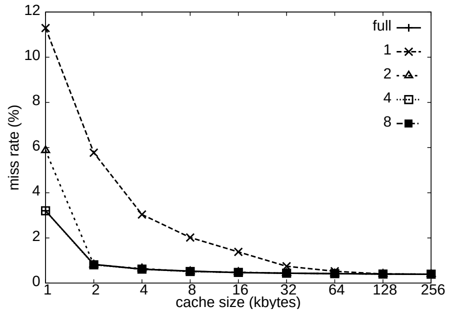
(c)


Figure A.42: Miss rate curve for various cache associativities for scene *trifs-90* and rasterization order H for index schemes (a) r (b) h and (c) z.

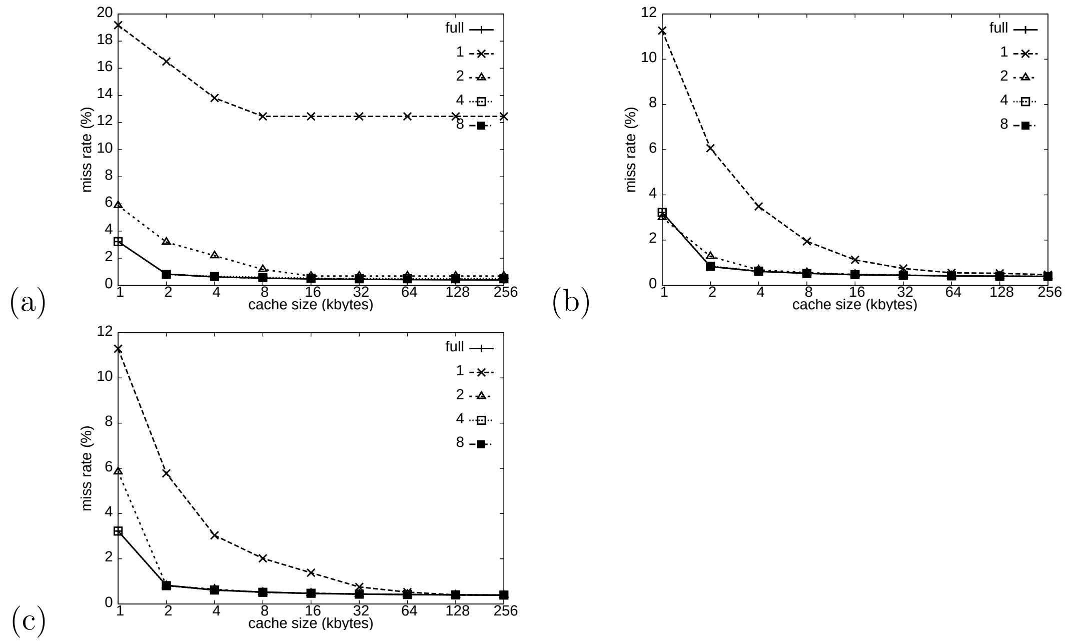


Figure A.43: Miss rate curve for various cache associativities for scene *trifs-0* and rasterization order H for index schemes (a) r (b) h and (c) z.

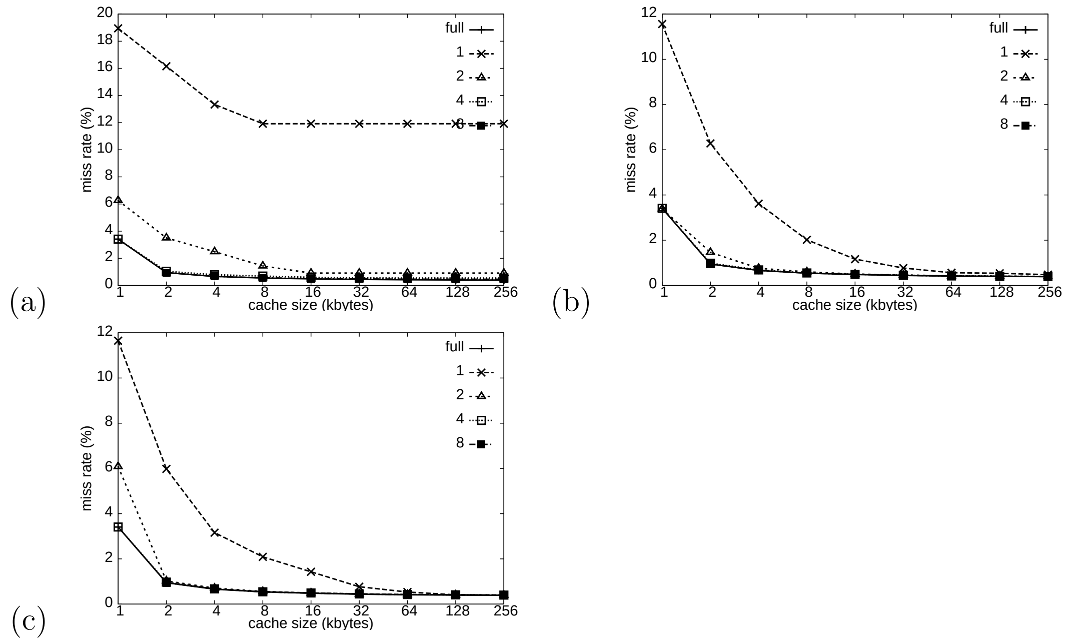
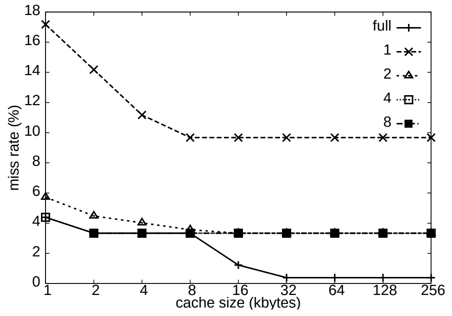
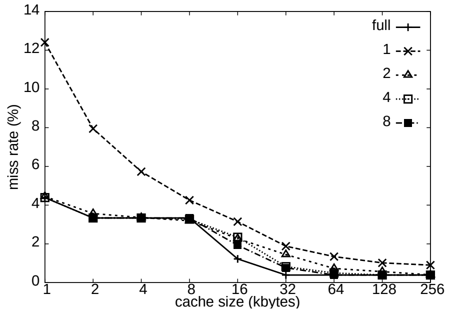


Figure A.44: Miss rate curve for various cache associativities for scene *trifs-90* and rasterization order Z for index schemes (a) r (b) h and (c) z.

(a)


(b)


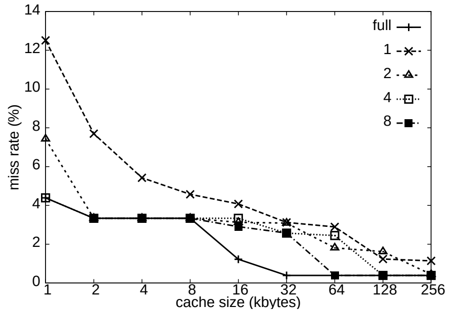
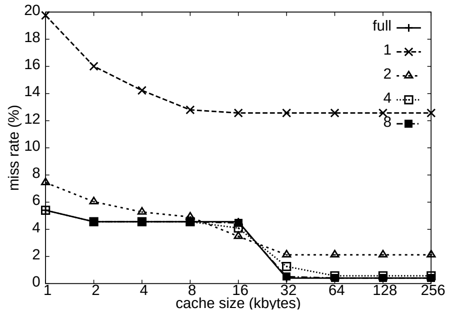
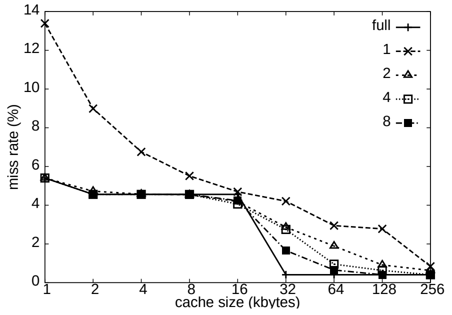
(c)


Figure A.45: Miss rate curve for various cache associativities for scene *trifs-90* and rasterization order R for index schemes (a) r (b) h and (c) z.

(a)


(b)


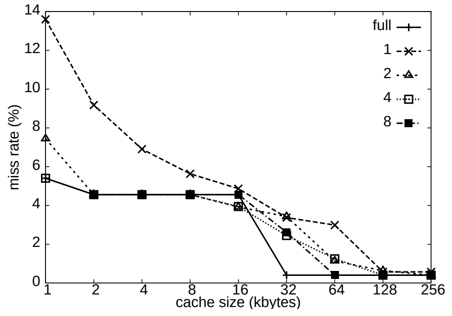
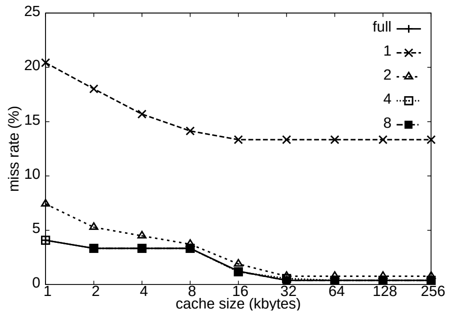
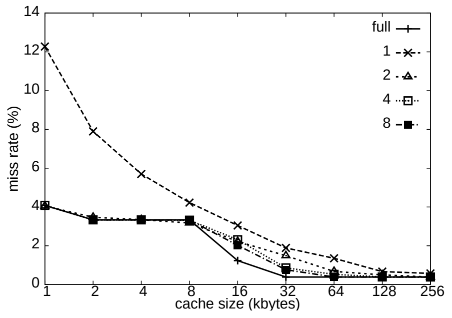
(c)


Figure A.46: Miss rate curve for various cache associativities for scene *trifs-45* and rasterization order R for index schemes (a) r (b) h and (c) z.

(a)


(b)


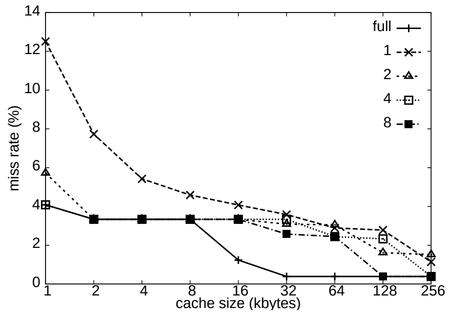
(c)


Figure A.47: Miss rate curve for various cache associativities for scene *trifs-0* and rasterization order R for index schemes (a) r (b) h and (c) z.

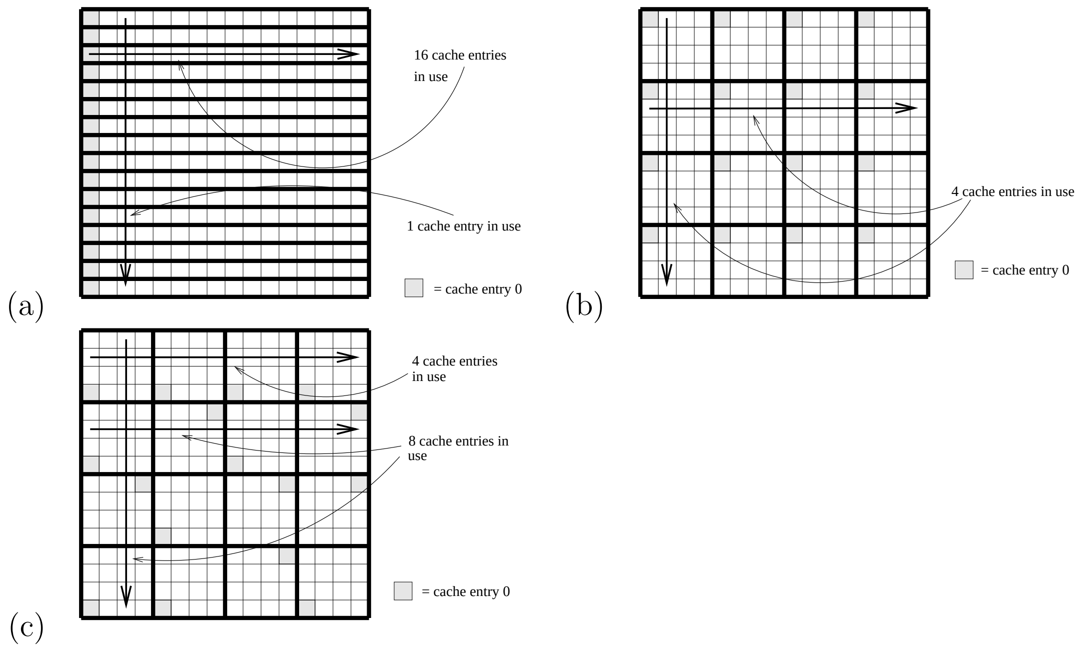
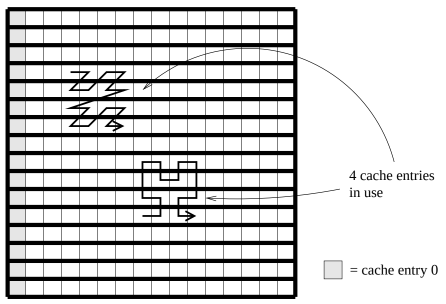


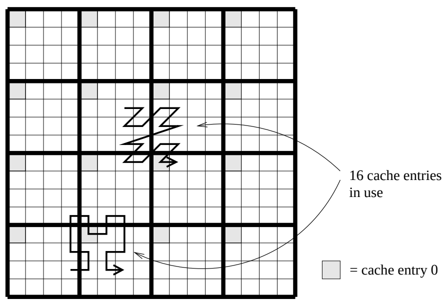
Figure A.48: Picture roughly showing how the cache indices are visited when rasterizing in R order. The bold lines delineate the way the texture blocks are mapped onto the cache by the index scheme. Each boldly outlined group of texture blocks map one-to-one contiguously to the entire cache. The index schemes used are: (a) r (b) z and (c) h.

curve for the 2-way set-associative cache is not close to the curve for the fully associative cache even at cache sizes much larger than the working set size.

Further explanation is found by looking at the curves for the 2-way set-associative and fully associative caches in Figures A.45(a), A.46(a), and A.47(a), which are the graphs using rasterization order R and index scheme r for the scenes *trifs-90*, *trifs-45*, and *trifs-0* respectively. The recurring motif of good performance when rasterization order R and index scheme r is aligned and bad performance when they are perpendicular occurs yet again. As shown in Figure A.48, the extreme anisotropy of the rasterization order R curve is transformed into extreme anisotropy of the path through the texture. This anisotropy means that the number of cache slots in use at a given time is sensitive to orientation if the index scheme is anisotropic as in the case of index scheme r (see Figure A.48(a)), and is restricted if the index scheme is more isotropic as in the case of index schemes z and h (see Figures A.48(b,c)). A rasterization order such as Z order or H order with many levels of nested square tiling would mean a less anisotropic path through the texture, resulting in less orientation sensitivity if the index scheme is r, and a more even distribution of the usage of cache entries for index schemes z and h (see Figure A.49). The former is supported by a comparison of Figures A.42(a) and A.43(a) which show that the miss rate curves for scenes *trifs-0* and *trifs-90* differ little when rasterization order H is used with index scheme r, the latter by a comparison of the differences between the curves for rasterization R in Figure A.45(b,c) and for rasterization order H in Figure A.42(b,c), where the difference in miss rate between the 2-way set-associative cache and the fully associative one for rasterization order R is high while the corresponding difference for rasterization order H is miniscule.

Now consider the miss rate curves that correspond to test cases where the rasterization order is H and index scheme z or h is used (see Figures A.37(b,c), A.39(b,c), A.41(b,c), A.42(b,c), and A.44(b,c)). They show that when the cache is of size at least 2K *i.e.* has at least eight entries, the miss rate curve for associativity 2 and

(a)


(b)


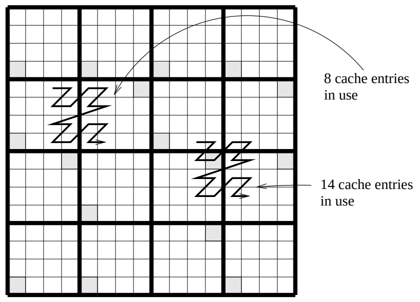
(c)


Figure A.49: Same as Figure A.48 except we rasterize using H and Z order. The index schemes used are: (a) r (b) z and (c) h.

index scheme h has a noticeably higher miss rate than the corresponding curve for index scheme z for the smaller cache sizes.

The explanation for this is in Figures A.49(b,c), where the cache index distribution for index scheme z is more even than the cache index distribution for index scheme h. Examining the figures, it is clear that the difference arises from the way the individual texture blocks are arranged in the delineated cache-sized superblocks; they are identical for index scheme z, but rotated in various directions for index scheme h. When rasterization order H is used, identical arrangements of cache entries for the texture blocks in each cache-sized superblock results in a more uniform distribution of usage of cache entries than if the arrangments are non-identical. It is also clear that any index scheme that tiles the texture blocks in cache-sized superblocks that are identical cache-index-wise *e.g.*, 6D-raster order and 6D-Z order, would produce identical miss rate curves if the other parameters are fixed.

An alternative way to deal with the uniform distribution problem is to calculate the cache index using a uniform pseudo-random function. The pseudo-randomness of the function removes any effect the rasterization curve has on the distribution of the indices. However, disadvantages include not guaranteeing that any four neighbouring blocks do not map to the same cache index and being more expensive to compute.

In summary, we find that in addition to using a 2-way set associative cache and an index scheme like 6D-raster or z-order, multiple levels of nested tiling as in H or Z order rasterization are necessary to minimize the number of conflict misses. Also, under the above conditions using index scheme h instead produces comparable miss rates to using index scheme z except at small cache sizes where they are higher.

A.4.3 Test: Cache Associativity and Turbulence

The previous test shows that the interaction of the rasterization curve and the index scheme is important for reducing the number of conflict misses. This brings up the question of what effect something like turbulence has on the number of conflict misses.

The test parameters are:

scenes: trifs-90, teapot

textures: dense

rasterization order: R, H

index schemes: r, z, h

shader parameters:

- ω**:** 0.0, 0.2, 0.4, 0.6
- **shader derivatives:** yes

sampling: trilinear

block size: 256

cache design:

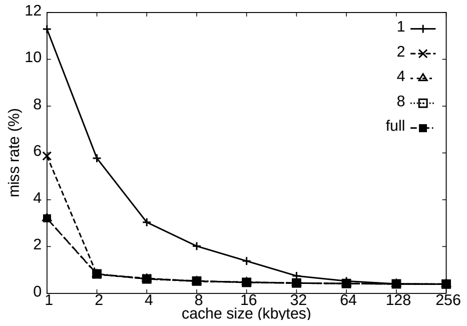
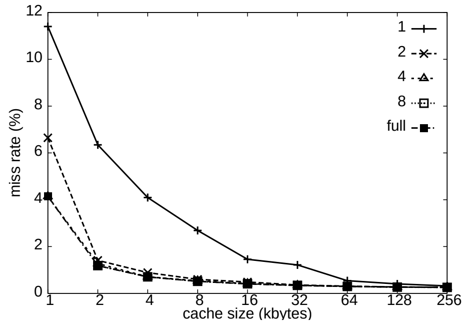
(a)  (b) 

Figure A.50: Miss rate curves for various cache associativities for scene *trifs-90*, rasterization order H, and index scheme z when $\omega =$ (a) 0.0 and (b) 0.6.

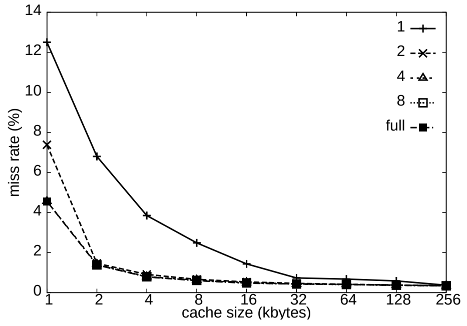
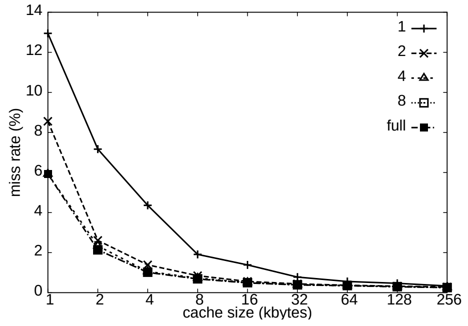
(a)  (b) 

Figure A.51: Miss rate curves for various cache associativities for scene *teapot*, rasterization order H, and index scheme z when $\omega =$ (a) 0.0 and (b) 0.6.

cache 1: One cache for only texture blocks (T blocks) and void blocks (V blocks).

associativity: full, 1, 2, 4, 8

eviction policy: LRU

Some results are in Figures A.50 and A.51.

Discussion

Similarly to the case where there is no turbulence, when there is metatiling in the rasterization order and the index scheme is z, a cache associativity of 2 removes most of the conflict misses.

A.4.4 Test: Cache Eviction Policy

Now we test the effect the eviction policy has on the miss rate curve. The test parameters are:

scenes: trifs-90, teapot, box

textures: dense

rasterization order: R, H

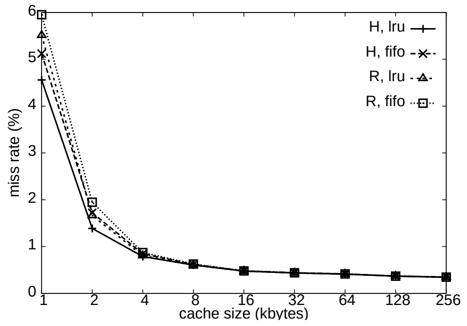
index schemes: r, z, h

sampling: trilinear

shader parameters: no turbulence shader

block size: 256

cache design:

(a) 

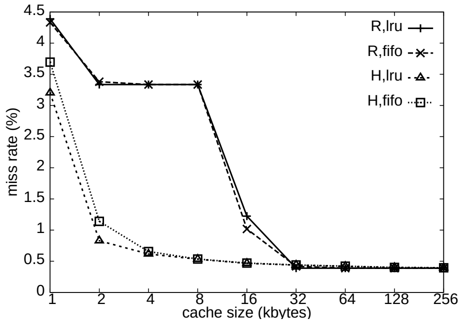
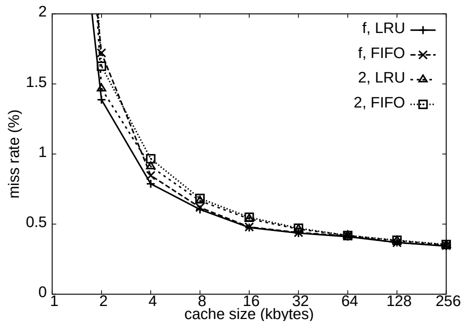
(b) 

Figure A.52: Miss rate curves for LRU and FIFO eviction policies and rasterization orders H and R using a fully associative cache for scenes (a) teapot and (b) trifs90.

(a) 

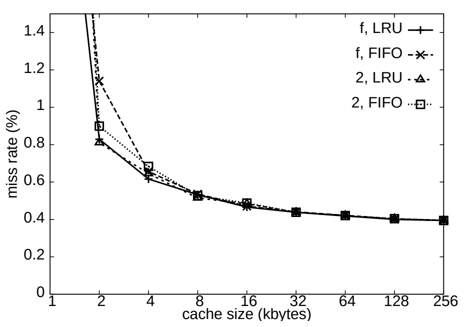
(b) 

Figure A.53: Miss rate curves for LRU and FIFO eviction policies and rasterization order H and index scheme z using a cache of various associativies for scenes (a) teapot and (b) trifs90.

cache 1: One cache for only texture blocks (T blocks) and void blocks (V blocks).

associativity: full, 1, 2, 4, 8

eviction policy: LRU, FIFO

Some results are in Figure A.52 and A.53.

Discussion

The change in eviction policy makes hardly any difference to the miss rate in the given graphs, and for rasterization order H any differences are only apparent at small cache sizes. Similar results were obtained with other index schemes and scenes. We conclude that FIFO can replace LRU as the eviction policy for most practical purposes.

A.5 Separate Void Cache Tests

This section tests bandwidth and miss rate reductions for cache designs that have a separate cache for the void parts of the texture. The effects of the B-tree index blocks are ignored.

A.5.1 Test: Unified Cache

This test establishes for comparison with future tests the bandwidth consumption and the miss rate we get in a block-based fully associative cache if the bandwidth due to void blocks is excluded. The test parameters are:

scenes: trifs-90, teapot, box

textures: cat, funny3, cellorig, check, sine11-91, dense

rasterization order: R, H – we do not test Z because it is so similar to H for a fully associative cache.

index schemes: r

sampling: trilinear

block size: 64, 256, 1024

cache design:

- **cache 1:** One cache for only texture blocks (T blocks) and void blocks (V blocks).
  - **associativity:** full
  - **eviction policy:** LRU

The miss rate is shown in Figure A.54. The bandwidth consumption if void blocks do not contribute to bandwidth are shown in Figures A.56 to A.58.

A.5.2 Test: Separate Cache for Void Blocks

The next test sees if using separate caches for the texture (T) and void (V) blocks can reduce the miss rate and bandwidth consumption. The test parameters are:

scenes: trifs-90, teapot, box

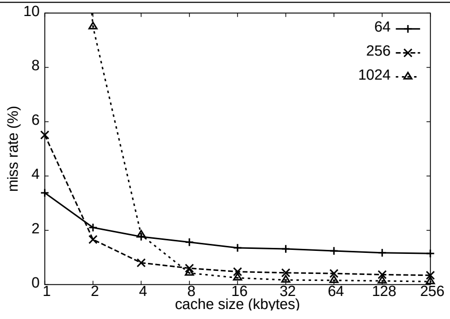
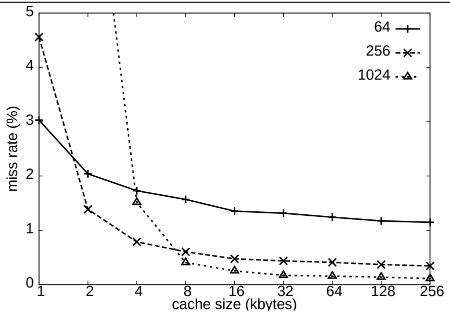
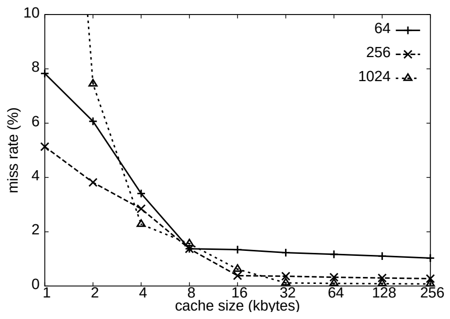
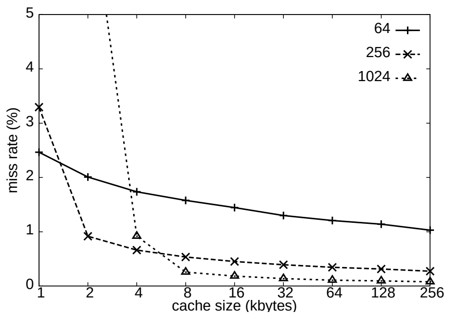
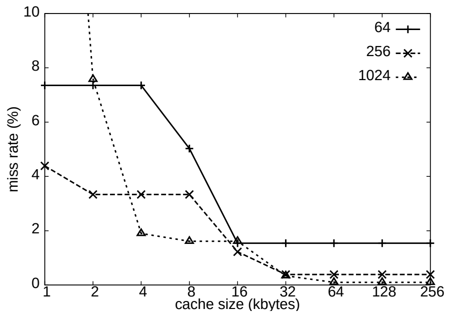
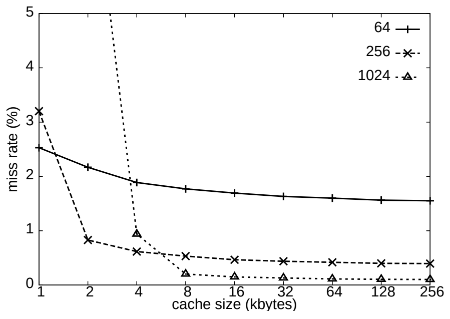
Scene	R	H
teapot		
box		
trifs90		

Figure A.54: The miss rate curves when using a fully associative block-based cache for various scenes and rasterization orders R and H.

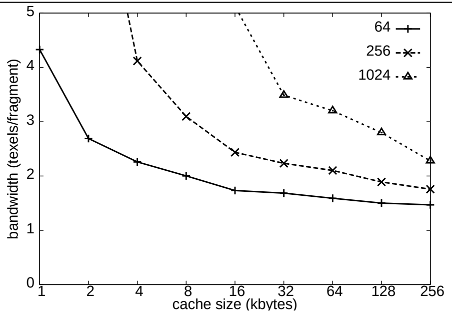
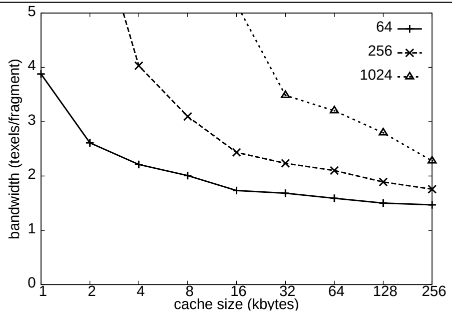
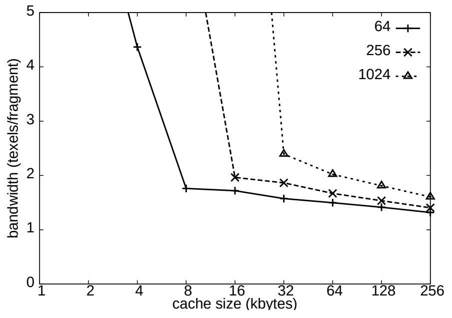
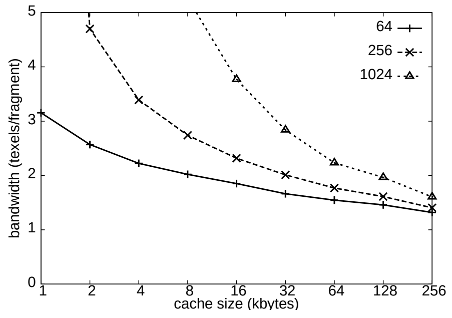
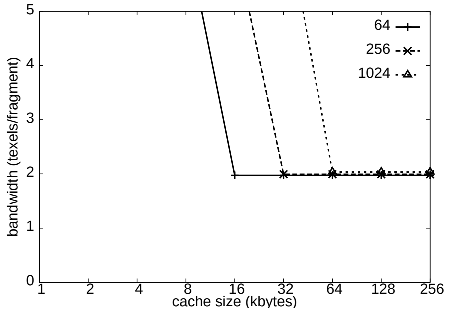
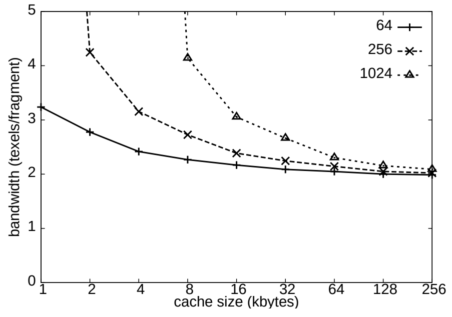
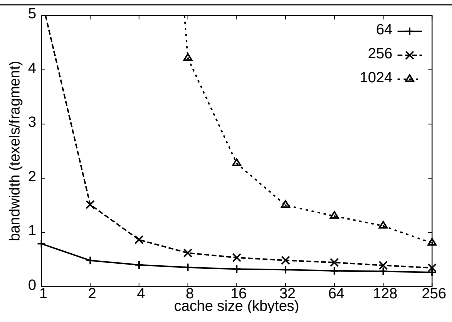
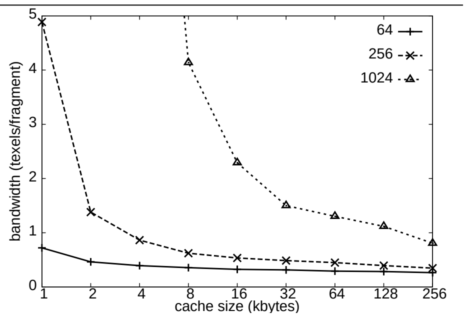
Scene	*R*	*H*
teapot		
box		
trifs90		

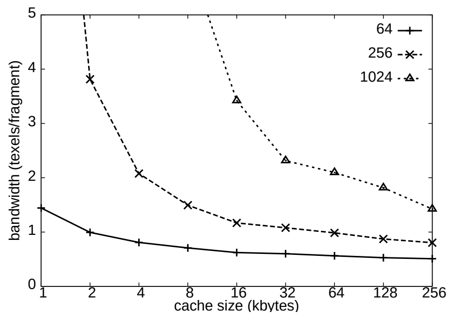
Figure A.55: The bandwidth consumption curves when using a fully associative block-based cache for various scenes and rasterization orders *R* and *H*.

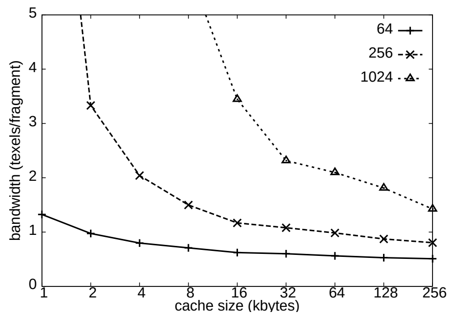
Texture R H



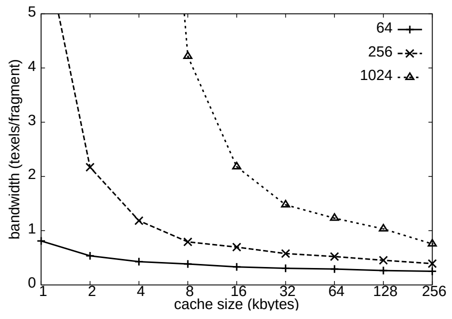


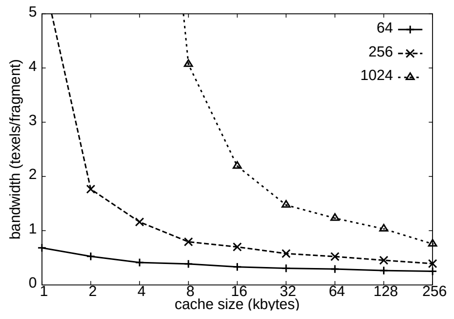
cat



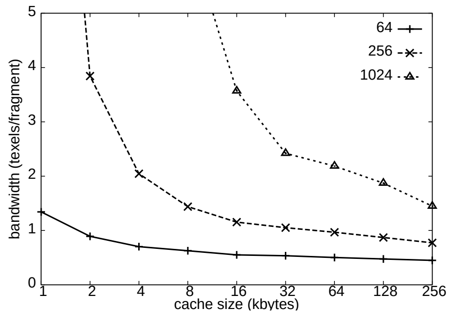


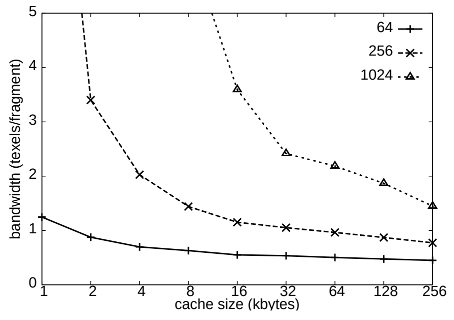
cellorig





funny3





sine11-91

Figure A.56: The bandwidth consumption curves for scene *teapot* and rasterization orders R and H when void blocks contribute no bandwidth.

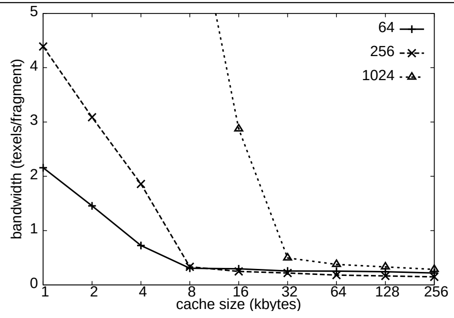
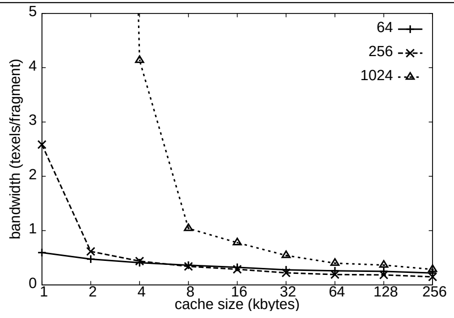
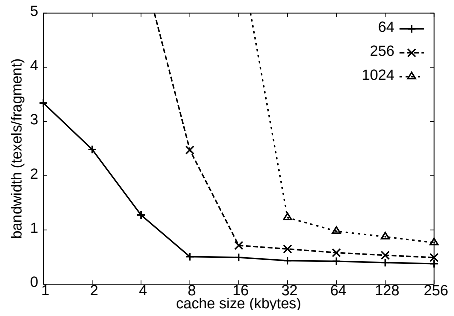
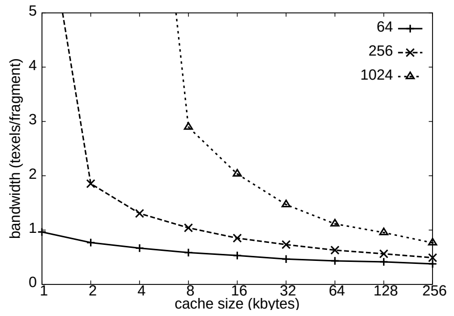
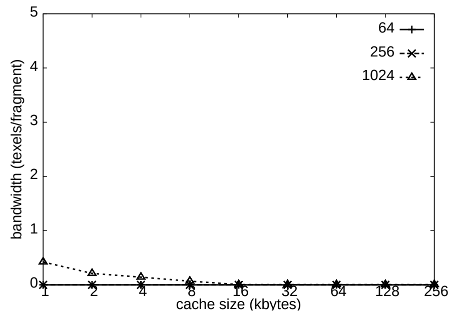
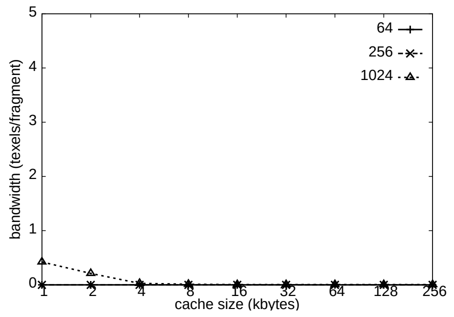
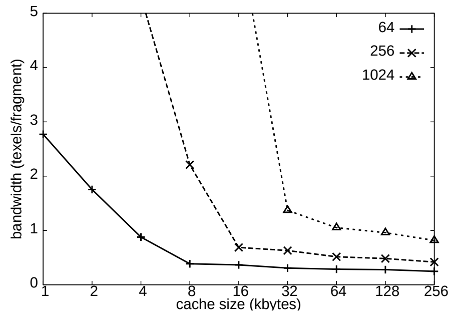
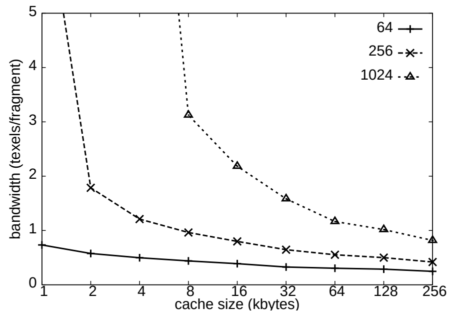
Texture	R	H
cat		
cellorig		
funny3		
sine11-91		

Figure A.57: The bandwidth consumption curves for scene *box* and rasterization orders R and H when void blocks contribute no bandwidth.

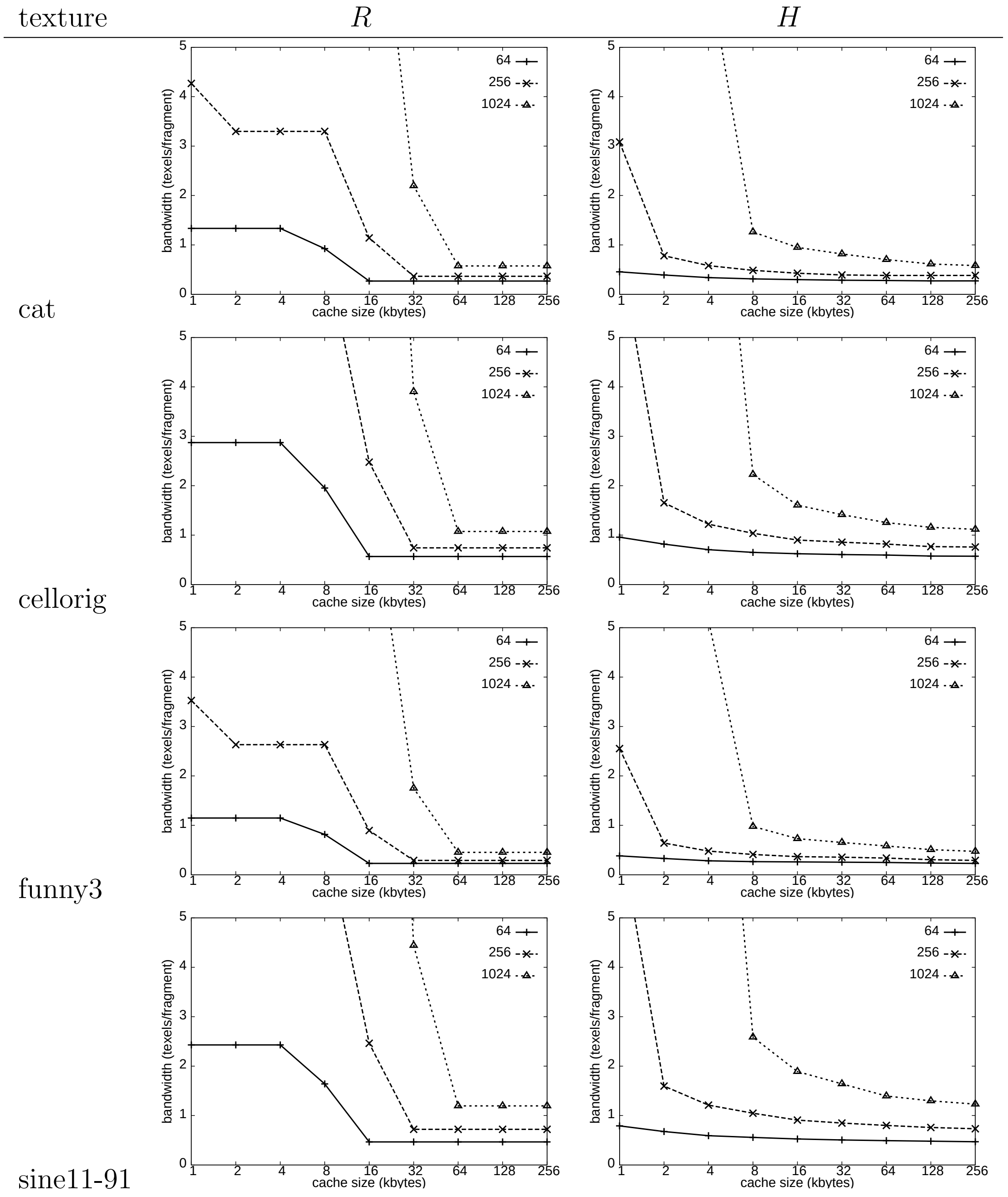


Figure A.58: The bandwidth consumption curves for scene *trifs90* and rasterization orders R and H when void blocks contribute no bandwidth.

textures: cat, funny3, cellorig, check, sine11-91, dense

rasterization order: R, H – we do not test Z because it is so similar to H for a fully associative cache.

index schemes: r

sampling: trilinear

block size: 64, 256

cache design:

cache 1: One cache for only texture blocks (T blocks).

associativity: full

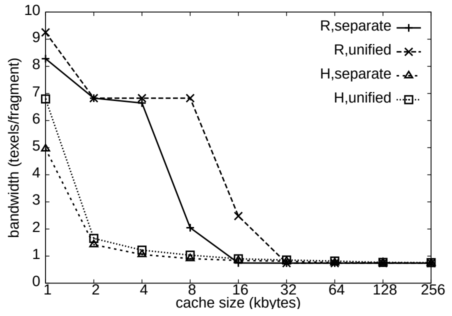
eviction policy: LRU

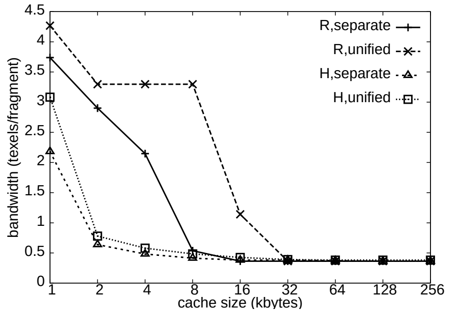
cache 2: One cache for only void blocks (V blocks).

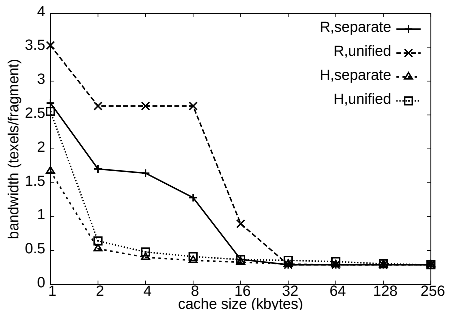
associativity: full

eviction policy: LRU

Some results are in Figures A.59 to A.61.

(a)


(b)


(c)


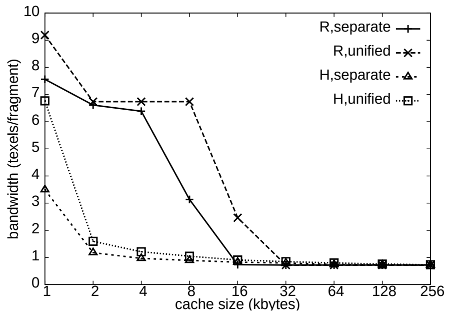
(d)


Figure A.59: Comparison of bandwidth consumption curves for scene *trifs90* and block size 256 when using one cache and when using a separate cache for the void parts of the texture for textures (a) cellorig (b) cat (c) funny3 and (d) sine11-91.

Discussion

The graphs in Figures A.59 to A.60 show that the less spatial locality there is, the more effective using separate caches for texture and void blocks is. The most drastic improvements occur when rasterizing in R order for scene *trifs-90* (Figure A.59). Even the bandwidth reductions for scene *teapot* are noticeably better than the small bandwidth reductions for scene *trifs-90* and rasterization order H. Given the way that the occupied blocks tend to be clustered together in the texture, a working set is more likely to contain both void blocks and texture blocks when the spatial locality is poor than when the spatial locality is good, resulting in the given

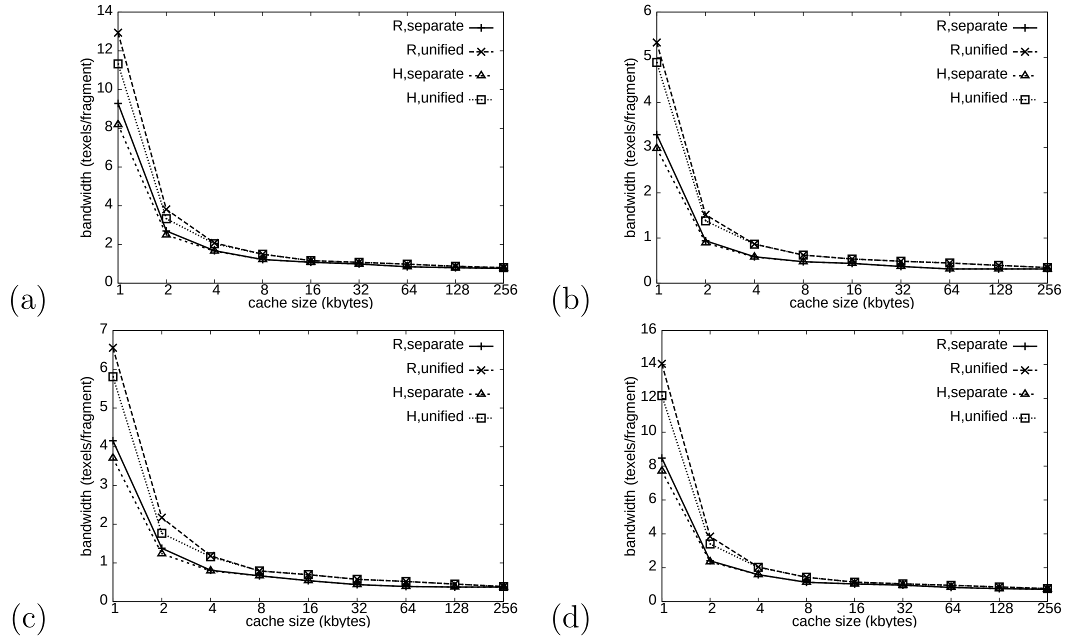


Figure A.60: Comparison of bandwidth consumption curves for scene *teapot* and block size 256 when using one cache and when using a separate cache for the void parts of the texture for textures (a) cellorig (b) cat (c) funny3 and (d) sine11-91.

variance in effectiveness of the use of separate caches. If the goal of separate caches is to reduce bandwidth consumption, then they are best used when poor spatial locality of the rasterization curve is expected.

It is also interesting to note that when the graph of the bandwidth consumption for texture *funny3*, rasterization order *R*, scene *trifs-90*, and block size 256 in Figure A.59(c) is compared to the corresponding graph for a dense texture and scene *trifs-90* in Figure A.55, the minor reduction in working set size does not reflect the huge decrease in texture occupation. However, the corresponding curves for rasterization order *H* in the same graphs do reflect the decrease in texture occupation. This is

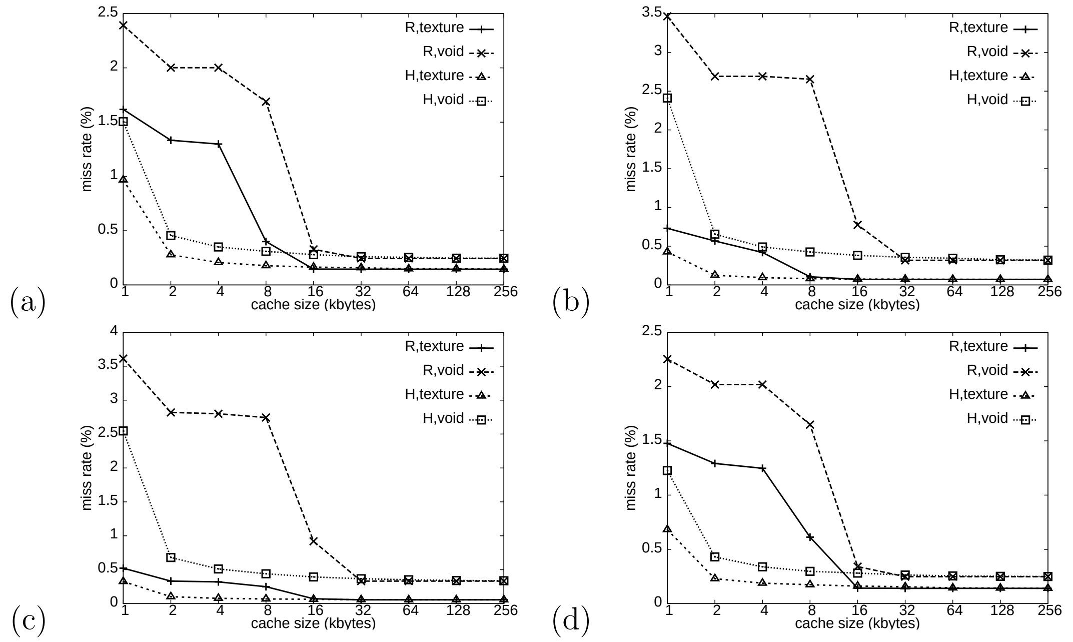


Figure A.61: Miss rate contribution of the individual void block and texture block caches to the total miss rate. The scene is trifs-90 , the block size is 256 and the texture is (a) cellorig (b) cat (c) funny3 and (d) sine11-91.

explained by the anisotropic features in the texture being aligned with the direction of rasterization in rasterization order R ensuring that the working set size is still roughly the amount of texture required to render a scanline. This problem does not occur when rasterizing in H order because it is not very anisotropic.

Theoretically, we can reduce the total miss rate (ignoring the B-tree indices) by making the void block cache larger than the texture block cache. (In order to deal with rendering from mostly void parts of the texture as ably as with rendering from mostly occupied parts of the texture, the void block cache must be at least as large as the texture block cache in terms of number of entries.) However, for rasterization order R both texture and void block caches need to be large enough to contain their respective working sets so that most of the misses will already be cold misses (see Figure A.61). For rasterization order H, the miss rate curve is so flat that again most of the void block misses are cold misses (again see Figure A.61). Making the void block cache larger does not significantly affect the miss rate unless the original cache was too small to begin with. What is necessary instead is a way to reduce the number of cold misses due to the void parts of the texture.

A.5.3 Test: Separate Cache for Void Key-Intervals

We have already seen in previous tests (see Section A.2) how using a range-cache to store void key-intervals reduces the miss rate due to the void parts of the texture to well below the cold miss rate for void blocks, even with very few entries in the range-cache. Here we test with more typical scenes. The test parameters are:

scenes: trifs-90, teapot, box

textures: cat, funny3, cellorig, check, sine11-91, dense

rasterization order: R, H – we do not test Z because it is so similar to H for a fully associative cache.

index schemes: r

sampling: trilinear

block size: 64, 256

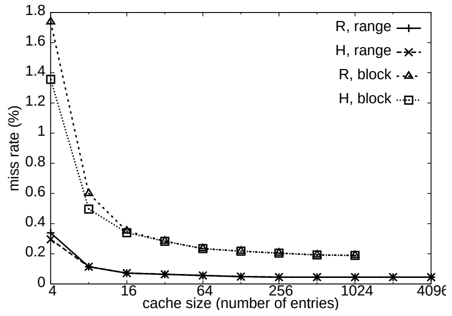
cache design:

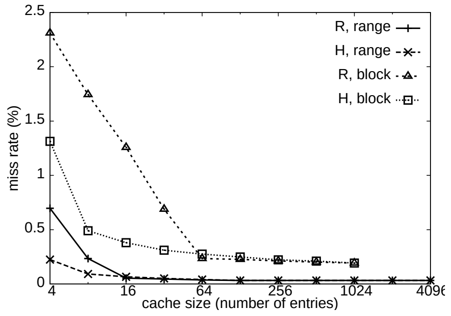
> **cache 1:** One cache for only texture blocks (T blocks).
>
> > **associativity:** full
> >
> > **eviction policy:** LRU
>
> **cache 2:** One range-cache for void key-intervals.
>
> > **eviction policy:** LRU, FIFO

Some results are in Figure A.62 to A.64.

Discussion

We can see from the graphs that we get miss rates well below the cold miss rate for a void block cache if we use a range-cache and rasterization order H for all the scenes, even for very small cache sizes. We got similar results using other textures and index schemes.

(a)


(b)


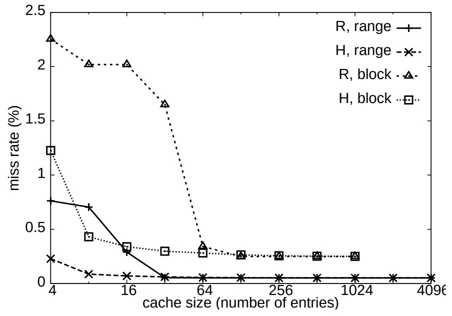
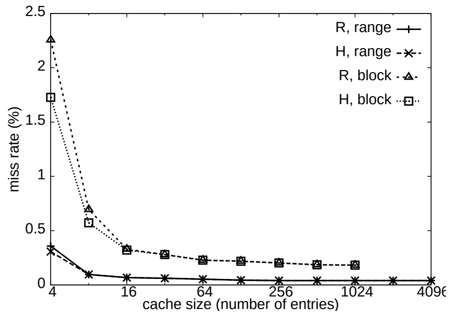
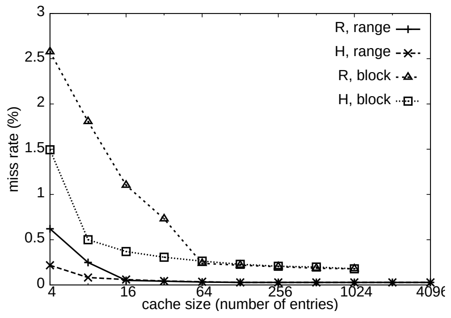
(c)


Figure A.62: Graphs comparing the miss rates of a void range-cache versus a void block-cache for rasterization orders R and H, index scheme z, and texture *sine11-91*. The scenes used are (a) teapot (b) box and (c) trifs-90.

(a)


(b)


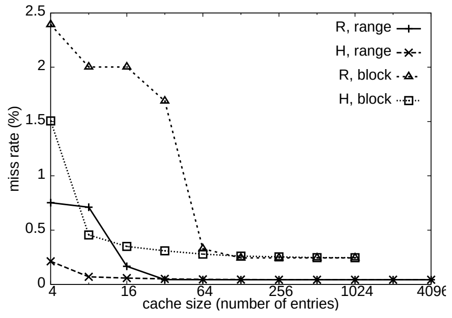
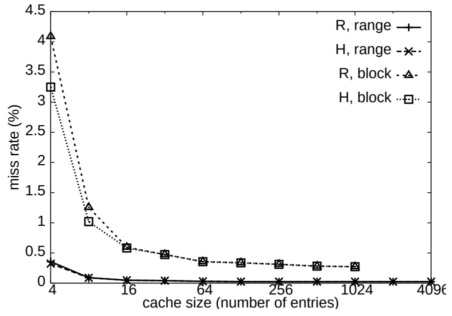
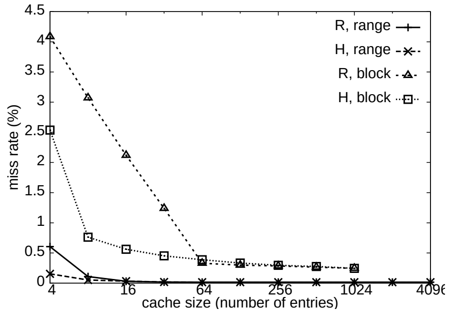
(c)


Figure A.63: The same as Figure A.62 except the texture is *cellorig*.

(a)  (b) 

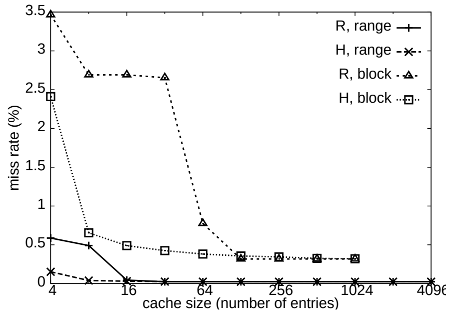
(c) 

Figure A.64: The same as Figure A.62 except the texture is *cat*.

A.6 Test: Storing the Index Blocks With the Texture Blocks

The next test examines the additional bandwidth incurred when the index blocks are stored in the same cache as the texture blocks. Since the miss rate curve for a void range-cache is very flat, we use a very large void range-cache in our test. The test parameters are as follows:

scenes: trifs-90, trifs-0, teapot

textures: dense, check, cat

rasterization order: H – we do not test Z because it is so similar to H in all previous tests. R is not tested because in previous tests it was shown that using it required the cache to have a high associativity to give a miss rate comparable to that for a fully associative cache.

index schemes: z, h – r is not tested for the same reason R is not.

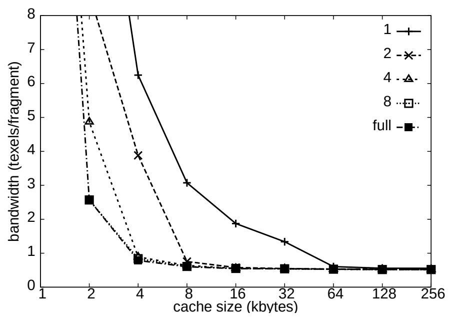
sampling: trilinear

block size: 256

cache design:

- **cache 1:** One cache for only texture blocks (T blocks) and index blocks (I blocks). We refer to it as a TI cache.
  - **associativity:** full, 1, 2, 4, 8
  - **eviction policy:** LRU
- **cache 2:** A void range-cache of size 8192, large enough so all misses are cold misses.

Some results for scene *trifs-90* are in Figures A.65 and A.66.

(a)


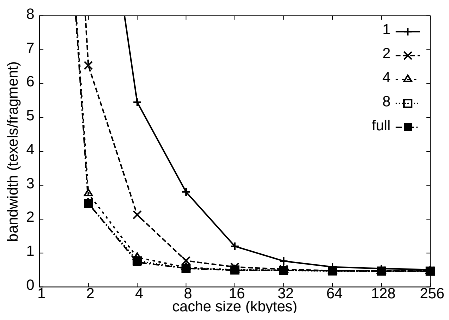
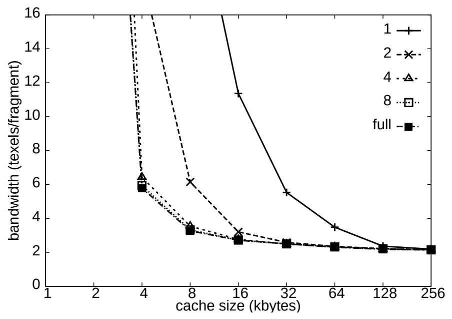
(b)


Figure A.65: The bandwidth consumption curves for various cache associativies. The scene is *trifs-90*, the texture is *cat*, and the rasterization order is H. The index scheme is (a) z (b) h.

(a)


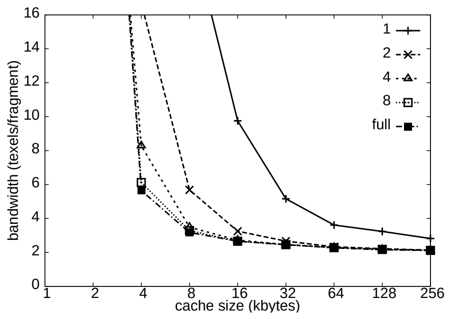
(b)


Figure A.66: The bandwidth consumption curves for various cache associativies. The scene is *trifs-90*, the texture is *dense*, and the rasterization order is H. The index scheme is (a) z (b) h.

Discussion

Regardless of the rasterization order/index scheme combination, Figures A.65 to A.66 show that a high associativity is needed to get comparable miss rates to that achieved with a fully associative cache, even when the texture is relatively sparse. The same effect was noticed with the other scenes and textures. Since this effect is quite strong for the comparatively small number of index blocks relative to the number of texture blocks, the index blocks should be cached separately from the texture data.

A.7 The Index Block Cache

This set of tests examines the effect of test parameters on the bandwidth incurred by the index blocks and latency of access when a cache is used to store the index blocks exclusively.

A.7.1 Test: Curve Shapes

We determine the bandwidth consumption curve shapes for a fully associative cache that stores only index blocks. Since the index block cache is only accessed when there is a miss when accessing texture data, the miss rate curve for the index block cache is dependent on the misses for texture blocks and void blocks or key-intervals. Again we use a very large void range-cache. For the texture blocks we use a fully associative cache of different sizes to give different miss rates. The test parameters are as follows:

scenes: trifs-90, trifs-0, teapot

textures: dense, check, cat, cellorig

rasterization order: R, H – we do not test Z because it is so similar to H in all previous tests.

index schemes: r, z, h

sampling: trilinear

block size: 64

cache design:

- **cache 1:** One cache for index blocks (I blocks).
  - **associativity:** full
  - **eviction policy:** LRU
- **cache 2:** A void range-cache of size 8192, large enough so all misses are cold misses.
- **cache 3:** A fully associative cache used to store texture blocks.
  - **eviction policy:** LRU
  - **size:** 4K, 16K, 64K

The results are in Figures A.67 to A.69.

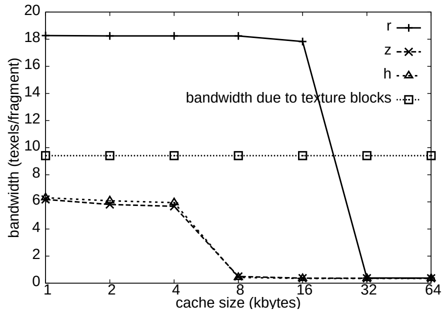
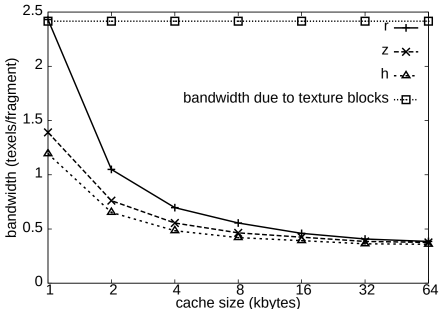
(a)  (b) 

Figure A.67: The bandwidth due to the index blocks for various index schemes. The scene is *trifs90*, the texture is *dense*, the texture block cache size is 4K, and the rasterization order is (a) R and (b) H.

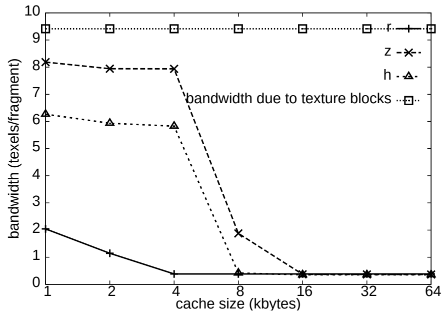
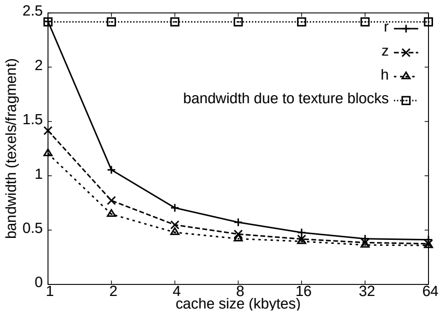
(a)  (b) 

Figure A.68: The bandwidth due to the index blocks for various index schemes. The scene is *trifs0*, the texture is *dense*, the texture block cache size is 4K, and the rasterization order is (a) R and (b) H.

Discussion

The results for scene *trifs-90* in Figure A.67 shows that when geometry is not a factor, the basic shape of the bandwidth consumption curve is determined by the rasterization order independent of the index scheme. Rasterization order R gives a distinct step corresponding to a distinct working set size, and rasterization order H giving a smooth decline. The bandwidth consumption when using rasterization

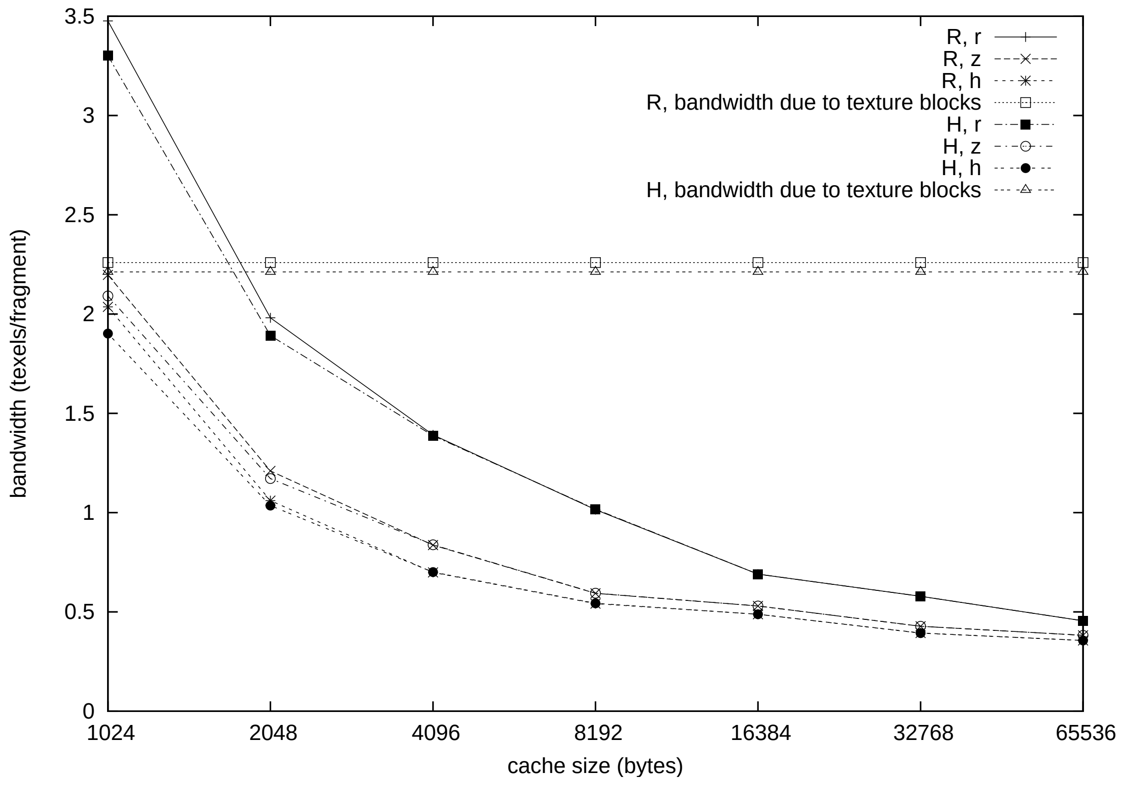
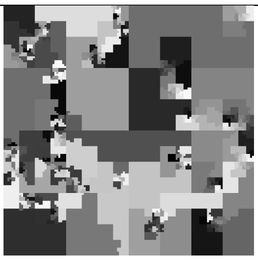
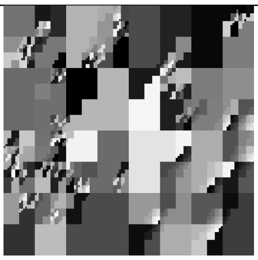
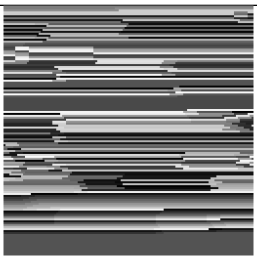
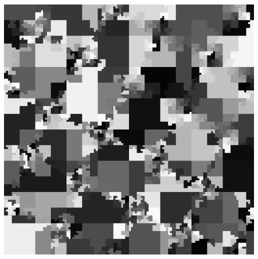
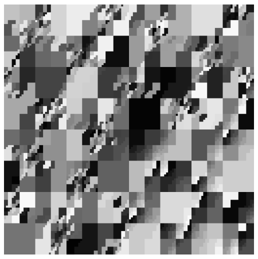
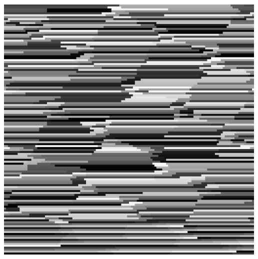
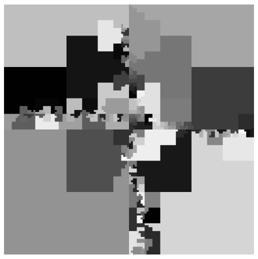
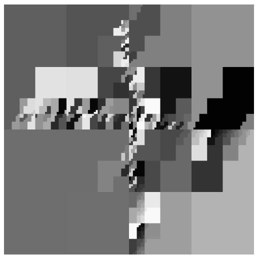
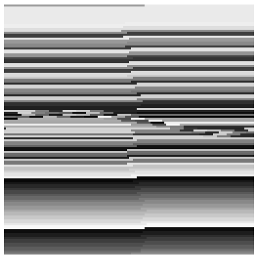


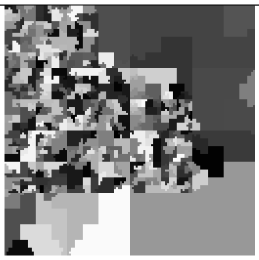
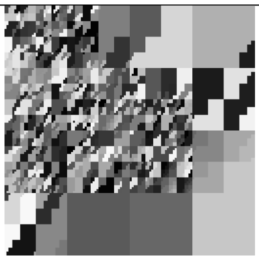
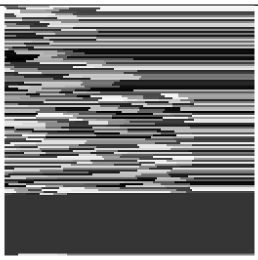
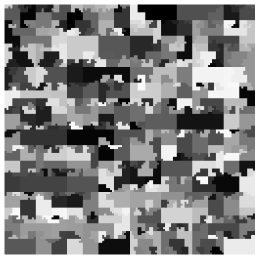
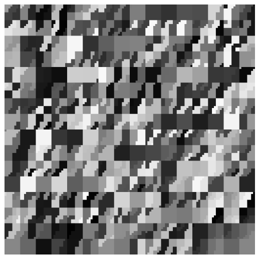
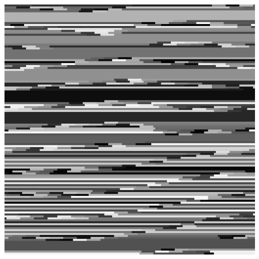
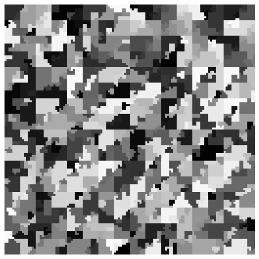
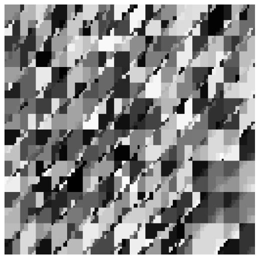
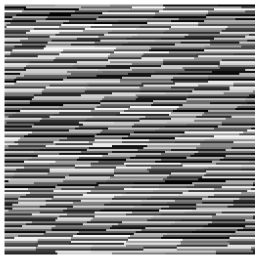
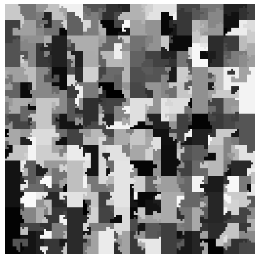
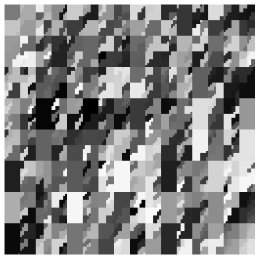
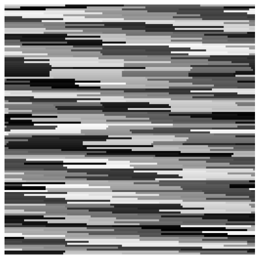
Figure A.69: The bandwidth due to the index blocks for various index schemes and rasterization orders for scene *teapot*, texture block cache size is 4K, and texture *dense*.

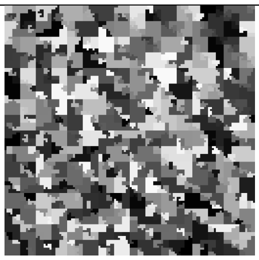
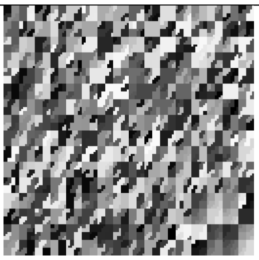
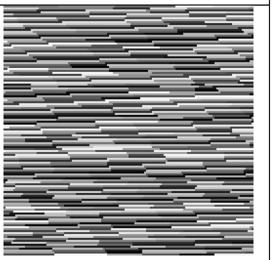
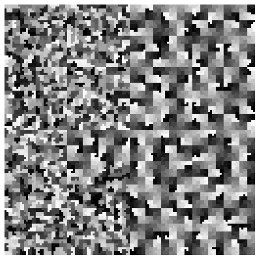
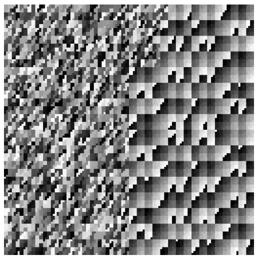
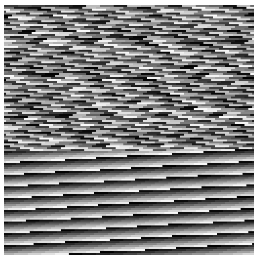
order H is also much lower than that when using rasterization order R for small cache sizes.

Each part of the texture corresponds to a path of index blocks in the B-tree index, and the vast majority of these blocks are the key-blocks at the end of a path. Similarly to the case where the texture is partially covered by void key-intervals, each key-block covers part of the texture in shapes similar to the shapes of the larger void key-intervals in a sparse texture (see Table A.3). The shape of the curves can then be explained by the relationship between these shapes and the order and location of the texel accesses that result in a miss in the texture block cache (see Figure A.70). The former is primarily determined by the index scheme, the latter by the rasterization curve.

Table A.3: Pictures showing how the areas of each texture are assigned to the leaf blocks in a compacted B-tree depending on the index scheme chosen. The texture is blocked into 4 × 4 texel blocks.

Texture Name	Index Scheme		
	h	z	r
cat			
cellorig			
funny3			

Texture Name	Index Scheme		
	h	z	r
galaxies-bw			
sine11-1			
sine11-45			
sine11-91			

Texture Name	Index Scheme		
	h	z	r
sine11-135			
dense			

For scene *trifs-90*, the order and location of texel accesses corresponding to texture block misses is close enough to the original rasterization curve that the results are very similar to the results in the basic tests for the void-range-cache (see Section A.2) and can be explained by the same explanations. Supporting this is a comparison of Figure A.67 for scene *trifs-90* and Figure A.68 for scene *trifs-0*. They show that the same orientation-sensitivity issues that occur with void key-intervals occurs here, with the rasterization order R and index scheme r combination only working well when they are aligned in the same direction, and other combinations being relatively insensitive in comparison.

The results for scene *teapot* in Figure A.69 show that like the case for a texture

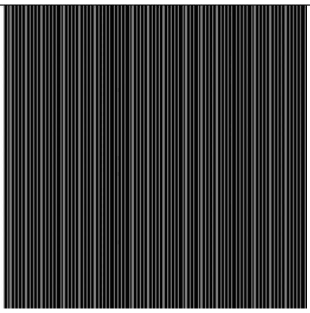
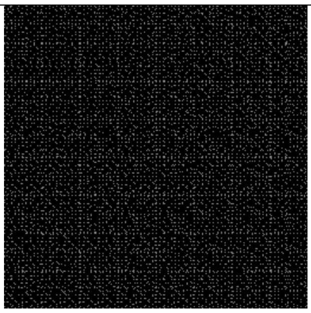
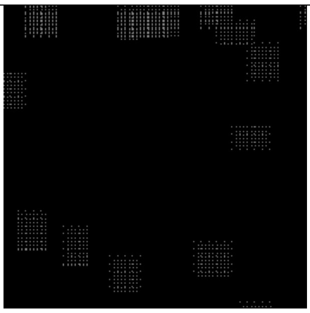
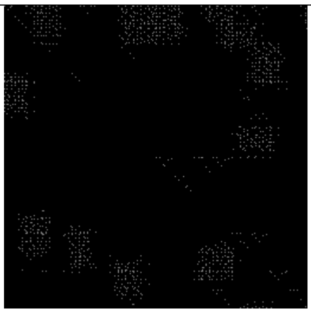
Texture	R	H
dense		
cat		

Figure A.70: The location of texture block and void key-interval misses that occur in the tests for a texture block cache size of 4K, scene *trifs-90*, and rasterization orders R and H.

block cache, when the triangles are not large the curve for the index block cache is not much affected by the rasterization order used. However, the curve is affected by the index scheme, with the bandwidth overhead increasing as we go from index scheme h, to z, to r, with the first two close to each other and significantly lower than the third. We attribute the ordering to the much better spatial locality of index schemes h and z better translating the spatial locality generated by the rasterizer into locality of reference for the index blocks.

In conclusion, to keep the bandwidth overhead due to the index blocks as low as possible the rasterization order and index scheme combination must be chosen to give as good locality as possible in the B-tree key-space for the scene. Usually this will mean using rasterization order H with index scheme h or z. Using index scheme z will produce slightly higher index block bandwidth consumption than using index scheme h with this cache setup in most situations.

We get similar results for other textures and texture block cache sizes.

A.7.2 Test: Block Size

This test is to see how changing the block size changes the bandwidth usage due to the index blocks. The test parameters are:

scenes: trifs-90, trifs-0, teapot

textures: dense, check, cat, cellorig

rasterization order: R, H – we do not test Z because it is so similar to H in all previous tests.

index schemes: r, z, h

sampling: trilinear

block size: 64, 256, 1024

cache design:

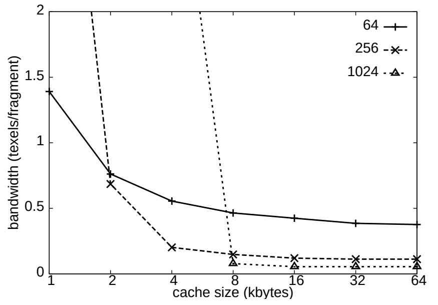
- **cache 1:** One cache for index blocks (I blocks).
  - **associativity:** full
  - **eviction policy:** LRU
- **cache 2:** A void range-cache of size 8192, large enough so all misses are cold misses.
- **cache 3:** A fully associative cache used to store texture blocks.
  - **eviction policy:** LRU
  - **size:** 4K, 16K, 64K

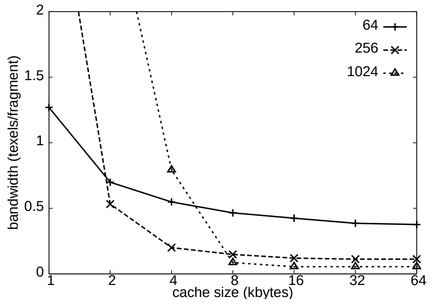
Some results are in Figures A.71 to A.80.

Discussion

Figures A.71 and A.72 show that for sufficiently large caches, a larger block size results in less bandwidth consumption overhead from the index blocks. When the block size is 256, a cache size of at least 2K is sufficient in our tests; a cache size of at least 8K is necessary when the block size is 1024.

Figures A.73 and A.74 show that the reduction in bandwidth due to a larger block size is mostly counteracted by the corresponding increase in bandwidth from

(a)


(b)


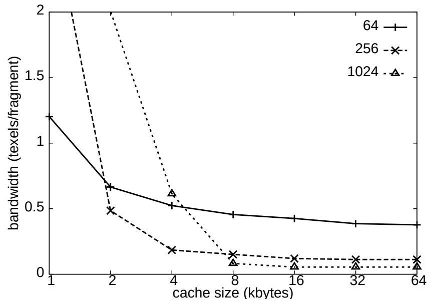
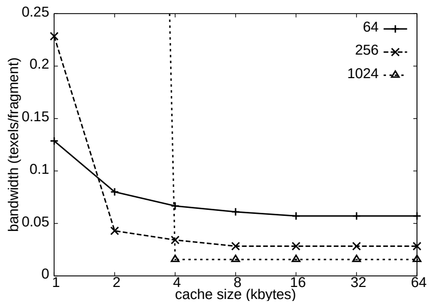
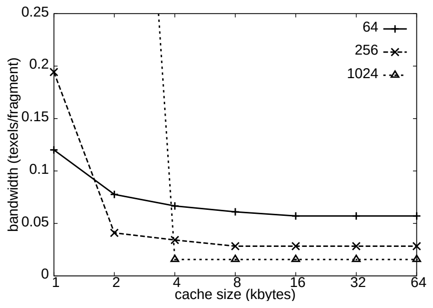
(c)


Figure A.71: The bandwidth consumption due to the index blocks for various block sizes for scene *trifs-90*, texture *dense*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)


(b)


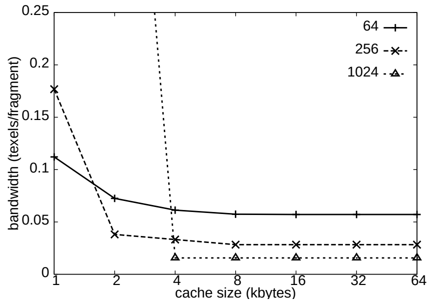
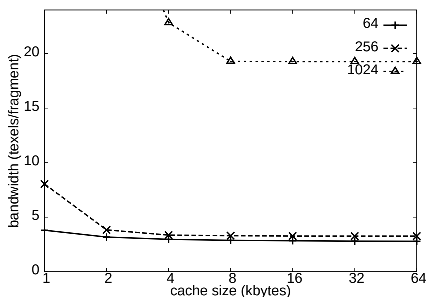
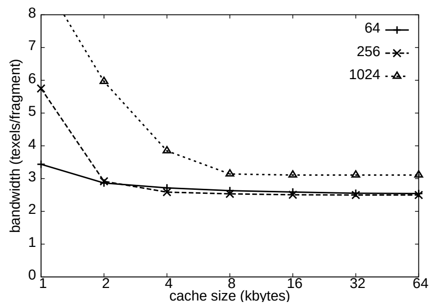
(c)


Figure A.72: The same as in Figure A.71 except that the texture used is *cat*.

(a)


(b)


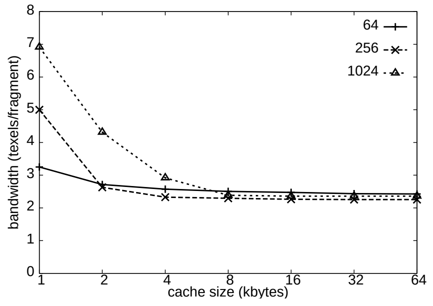
(c)


Figure A.73: The bandwidth consumption due to both the texture and index blocks for various block sizes for scene *trifs-90*, texture *dense*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

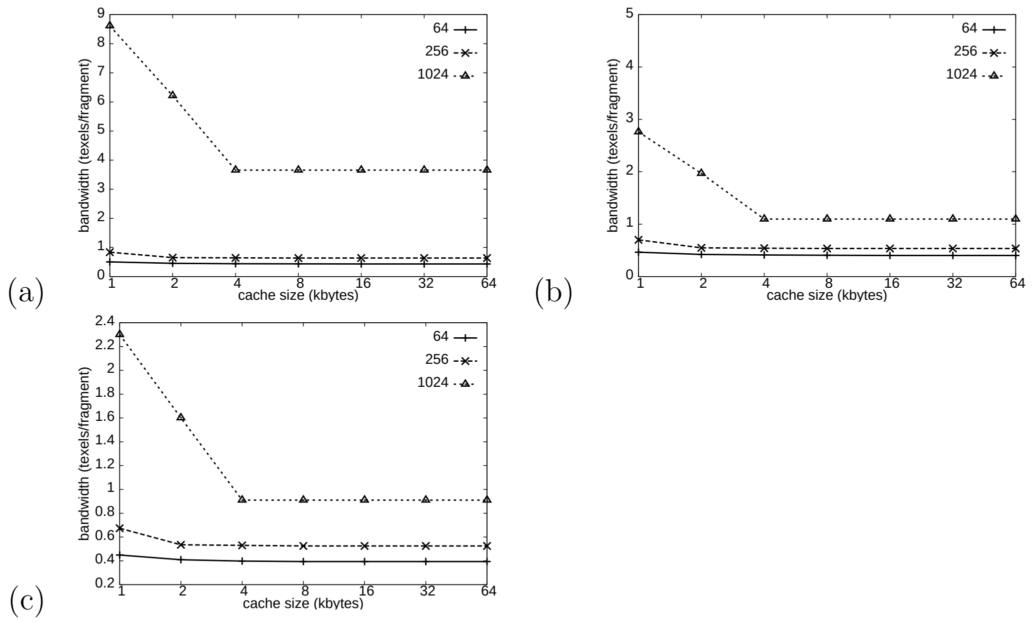


Figure A.74: This is the same as in Figure A.73 except that the texture used is *cat*.

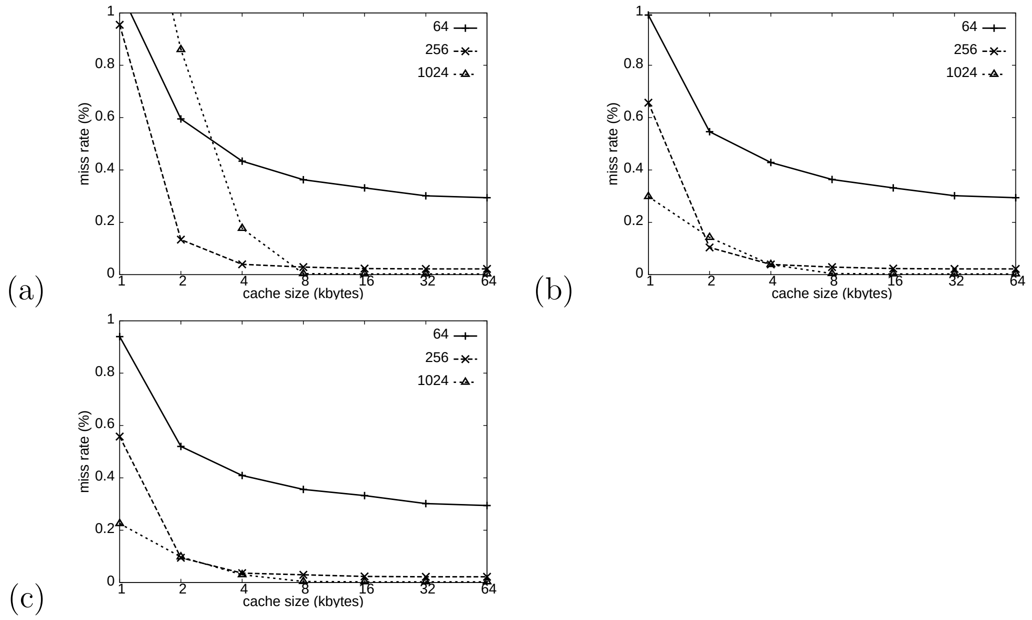
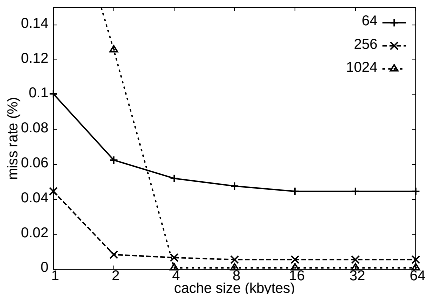
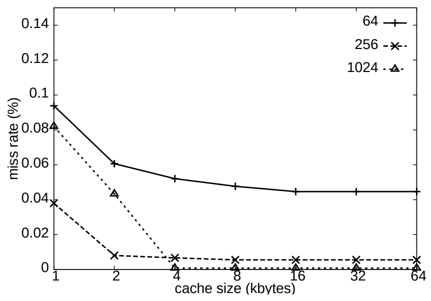


Figure A.75: The miss rate of the index block cache for various block sizes for scene *trifs-90*, texture *dense*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)


(b)


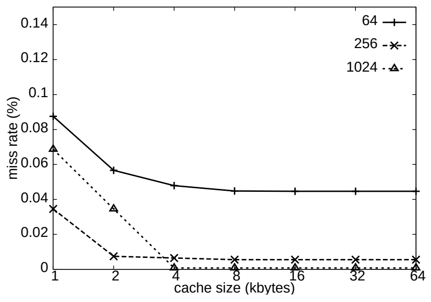
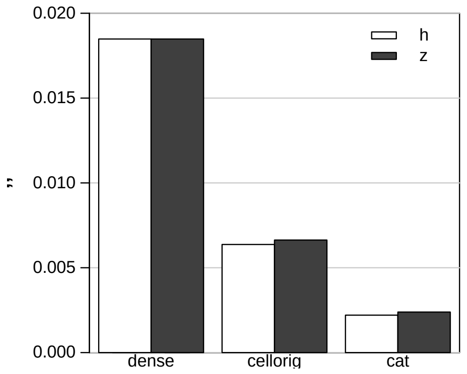
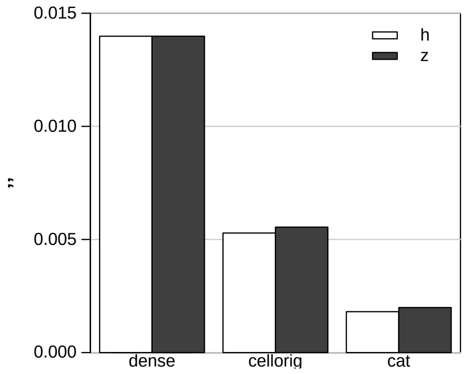
(c)


Figure A.76: This is the same as in Figure A.71 except that the texture used is *cat*.

(a)


(b)


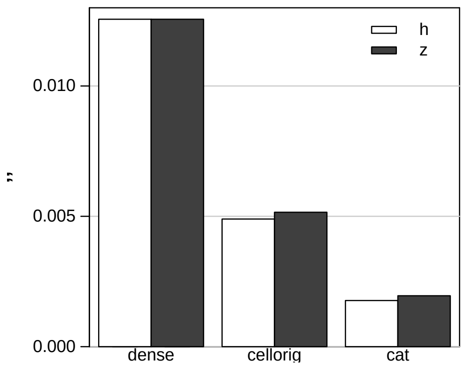
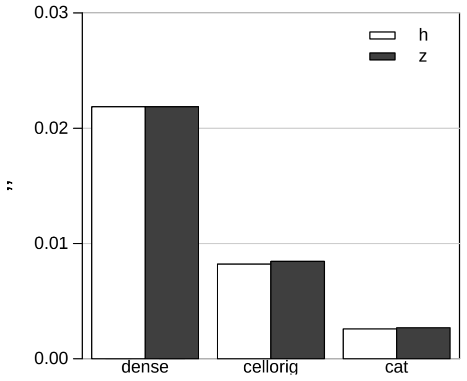
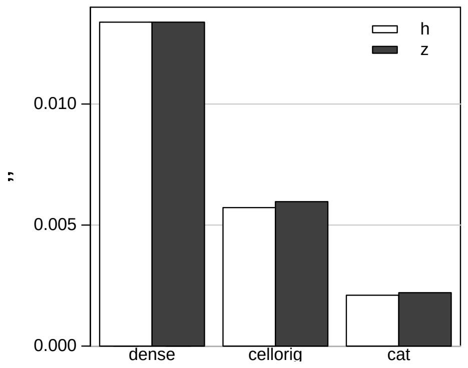
(c)


Figure A.77: Comparison of the average number of index blocks accessed per texel access for index schemes z and h. The block size is 256 and the scene is *trifs-90*. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)



(b)



(c)

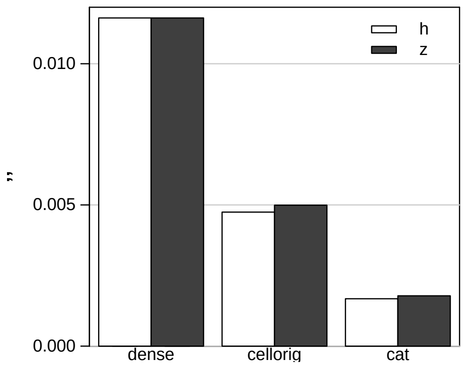
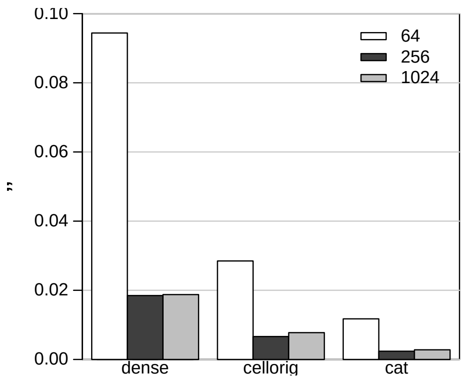
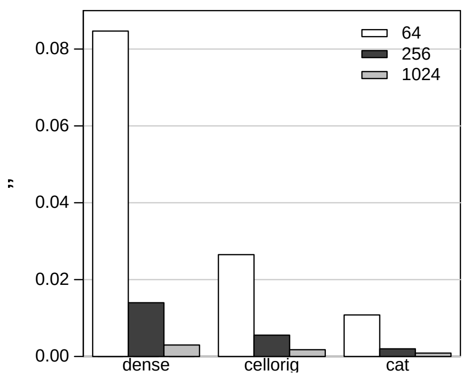


Figure A.78: Comparison of the average number of index blocks accessed per texel access for index schemes z and h. The block size is 256 and the scene is *teapot*. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)



(b)



(c)

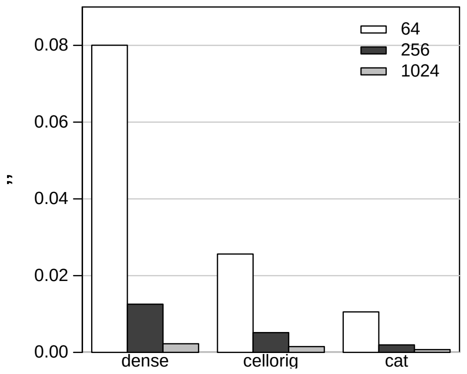
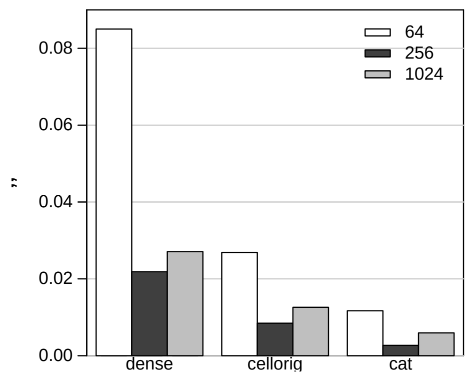
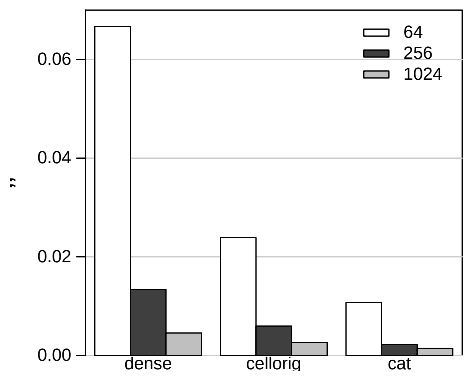


Figure A.79: The average number of index blocks accessed per texel access for various block sizes for scene *trifs-90*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)


(b)


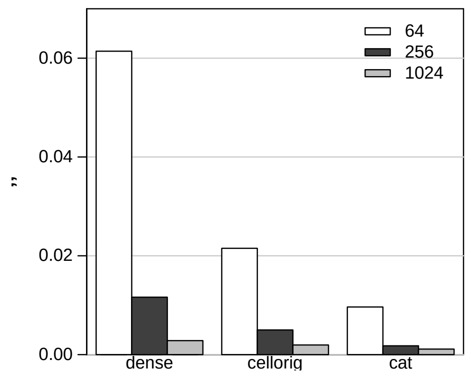
(c)


Figure A.80: The average number of index blocks accessed per texel access for various block sizes for scene *teapot*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

the texture blocks. However, for block size 256 the total bandwidth curve is still close to the one for block size 64, sitting slightly above for the highest texture block miss rate and slightly below for the lower ones when the cache size is at least 2K. Increasing the block size from 256 to 1024 clearly does not decrease the bandwidth consumption in our tests.

One measure of the latency due to going through the B-tree is the average number of index blocks accessed per texel access. Figures A.77 and A.78 show that there is little difference in latency between using index scheme h and using index scheme z. Figures A.79 and A.80 gives the latency for various block sizes. The number when the block size is 64 is quite high, indicating that we should avoid using such a small block size if latency of access is an issue.

A.7.3 Test: Eviction Policy

We next test how the eviction policy of the index block cache affects the bandwidth consumption. The test parameters are the same as the previous test except that some parameters are restricted and we add FIFO to the eviction policies tested for the index block cache.

scenes: trifs-90, trifs-0, teapot

textures: dense, check, cat, cellorig

rasterization order: R, H – we do not test Z because it is so similar to H in all previous tests.

index schemes: r, z, h

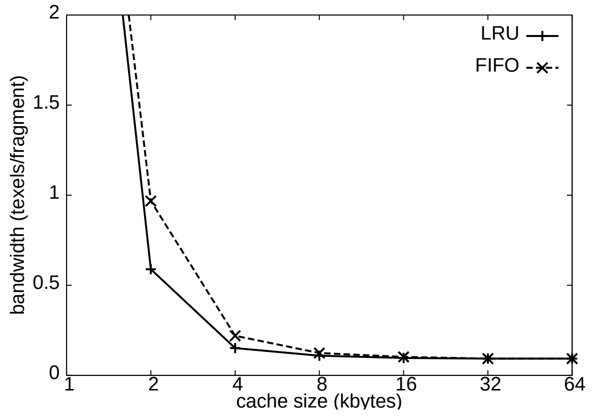
sampling: trilinear

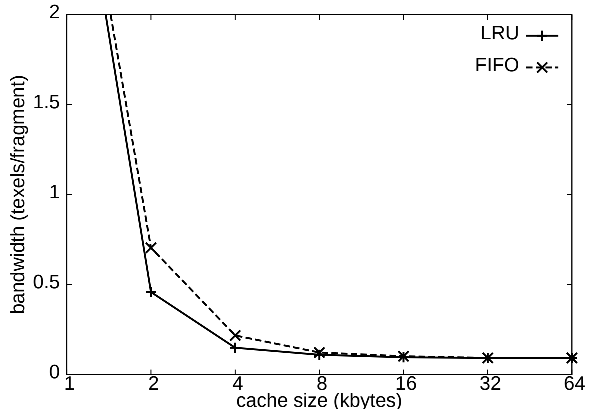
block size: 256 – we do not test block size 1024 because it results in overly high bandwidth consumption, and we do not test block size 64 because using it results in high latency of texel access

cache design:

- **cache 1:** One cache for index blocks (I blocks).
  - **associativity:** full
  - **eviction policy:** LRU, FIFO
- **cache 2:** A void range-cache of size 8192, large enough so all misses are cold misses.
- **cache 3:** A fully associative cache used to store texture blocks.
  - **eviction policy:** LRU
  - **size:** 16K

Some results are in Figures A.81 to A.84.

(a)


(b)


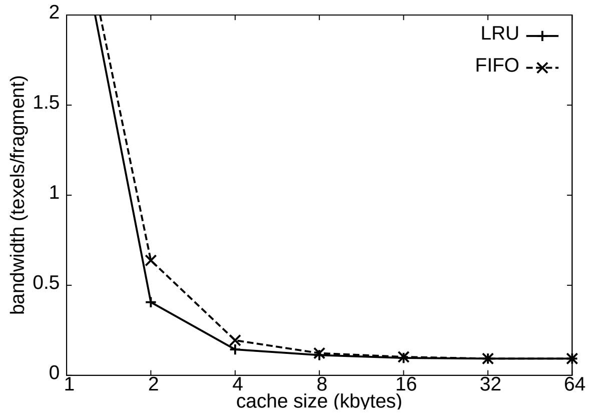
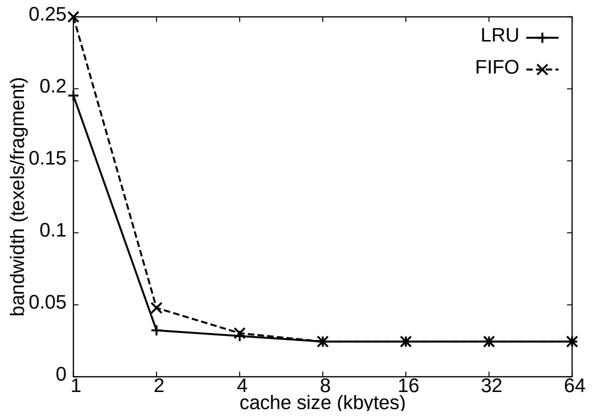
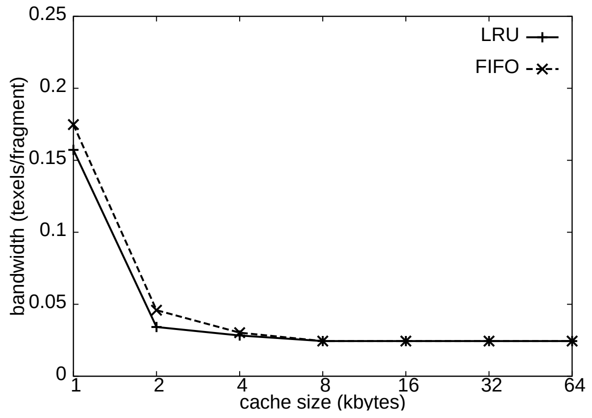
(c)


Figure A.81: The bandwidth consumption due to the index blocks as the eviction policy changes. The curves are for block size 256, scene *trifs-90*, texture *dense*, rasterization order H, and index scheme h. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)


(b)


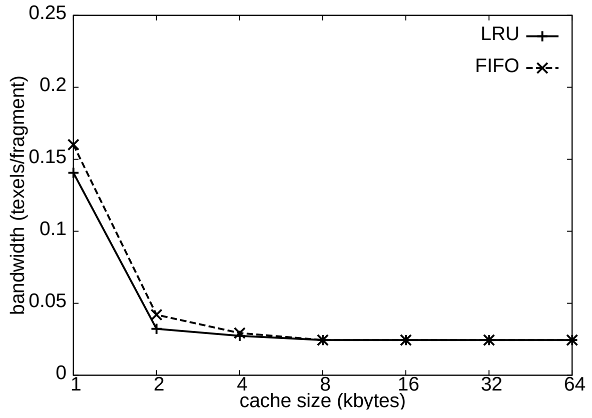
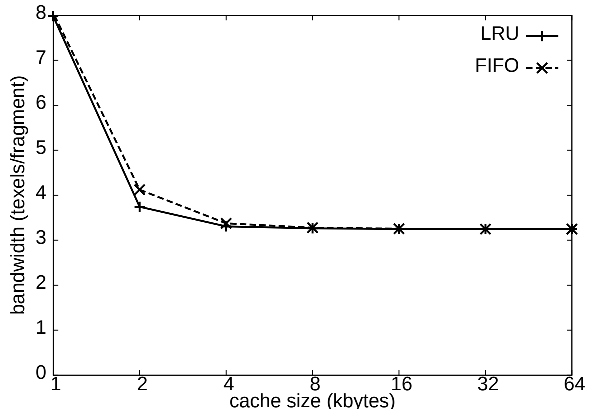
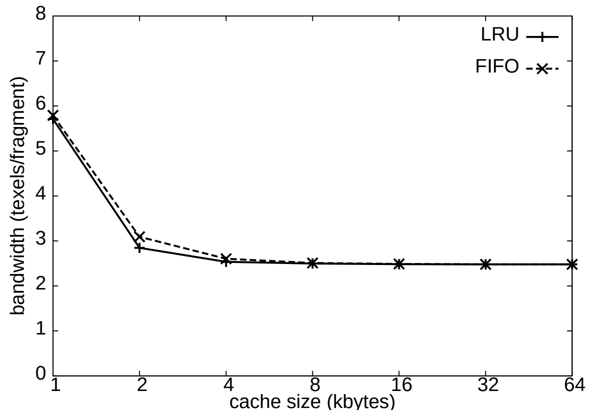
(c)


Figure A.82: This is the same as in Figure A.81 except that the texture used is *cat*.

(a)


(b)


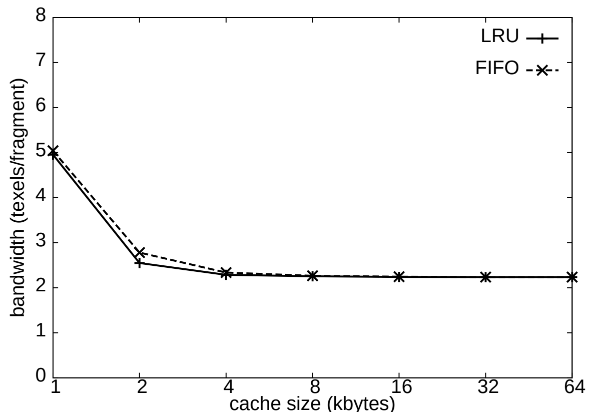
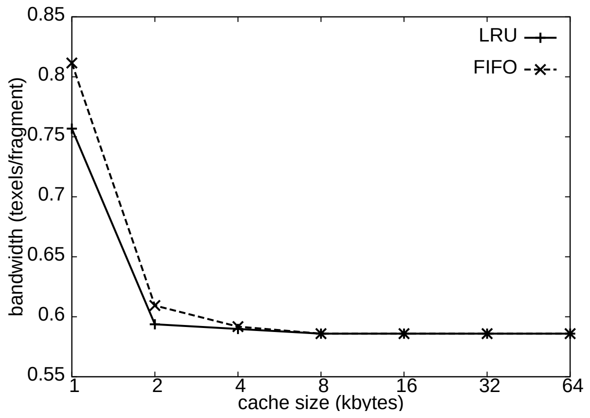
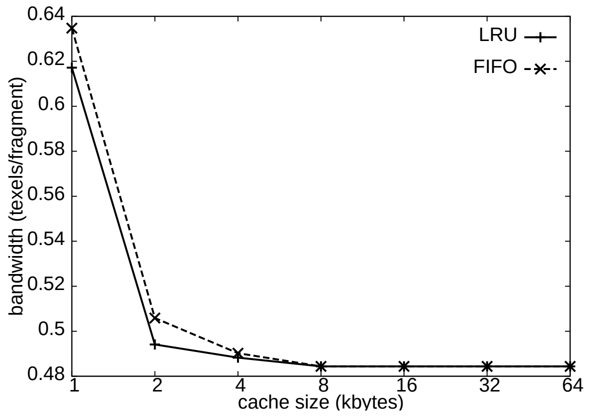
(c)


Figure A.83: The bandwidth consumption due to both the texture and index blocks as the eviction policy changes. The curves are for block size 256, scene *trifs-90*, texture *dense*, rasterization order *H*, and index scheme *h*. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

(a)


(b)


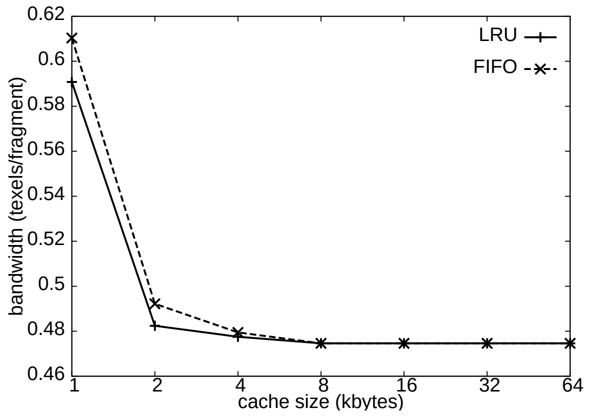
(c)


Figure A.84: This is the same as in Figure A.83 except that the texture used is *cat*.

Discussion

Figures A.81 and A.82 show that using FIFO instead of LRU for the eviction policy only slightly increases the bandwidth consumption due to the index blocks when the cache size is at least 4K in our tests. Figures A.83 and A.84 indicate that when the total bandwidth consumption is taken into account, using FIFO instead of LRU for the eviction policy hardly makes a difference.

A.7.4 Test: Reducing Latency in Searching the Index

We implemented one path-per-MIP-map-level-per-texture-unit caches. Each path cache caches the pointers on the path of a B-tree, and we look up blocks by searching backwards on the path stored in the path cache, then searching forwards. The actual blocks on the path are stored in the index block cache, so what the path cache does is alter the access pattern sent to the index block cache. The test parameters are:

scenes: trifs-90, trifs-0, teapot

textures: dense, check, cat, cellorig

rasterization order: R, H – we do not test Z because it is so similar to H in all previous tests.

index schemes: r, z, h

sampling: trilinear

block size: 64, 256

cache design:

- **cache 1:** One cache for index blocks (I blocks).
  - **associativity:** full
  - **eviction policy:** FIFO
- **cache 2:** A void range-cache of size 8192, large enough so all misses are cold misses.
- **cache 3:** A fully associative cache used to store texture blocks.
  - **size:** 4K, 16K, 64K

Some results are in Figures A.85 to A.92.

Discussion

The use of path-caches has little effect when the index block cache is large but somewhat reduces the bandwidth consumption when the cache size is small, as shown in Figures A.85 to A.88. The reductions in bandwidth consumption are generally modest except for when the block size is 256, the texture is *dense*, and the cache size is less than 2K (Figure A.87). This can be explained by looking at

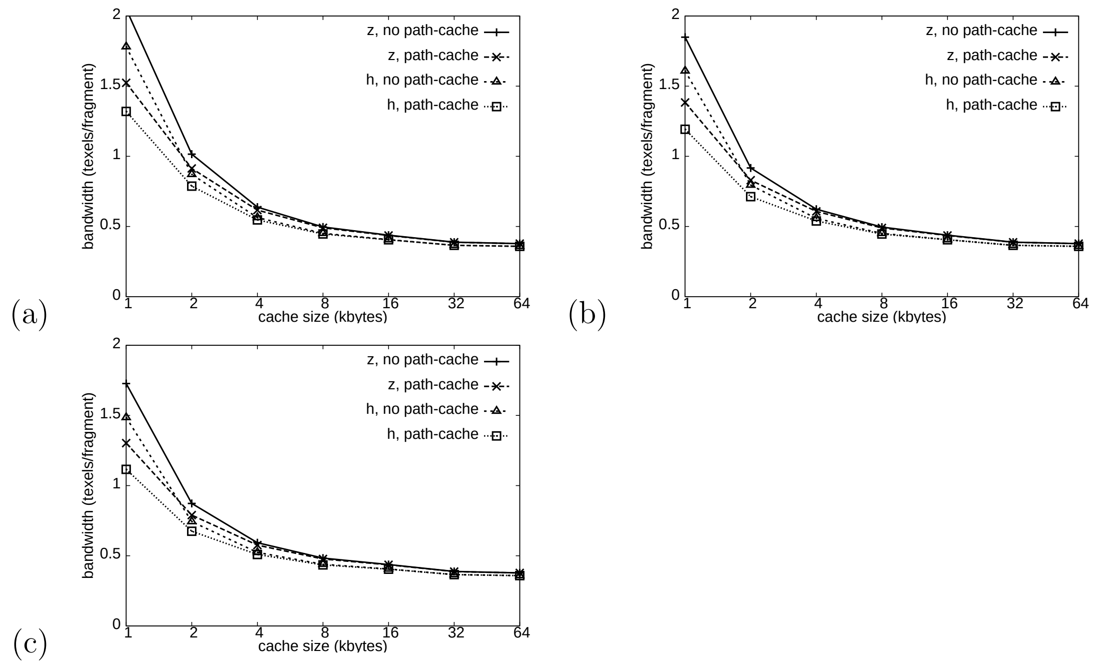
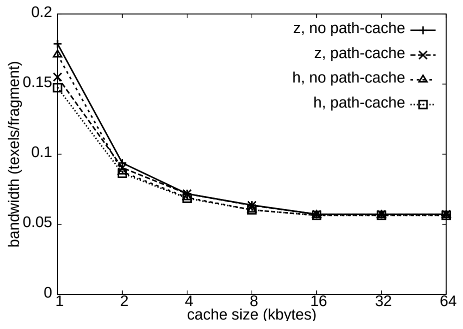
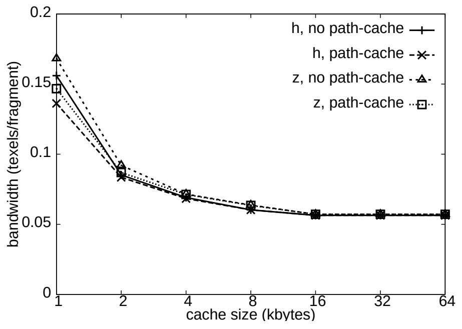


Figure A.85: The bandwidth consumption due to the index blocks for block size 64, scene *trifs-90*, texture *dense*, rasterization order H, and index schemes h and z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

the height for the B-tree for the 512×512 level of each texture in Figure 6.12 (the scene *trifs-90* is set up to sample from the 512×512 and 256×256 levels).

For texture *dense*, the B-tree height is 3 for block size 256 and 5 for block size 64; the corresponding numbers for texture *cat* are 2 and 4. When the block size is 64 a cache size of 1024 can contain 16 entries which is enough to contain a path down the B-tree for the 512×512 and 256×256 levels for both textures *dense* and *cat*. The cache can only contain 4 entries when the block size is 256 which is insufficient to contain two B-tree paths when the texture is *dense* but sufficient when the texture is *cat*. This leads to very high bandwidth consumption when

(a)


(b)


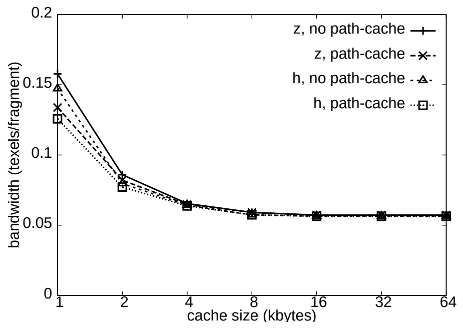
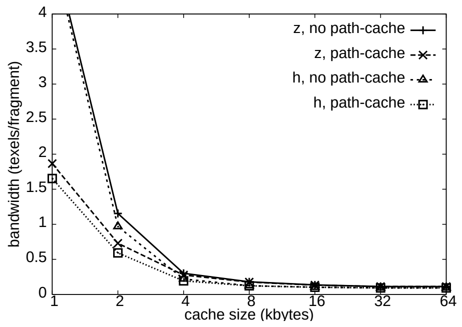
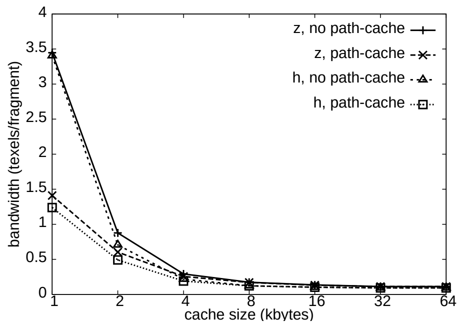
(c)


Figure A.86: This is the same as in Figure A.85 except that the texture used is *cat*.

path-caches are not used in our bad test case (Figure A.87). From this we expect the use of path-caches to be most effective at reducing bandwidth consumption from the index blocks when the cache is not large enough to contain the paths down B-trees that are involved in rendering the scene.

Figures A.89 to A.91 demonstrate that using path-caches greatly decreases latency of access if there is good spatial locality in the sequence of block keys in B-tree key-space. When the locality of the sequence of block keys in key-space is poor as in Figure A.92 where the parameters of scene *trifs-90*, rasterization order *R*, and index scheme *r* were chosen for that property, the benefits are reduced. In

(a) 

(b) 

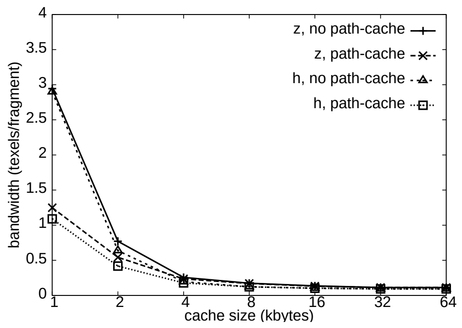
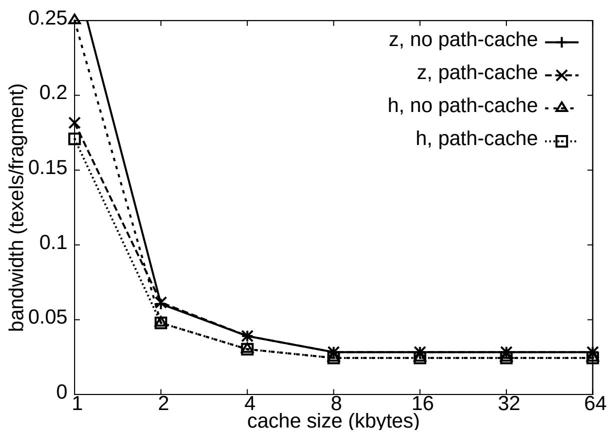
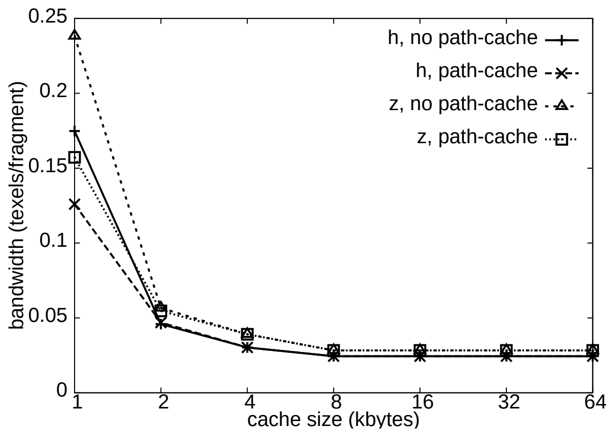
(c) 

Figure A.87: The bandwidth consumption due to the index blocks for block size 256, scene *trifs-90*, texture *dense*, rasterization order H, and index schemes h and z. The texture block cache size is (a) 4K (b) 16K and (c) 64K.

the case where additionally the block size is small (64 bytes) and the path long, using a path-cache actually produces greater latency than not using one. However, the poor spatial locality necessary for this is unlikely to occur when the rendering parameters are chosen properly.

(a)


(b)


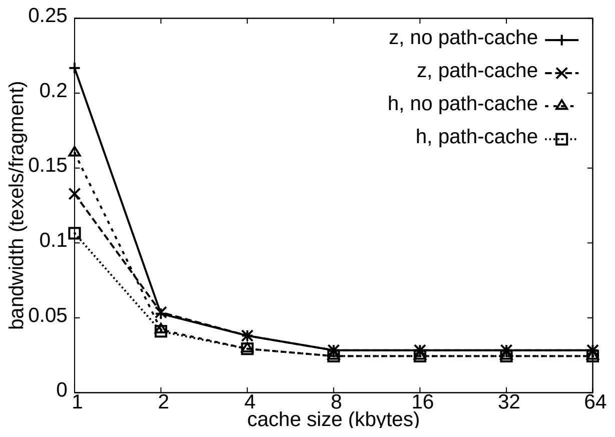
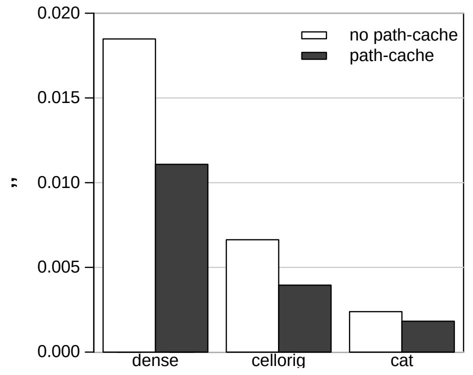
(c)


Figure A.88: This is the same as in Figure A.87 except that the texture used is *cat*.

(a) 

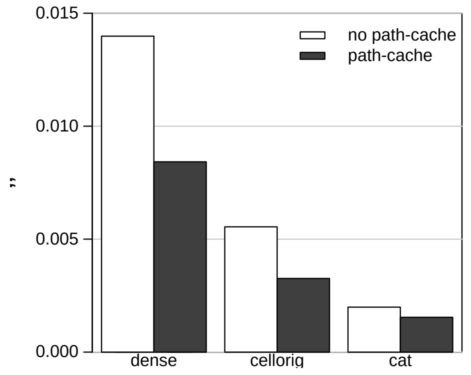
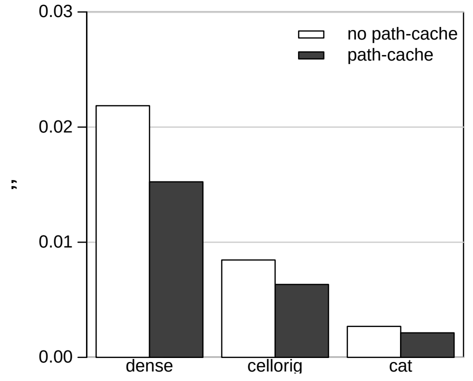
(b) 

Figure A.89: The average number of index blocks accessed per texel access for block size 256, scene *trifs-90*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K and (b) 16K.

(a) 

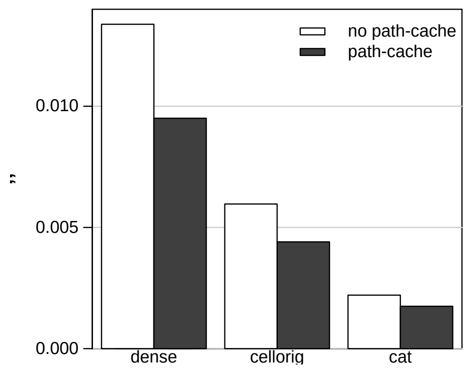
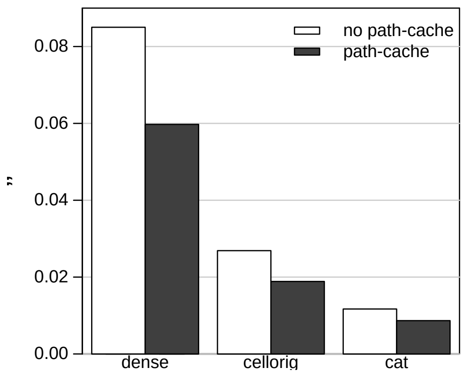
(b) 

Figure A.90: The average number of index blocks accessed per texel access for block size 256, scene *teapot*, rasterization order H, and index scheme z. The texture block cache size is (a) 4K and (b) 16K.

(a) 

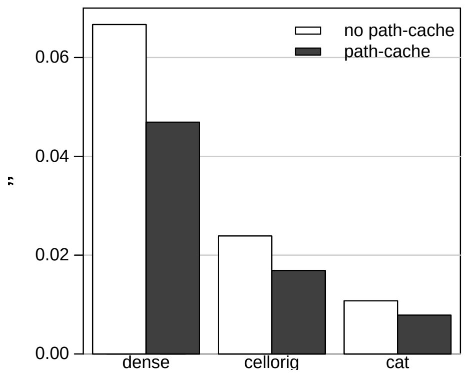
(b) 

Figure A.91: The same as Figure A.90 except for block size 64.

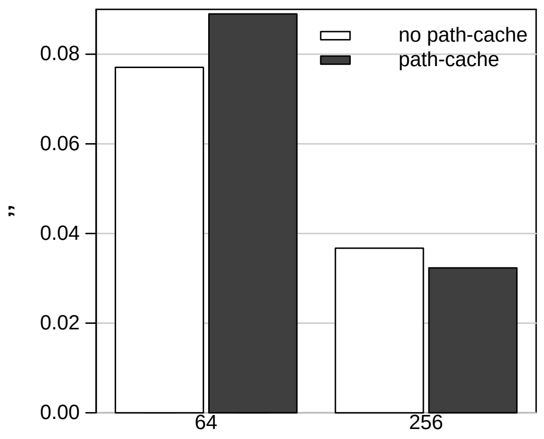


Figure A.92: A case where poor spatial locality makes the use of a path-cache less effective at reducing latency. The average number of index blocks accessed per texel access for texture block cache size of 16K, scene *trifs-90*, texture *dense*, rasterization order H, and index scheme r. The block sizes are 64 and 256.

Bibliography

[BAC96] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha, *Rendering from compressed textures*, SIGGRAPH 1996, Computer Graphics Proceedings, 1996, pp. 373–378.

[BD02] David Benson and Joel Davis, *Octree textures*, SIGGRAPH 2002, Computer Graphics Proceedings, 2002, pp. 785–790.

[BIP00] Chandrajit L. Bajaj, Insung Ihm, and Sanghun Park, *Compression-based 3D texture mapping for real-time rendering*, Graphical Models **62** (2000), no. 6, 391–410.

[Bli96] Jim Blinn, *Jim Blinn's corner: A trip down the graphics pipeline*, Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996.

[CBS98] Michael Cox, Narendra Bhandari, and Michael Shantz, *Multi-level tex-*

ture caching for 3D graphics hardware, Proceedings of the 25th Symposium on Computer Architecture, 1998.

[CDF$^+$86] Graham Campbell, Thomas A. DeFanti, Jeff Frederiksen, Stephen A. Joyce, and Lawrence A. Leske, *Two bit/pixel full color encoding*, SIGGRAPH 1986, Computer Graphics Proceedings, 1986, pp. 215–223.

[CH02] Nathan A. Carr and John C. Hart, *Meshed atlases for real-time procedural solid texturing*, ACM Transactions on Graphics **21** (2002), no. 2, 106–131.

[CHM99] Nate Carr, John Hart, and Jerome Maillot, *The solid map: Methods for generating a 2-D texture map for solid texturing*, 1999.

[Den68] P. J. Denning, *The working set model for program behaviour*, Communications of the ACM **11** (1968), no. 5, 323–333.

[EMP$^+$94] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley, *Texturing and modeling: A procedural approach*, Academic Press, October 1994, ISBN 0-12-228760-6.

[Fen03] Simon Fenney, *Texture compression using low-frequency signal modulation*, Proceedings of the 2003 SIGGRAPH / Eurographics Workshop on Graphics Hardware, 2003, pp. 84–91.

[FFBG01] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg, *Adaptive shadow maps*, SIGGRAPH 2001, Computer Graphics Proceedings, 2001, pp. 387–390.

[Fou95] A. Fournier, *Wavelets and their applications in computer graphics*, SIGGRAPH 1995 Course Notes, vol. 26, 1995.

[FPRJ00] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones, *Adaptively sampled distance fields: A general representation of shape for computer graphics*, SIGGRAPH 2000, Computer Graphics Proceedings, 2000, pp. 249–254.

[FvFH90] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer graphics: Principles and practice*, 2nd ed., Addison-Wesley, 1990.

[GG91] A. Gersho and R. Gray, *Vector quantization and signal compression*, Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[Gla95] Andrew S. Glassner, *Principles of digital image synthesis*, vol. 1, Morgan Kaufmann, San Francisco, CA, 1995.

[HD02] John C. Hart and Peter K. Doenges, *A framework for analyzing real-time advanced shading techniques*, SIGGRAPH Course Notes, vol. 36, 2002.

[HDKS00] Wolfgang Heidrich, Katja Daubert, Jan Kautz, and Hans-Peter Seidel, *Illuminating micro geometry based on precomputed visibility*, SIGGRAPH 2000, Computer Graphics Proceedings, 2000, pp. 455–464.

[Hec86] Paul S. Heckbert, *Survey of texture mapping*, IEEE Computer Graphics and Applications, 1986.

[Hei99] Wolfgang Heidrich, *High-quality shading and lighting for hardware-accelerated rendering*, Ph.D. thesis, University of Erlangen, Computer Graphics Group, 1999.

[HG97] Ziyad S. Hakura and Anoop Gupta, *The design and analysis of a cache architecture for texture mapping*, Proceedings of ISCA 97: International Symposium on Computer Architecture, 1997, pp. 108–120.

[HP03] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach, 3rd ed.*, Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2003.

[IEH99] Homan Igehy, Matthew Eldridge, and Pat Hanrahan, *Parallel texture caching*, Proceedings of the 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1999, pp. 95–106.

[IEP98] H. Igehy, M. Eldridge, and K. Proudfoot, *Prefetching in a texture cache* *architecture*, Proceedings of the 1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware, 1998, pp. 133–142.

[Ige00] Homan Igehy, *Scalable graphics architectures: Interface and texture*, Ph.D. thesis, Stanford University, 2000.

[Inc98] S3 Incorporated, *S3TC DirectX 6.0 standard texture compression*, 1998.

[KE02] Martin Kraus and Thomas Ertl, *Adaptive texture maps*, Proceedings of the 2002 SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2002, pp. 7–15.

[KSKS96] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer, *Hardware for superior texture performance*, Computers & Graphics **20** (1996), no. 4, 475–481.

[Lan91] Robert C. Landsdale, *Texture mapping and resampling for computer graphics*, Master's thesis, University of Toronto, 1991.

[LD91] Harry R. Lewis and Larry Denenberg, *Data structures and their algorithms*, Harper Collins Publishers, New York, 1991.

[Ma02] Vincent Ma, *Low latency photon mapping using block hashing*, Master's thesis, University of Waterloo, 2002.

[McC00] M. McCool, *SMASH: A next-generation API for programmable graphics accelerators*, Tech. Report CS-2000-14, University of Waterloo, 2000.

[MFPJ99] Joel McCormack, Keith I. Farkas, Ronald Perry, and Norman P. Jouppi, *Simple and table FELINE: Fast elliptical lines for anisotropic texture mapping*, Tech. Report 99.1, Compaq Computer Corporation, Western Research Laboratory, Palo Alto, California, 1999.

[MGW01] Tom Malzender, Dan Gelb, and Hans Wolters, *Polynomial texture maps*, SIGGRAPH 2001, Computer Graphics Proceedings, 2001, pp. 519–528.

[MH99] Michael D. McCool and Wolfgang Heidrich, *Texture shaders*, Proceedings of the 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1999, pp. 117–126.

[MM00] Joel McCormack and Robert McNamara, *Tiled polygon traversal using half-plane edge functions*, Proceedings of the 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware, 2000, pp. 15–21.

[MM02] Vincent C.H. Ma and Michael D. McCool, *Low latency photon mapping via block hashing*, Proceedings of the 2002 SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2002, pp. 89–98.

[Mol95] Steven Molnar, *The PixelFlow texture and image subsystem*, Proceedings of the 1995 SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1995, pp. 3–13.

[MWM01] Michael McCool, Chris Wales, and Kevin Moule, *Incremental and hierarchical Hilbert order edge equation polygon rasterization*, Proceedings of the 2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2001, pp. 65–72.

[OG97] Marc Olano and Trey Greer, *Triangle scan conversion using 2D homogeneous coordinates*, Proceedings of the 1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware (New York City, NY) (Steven Molnar and Bengt-Olaf Schneider, eds.), ACM Press, 1997, pp. 89–96.

[Ola98] Mark Olano, *A programmable pipeline for graphics hardware*, Ph.D. thesis, University of North Carolina, 1998.

[OM84] J. A. Orenstein and T. H. Merrett, *A class of data structures for associative searching*, Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984.

[Owe02] John Owens, *Computer graphics on a stream architecture*, Ph.D. thesis, Stanford, 2002.

[Per99] A. Pereberin, *Hierarchical approach for texture compression*, Proceedings of GraphiCon '99, 1999, pp. 195–199.

[SA02] Mark Segal and Kurt Akeley, *The OpenGL graphics system: A specification (version 1.4)*, 2002.

[Sho38] I.J. Shoenberg, *On the Peano curve of Lebesgue*, Bulletin of the American Mathematical Society **44** (1938), no. 519.

[Tan92] Andrew S. Tanenbaum, *Modern operating systems*, Prentice Hall, 1992.

[Wei01] Li-Yi Wei, *Texture synthesis by fixed neighborhood searching*, Ph.D. thesis, Stanford, 2001.

[Wil83] Lance Williams, *Pyramidal parametrics*, SIGGRAPH 1983, Computer Graphics Proceedings, 1983, pp. 1–11.