01 THE IMPLEMENTATION OF A SCALABLE TEXTURE CACHE

Matt Pharr

5 March 2017

Abstract

Texture caching—loading rectangular tiles of texture maps on demand and keeping a fixed number of them cached in memory—has proven to be an effective approach for limiting memory use when rendering complex scenes; scenes with gigabytes of texture can often be rendered with just tens or hundreds of megabytes of texture resident in memory. However, implementing an efficient texture cache is challenging in the presence of tens of rendering threads; each traced ray generally performs multiple cache lookups, one for each texel accessed during texture filtering, all while new tiles are being added to the cache and old tiles are recycled. Traditional thread synchronization approaches based on mutexes give poor parallel scalability with this workload.

We present a texture cache implementation based on lock-free algorithms and the "read-copy update" technique, and show that it scales linearly up to (at least) 32 threads. Our implementation performs just as well as `pbrt` *does with preloaded textures where there are no such cache maintenance complexities.*

Note: This document describes new functionality that may be included in a future version of `pbrt`. The implementation is available in the `texcache` branch in the `pbrt-v3` repository on github. We're interested in feedback (especially bug reports) about our implementation; please send a note to authors@pbrt.org or post to the pbrt Google Groups list if you have comments or questions. (Given that the implementation does some fairly tricky things to minimize locking and inter-thread coordination, it is entirely possible that subtle bugs remain.)

1.1 INTRODUCTION

Most visually complex computer graphics scenes make extensive use of texture maps to represent variation in surface appearance. Scenes used in movie production often have hundreds of gigabytes of texture data—much more than can fit into system memory on most computers. Therefore, production renderers generally use a *texture cache* that keeps a relatively small fixed amount of recently-accessed texture data in memory (e.g. tens or hundreds of megabytes), making it possible to render such scenes using a reasonable amount of memory.

Texture caching originated in Pixar's RenderMan renderer. Peachey's technical memo (1990) describing the idea remains an excellent reference. The key insight that explains why texture caching works well is the *Principle of Texture Thrift*, which Peachey credits to Pat Hanrahan:

Given a scene consisting of textured 3D surfaces, the amount of texture information minimally required to render an image of the scene is proportional to the resolution of the image and is independent of the number of surfaces and the size of the textures.

To see why this principle is valid, first recall that texture filtering algorithms like trilinear filtering and EWA access a bounded number of texels for each filtered value, regardless of the image map's resolution. Further, in a path tracer, the number of points where texture lookups are performed is proportional to the total number of image samples times the average path length; it isn't related to the scene's geometric complexity. Together, these bound the amount of texture data required.

As an extreme example of this principle, consider a quadrilateral with an 8k $\times$ 8k texture map, but where the quad only covers one pixel's area in the image; we would expect the texture filtering code to only use the highest MIP map level to compute its color—thus, only a single texel is needed, in spite of the texture's full resolution.

Not only is the number of texels needed proportional to the image (not texture) resolution, but texture accesses also exhibit good spatial and temporal locality, which allows small caches of texture data to be effective. Spatial locality comes as a wonderful side-effect of high-quality texture filtering for anti-aliasing. Consider rendering a 1024 $\times$ 1024 image of a quadrilateral that just fills the screen using one sample per pixel: because MIP-mapping matches the texture's frequency content to the rate at which it is being sampled, we expect that even if a 4096 $\times$ 4096 texture map is applied to the quad, all texel accesses will still come from the MIP pyramid level with 1024 $\times$ 1024 resolution; no other levels are needed. In general, adjacent image samples will access adjacent (and partially overlapping) texels. Accesses to texels will be dense, without gaps between them—if there were gaps, then aliasing would result, since gaps would imply that the texel sampling rate was lower than the frequency content of the texels.

As far as temporal locality, consider how the ray tracer would access texels in the texture map for the full-screen quad scene: recall that with most `Integrator`s, pbrt renders images in 16×16 pixel tiles, scanning across the image from left-to-right, top-to-bottom. If we consider the resulting texel access patterns, we can see that after its first access, a given texel will likely be accessed again a few more times in the near future, and possibly again during the next line of buckets, but then never again. Thus, we can have a sense that storing a recently-accessed subset of the texture will be effective.

Of course, these examples are ideal scenarios for texture caching; in practice, access patterns will generally be less regular due to objects being partially occluded by other objects, rays following multiple bounces for global illumination, textures reused in multiple parts of the scene, and so forth. In the following, we'll see that calculating ray differentials for indirect rays of all types—not just specular, but diffuse and glossy as well—is critical for maintaining texture spatial coherence so that the texture cache performs well.

In the following, we will see that texture caching gives three main advantages:

- Faster startup: MIP maps are computed ahead of time, and no texels need to be read from disk as the scene is being initialized.
- Loading only what is needed: texture that is never used isn't read into memory; such texture comes from both hidden geometry and MIP levels that aren't accessed.
- Limited memory use: thanks to locality of reference, a small texture cache usually performs well.

The main challenge in implementing a texture cache is to do all this efficiently even with tens (or more) of rendering threads—many texture cache lookups are generally performed for each ray, while texture tiles are also being added to and removed from the cache concurrently. Implementing a texture cache that provides good parallel scalability—performance that scales well with the number of threads used—requires careful attention to minimizing points of mutual exclusion among the threads.

1.1.1 USING CACHED TEXTURES IN PBRT

Using cached textures when rendering with `pbrt` is fairly straightforward. First, convert preexisting textures that you'd like to have managed with the texture cache to `pbrt`'s custom texture format using the `imgtool` utility:

```
% imgtool maketiled tex.png tex.txp
```

`pbrt` uses the "txp" extension for its cached texture format; these files not only include all of the MIP map levels, but also have the images arranged in rectangular tiles for the texture cache. `imgtool maketiled` takes an optional `--wrap` parameter to specify how to handle out-of-bounds texel accesses while resampling the image to create the MIP maps; options include "clamp", "repeat", and "black".

Next, modify your `pbrt` scene files to update the filenames referring to these textures with the new "txp" ones. (Note: only "image" textures support texture caching; image maps used for HDR environment maps, projection light sources, etc., must be one of the regular formats supported by `pbrt`—EXR, PFM, PNG, or TGA.)

By default, `pbrt` uses a 96 MB texture cache. The `--texcachemb` command line option can be used to specify a different size.

1.2 MODIFICATIONS ELSEWHERE IN PBRT

In order to keep this document focused on the texture cache implementation, we'll give an overview of the changes elsewhere in `pbrt`'s texture-related code that made it possible to add a texture cache, but won't include their implementations here.

1.2.1 IMAGE CLASS

The `Image` class, also a new addition in the `texcache` branch, stores 2D images in memory, using efficient 8- and 16- bit encodings when applicable. (For example, when an 8-bit PNG image is used, only a single byte is used for each image channel. It can be found in the files `core/image.h` and `core/image.cpp`. In contrast, `pbrt-v3` and previous used 32-bit `float`s to store all image data.) When a particular texel in an `Image` is accessed, its value is converted to a `Float` or `Spectrum`, depending on the caller's request.

The `Image` class implementation isn't directly used in the texture cache code we will describe in this document, but there are two related additions that are used. First, the `PixelFormat` enumeration records the format of in-memory image data. For example, `RGB8` corresponds to 8-bit-per-channel RGB data, linearly mapped from $[0, 255]$ to $[0, 1]$. `SRGB8` is the same, but where texels are also encoded with the SRGB gamma curve. The "Y" formats encode just a single channel.

⟨*PixelFormat Declarations*⟩ ≡
```
enum class PixelFormat { SY8, Y8, RGB8, SRGB8, Y16, RGB16, Y32, RGB32 };
```

A utility function, `ConvertTexel()`, converts a texel in memory in a given pixel format to either a `Float` or a `Spectrum`. (One channel formats are replicated across channels if the target is a `Spectrum`, and RGB formats are averaged across their three channels if the target is a `Float`.)

⟨*Image Function Declarations*⟩ ≡
```
template <typename T> T ConvertTexel(const void *texel, PixelFormat format);
```

1.2.2 MIP MAP AND TEXTURE INTERFACES

We have updated `pbrt`'s texture filtering defaults: now, bilinear filtering of four texels from the more detailed of the nearest MIP levels for the point being shaded's filter footprint is the default, while before EWA filtering was the default. A new parameter to `ImageTexture`, "filter", can be used to control which filtering algorithm to use: valid options are "point", "bilinear", "trilinear", and "EWA". For path tracing, where many samples per pixel are usually taken, the expense of higher-quality texture filtering algorithms like EWA isn't worthwhile; it's just as well to include texture filtering as part of the rest of the integration for pixels' values.

To support texture caching, we have modified `pbrt`'s `MIPMap` implementation to operate in terms of an abstract `TexelProvider` class, which provides an interface that abstracts away how the texels used by the `MIPMap` are stored and represented. Thus, the `MIPMap` no longer stores the image pyramid directly, but relies on a `TexelProvider` instead. The `TexelProvider` declaration and implementations are in the files `core/mipmap.h` and `core/mipmap.cpp`.

⟨*TexelProvider Declarations*⟩ ≡

```
class TexelProvider {
public:
    ⟨TexelProvider Public Methods⟩
};
```

The `MIPMap` can query how many levels are present in the image pyramid and what the resolution of each level is.

⟨*TexelProvider Public Methods*⟩ ≡

```
virtual int Levels() const = 0;
virtual Point2i LevelResolution(int level) const = 0;
```

The `MIPMap` can also request the value of a texel at a given position in a given level. Remapping out-of-bounds positions based on the image's wrap mode is handled by the `MIPMap`, so `TexelProvider` implementations can assume that the given position is a valid texel coordinate.

⟨*TexelProvider Public Methods*⟩ ≡

```
virtual Float TexelFloat(int level, Point2i p) const = 0;
virtual Spectrum TexelSpectrum(int level, Point2i p) const = 0;
```

In order to continue to support preloaded textures (as is the only option for image maps in `pbrt-v3`), we have written an `ImageTexelProvider`. It stores a vector of `Image`s, one for each level of the pyramid, and passes MIP map requests on to the appropriate `Image`.

We have also implemented a `CachedTexelProvider` that provides a bridge between the `MIPMap` and the texture cache described in this document. Its implementation isn't very interesting; it just forwards requests from the `MIPMap` on to the texture cache.

1.2.3 RAY DIFFERENTIALS

Recall from Section 10.1.3 of the third edition of *Physically Based Rendering* how important ray differentials are for high-quality texture filtering for directly-visible objects and for objects seen via specular reflections. There, the use of ray differentials was driven by an interest in anti-aliasing. Now, given the connection between anti-aliasing and the spatial coherence that texture caching depends on, we can see that ray differentials are also important for texture cache performance. (Section 1.6.5 has measurements that show just how important they are.)

Ideally, `pbrt`'s `BxDF` sampling methods would allow `BxDFs` to return ray differentials for sampled directions. Rather than overhaul all of those interfaces, we'll use a work-around for now: we've added a `SurfaceInteraction::GenerateSpawnedRay()` method that also takes the `BxDFFlags` encoding the type of `BxDF` component that was sampled by `BSDF::Sample_f()` and the surface's index of refraction. It uses these along with the incident ray's differentials to compute approximate differentials for the spawned ray. In turn, the `PathIntegrator` and `VolPathIntegrator` call this method to generate spawned rays. Computing appropriate ray differentials for bidirectional light transport algorithms remains a challenging problem; see the "Further Reading" section for more information.

```
⟨SurfaceInteraction Method Definitions⟩ ≡
  RayDifferential SurfaceInteraction::SpawnRay(const RayDifferential &rayo,
                                              const Vector3f &wi, int bxdfType,
                                              Float eta) const {
      RayDifferential rd = SpawnRay(wi);
      rd.hasDifferentials = true;
      rd.rxOrigin = p + dpdx;
      rd.ryOrigin = p + dpdy;

      if (bxdfType & BSDF_DIFFUSE) {
          ⟨Compute differentials for diffuse ray⟩
      } else {
          ⟨Compute ray differentials for glossy and specular rays⟩
      }
      return rd;
  }
```

Ray differentials for indirect diffuse and glossy rays are computed in an *ad-hoc* manner in the current implementation. We compute differentials for diffuse indirect rays as if they covered roughly 1/25th of the hemisphere, and 1/100th of the hemisphere for glossy. For highly glossy surfaces that are close to perfectly specular, differentials based on perfect specular reflection may be more appropriate. (See the "Further Reading" section for more information on better approaches.) Though our experiments haven't been comprehensive, we have found our current implementation to generally work well.

```
⟨Compute differentials for diffuse ray⟩ ≡
  Vector3f v[2];
  CoordinateSystem(wi, &v[0], &v[1]);
  rd.rxDirection = Normalize(wi + .2f * v[0]);
  rd.ryDirection = Normalize(wi + .2f * v[1]);
```

We won't include the code for glossy rays or for specular rays here (the specular code matches the existing code in `pbrt` for specular ray differentials.)

1.3 TiledImagePyramid IMPLEMENTATION

The `TiledImagePyramid` class takes care of the details of representing image pyramids with fixed-size texture tiles, including converting from pyramids of images stored in scanlines, storing tiled textures on disk, and reading tiles back.[1] It provides a key foundation for the implementation of the texture cache.

⟨*TiledImagePyramid Declarations*⟩ ≡

```
class TiledImagePyramid {
public:
    ⟨TiledImagePyramid Public Methods⟩
    ⟨TiledImagePyramid Public Data⟩
};
```

Its `Create()` method takes an image pyramid with each level specified by an `Image`, rearranges the texels into tiles with width and height equal to the given `tileSize` (which must be a power of 2), and writes them to disk. Because the top levels of image pyramids are generally very low resolution (and thus would occupy a small part of a tile, wasting memory), up to `topLevelsBytes` of them are stored separately in the file header, not arranged into tiles.

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
static bool Create(std::vector<Image> levels, const std::string &filename,
                   WrapMode wrapMode, int tileSize, int topLevelsBytes = 3800);
```

The `TiledImagePyramid::Read()` method reads the metadata from the file header (but not the texels from all of the texture tiles) and initializes the members of the given `TiledImagePyramid`.

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
static bool Read(const std::string &filename, TiledImagePyramid *tex);
```

Some basic information about the texture, including the in-memory format of the texels, the base-2 logarithm of the tile size, and the resolution of each level are stored on disk and returned in the `TiledImagePyramid` returned by `Read()`.

⟨*TiledImagePyramid Public Data*⟩ ≡

```
std::string filename;
PixelFormat pixelFormat;
WrapMode wrapMode;
int logTileSize;
std::vector<Point2i> levelResolution;
```

`levelOffset` gives the offset into the tiled texture file where the tiles for the corresponding level start. Within a level, tiles are laid out left-to-right, top-to-bottom. Within a tile, texels are laid out in scanline order.

1 Because the renderer generally accesses spatially-coherent 2D regions of the texture, storing textures in tiles rather than scanlines means that for a read of a given number of texels from disk, it's more likely that texel accesses in the immediate future will access one of the read texels.

⟨*TiledImagePyramid Public Data*⟩ ≡

```
std::vector<int64_t> levelOffset;
```

Starting at `firstInMemoryLevel`, the texels for that and all subsequent levels are stored in the file header and read from disk by `Read()`. `inMemoryLevels` stores pointers to the texels for these levels, which are stored in scanline order.

⟨*TiledImagePyramid Public Data*⟩ ≡

```
int firstInMemoryLevel;
std::vector<std::unique_ptr<char[]>> inMemoryLevels;
```

Given a `TiledImagePyramid`, a few utility methods provide various values related to the texture.

`TileBytes()` returns the total number of bytes needed to store all of the texels in a single tile of the texture. The `TexelBytes()` utility function, not included here, returns the number of bytes that a single texel in the given format uses (e.g., 3 for an 8-bit RGB texture).

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
size_t TileBytes() const {
    return (1 << logTileSize) * (1 << logTileSize) * TexelBytes(pixelFormat);
}
```

Texture tiles are stored so that each one starts at an integer multiple of `TileDiskAlignment` in the tiled texture file. Aligning them in this way can make I/O operations a bit more efficient. `TileDiskBytes()` therefore gives the number of bytes that each texture tile uses on disk, accounting for this padding (which is assumed to be a power of 2.)

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
size_t TileDiskBytes() const {
    return (TileBytes() + TileDiskAlignment - 1) & ~(TileDiskAlignment - 1);
}
```

⟨*Texture Cache Constants*⟩ ≡

```
static constexpr int TileDiskAlignment = 4096;
```

Given a texture coordinate `p`, `TileIndex()` returns the coordinates for the tile that the point's texel is in. Because tiles are square and have a power-of-2 size in each dimension, a 2D texel coordinate can be converted to a 2D tile coordinate by shifting off a number of low bits equal to the log of the tile size.

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
Point2i TileIndex(Point2i p) const {
    return {p[0] >> logTileSize, p[1] >> logTileSize};
}
```

Given a texture coordinate `p`, `TexelOffset()` gives the offset inside `p`'s tile where its texel is found. Again thanks to power-of-2 sized tiles, we just need to mask off the appropriate number of low bits to get a 2D tile coordinate and then can compute an offset into the tile using the fact that texels are laid out in scanline order inside tiles.

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
int TexelOffset(Point2i p) const {
    int tileMask = (1 << logTileSize) - 1;
    Point2i tilep{p[0] & tileMask, p[1] & tileMask};
    int tileWidth = 1 << logTileSize;
    return TexelBytes(pixelFormat) * (tilep[1] * tileWidth + tilep[0]);
}
```

`FileOffset()` returns the offset in the file where the tile with tile indices given by `p` at level `level` starts.

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
int64_t FileOffset(int level, Point2i p) const {
    int tileWidth = 1 << logTileSize;
    int xTiles = (levelResolution[level][0] + tileWidth - 1) >> logTileSize;
    return levelOffset[level] + TileDiskBytes() * (p[1] * xTiles + p[0]);
}
```

If the texel for the given coordinates is available from the top few pyramid levels that are stored in the file's header and were loaded by the `Read()` method, the `GetTexel()` method returns a pointer to it; otherwise it returns `nullptr`. These levels are stored in scanline order, so the indexing is straightforward given a pointer to the first texel in a level.

⟨*TiledImagePyramid Public Methods*⟩ ≡

```
const char *GetTexel(int level, Point2i p) const {
    if (level < firstInMemoryLevel) return nullptr;
    const char *levelStart = inMemoryLevels[level - firstInMemoryLevel].get();
    return levelStart +
           TexelBytes(pixelFormat) * (p[1] * levelResolution[level][0] + p[0]);
}
```

1.4 A HASH TABLE FOR TEXTURE TILES

Our texture cache stores texture tiles in a hash table that is implemented in the `TileHashTable` class. The hash table is specifically designed to allow concurrent lookups and insertions without requiring mutual exclusion between threads; this is important for good parallel scalability. In this section, we will describe the implementation of those two operations; we'll defer discussion of how tiles are removed from the hash table to Section 1.5.5, which describes the complete tile freeing algorithm.

⟨*TileHashTable Declarations*⟩ ≡

```
class TileHashTable {
public:
    ⟨TileHashTable Public Methods⟩
private:
    ⟨TileHashTable Private Data⟩
};
```

The hash table is a fixed size (and thus should be a bit larger than the number of tiles to be stored in it). For our application, where a fixed amount of memory is allocated for texture tiles and thus the maximum number of tiles is known at startup time, the lack of ability to grow or shrink the hash table isn't a limitation. The "Further Reading" section has pointers to papers about hash table implementations that can grow and shrink even with many threads accessing them concurrently.

⟨*TileHashTable Method Definitions*⟩ ≡

```
TileHashTable::TileHashTable(size_t size) : size(size) {
    table.reset(new std::atomic<TextureTile *>[size]);
    for (size_t i = 0; i < size; ++i)
        table[i] = nullptr;
}
```

Note that hash table elements are `atomic` pointers to texture tiles, not regular pointers. We'll see why this is necessary shortly, in the implementation of the the lookup and insertion methods.

⟨*TileHashTable Private Data*⟩ ≡

```
std::unique_ptr<std::atomic<TextureTile *>[]> table;
const size_t size;
```

The `TextureTile` class represents a single in-memory texture tile. Its primary member variables are a `TileId` that describes which texture, pyramid level, and tile the `TextureTile` represents and a pointer to a fixed amount of memory to store texels for the tile that is provided to the `TextureTile` by the texture cache at startup time.

⟨*TextureTile Declarations*⟩ ≡

```
struct TextureTile {
    ⟨TextureTile Public Methods⟩
    ⟨TextureTile Public Data⟩
};
```

⟨*TextureTile Public Data*⟩ ≡

```
TileId tileId;
char *texels = nullptr;
```

`TileId` packs an integer identifier representing an image pyramid, an integer for the level of the pyramid, and tile coordinates within the level into a single 64-bit member variable, `bits`. Doing so (versus storing them individually) makes hashing and comparisons slightly more efficient in the following.

⟨*TileId Declarations*⟩ ≡
```
struct TileId {
    ⟨TileId Public Methods⟩
private:
    uint64_t bits;
};
```

The `TileId` constructor packs the tile coordinates into `bits`. It also includes a few runtime assertions (not included here) that ensure that the given values won't overflow the 16-bit quantities that they've each been allotted. A default constructor stores an invalid value, which gives us a `TileId` to identify `TextureTile`s that do not refer to valid texture data.

⟨*TileId Public Methods*⟩ ≡
```
TileId() : TileId(0, -1, {0, 0}) {}
TileId(int texId32, int level32, Point2i p32) {
    bits = texId32;
    bits |= (uint64_t(level32) & 0xffff) << 16;
    bits |= uint64_t(p32[0]) << 32;
    bits |= uint64_t(p32[1]) << 48;
}
```

A few straightforward methods unpack elements from `bits` and return the original texture id, pyramid level, and tile coordinates.

⟨*TileId Public Methods*⟩ ≡
```
int texId() const { return bits & 0xffff; }
int level() const { return (bits >> 16) & 0xffff; }
Point2i p() const { return Point2i((bits >> 32) & 0xffff,
                                  (bits >> 48) & 0xffff); }
```

The `hash()` method computes a the value of a hash function based on the `TileId`. An expedient approach would be to used the packed bits directly as a hash value, though results from this would be poor: as a general principle, a good hash function should end up flipping each of the bits in the computed hash with probability 50% if a single bit in the input value changes—clearly not the case for that hash function.

Therefore, we apply the following `MixBits()` function to the packed id to compute a better hash value. It's based on a *bit mixer* function derived by Stafford (2011); through a few shifts, XORs, and multiplies, it computes a hash value that generally has much higher entropy than its input. The various constants in its implementation were derived by searching a parameter space, applying the resulting function to a variety of low-entropy inputs (e.g. sequences of small integers), and looking for values that maximized the overall entropy in the result.

⟨*TileId Public Methods*⟩ ≡
```
size_t hash() const { return MixBits(bits); }
```

⟨*Texture Cache Utility Functions*⟩ ≡

```
inline uint64_t MixBits(uint64_t v) {
    v ^= (v >> 31);
    v *= 0x7fb5d329728ea185;
    v ^= (v >> 27);
    v *= 0x81dadef4bc2dd44d;
    v ^= (v >> 33);
    return v;
}
```

`TileId` also provides comparison operations and an `operator<<` for printing to a C++ output stream; the implementations of these methods are straightforward and not included here.

1.4.1 MEMORY MODELS AND LOCK-FREE SYNCHRONIZATION

For scalability with many threads, it's important that threads be able to perform lookups and insert tiles into the hash table without mutual exclusion and with minimal inter-core communication—if the hash table was protected by a single mutex, for example, performance would be poor, both due to threads being held up by the mutex when another thread was accessing the hash table, as well as due to frequent communication between CPU cores when the mutex was stored in a different core's cache than the one acquiring it. (Recall the discussion of "read for ownership" on p. 1083 of the third edition of *Physically Based Rendering*.)

In order to eliminate any scalability issues due to mutual exclusion, we have implemented hash table lookups and insertion as *lock-free* algorithms that allow multiple threads to perform these operations concurrently without having mutexes or other constructs that prevent threads from accessing the data structure. In particular, insertion is carefully implemented so that other threads concurrently doing lookups always encounter a valid hash table with valid texture tiles. Before going into the implementations of those operations, we'll review some preliminaries.

When thinking about lock-free algorithms, it's important to know that both compilers and CPUs may reorder memory accesses—both reads and writes—from the order in which they appear in program source code. In a single-threaded program, these reorderings are required to be undetectable by the program—the compiler and hardware must not change the semantics of the program.

With multi-threaded execution, providing the same guarantee for the entire program isn't feasible. Synchronization mechanisms like `std::mutex` prevent memory operations from being moved from one side of the mutex to the other; for multi-threaded programs that use mutexes, this usually suffices such that programmers don't need to be aware of memory operation reordering at all. If no explicit thread synchronization is used (as with lock-free algorithms), it's necessary to use the programming language's *memory model* and mechanisms that the language provides for guaranteeing memory ordering in specific instances so that when data is shared between threads, the programmer can ensure that undesired orderings of memory operations don't cause bugs.

For the tile hash table, our needs are easily stated. First, when a `TextureTile *` is added to the hash table, all of the memory writes that initialized the members of the `TextureTile` must be in memory and visible to any other thread that can read that tile's pointer from the hash table. Second, any thread that reads a tile from the hash table must not read from any of the tile's members before loading its pointer.[2]

The C++11 memory model provides semantics that allow us to specify these requirements. The `std::atomic` class has `load()` and `store()` methods that also take a value that indicates a memory ordering to associate with the operation. For a load, `std::memory_order_acquire` ensures that all memory operations after the load in the program are in fact ordered (by both compiler and hardware) after the load. With a store, `std::memory_order_release` ensures that all memory operations in the program before the store are ordered to be before it.

As an example of the use of memory ordering semantics to ensure a reasonable program execution, consider these two variables:

```
int val = 0;
std::atomic<bool> updated{false};
```

Now, assume that one thread updates them as follows:

```
val = 1;
updated.store(true, std::memory_acquire_release);
```

A second thread waits for `updated` to become `true` and then reads `val`, as follows.

```
while (!updated.load(std::memory_order_acquire))
    ;  // spin
assert(val == 1);
```

Because the first thread used release semantics when writing to `updated`, we can be sure that the write to `val` will definitely be visible to the second thread after it sees that `updated` is `true`. Further, because acquire semantics are used for the read of `updated` in the second thread, there's no danger of the compiler moving the read of `val` before the read of `updated`, which otherwise might seem to the compiler to be an innocuous transformation.

Because the two threads in this example had this interaction through the `updated` variable, we say that they have a *synchronizes-with* relationship. These forms of interactions are extremely lightweight—on x86 CPUs, no special instructions are emitted by the compiler for loads and stores with these ordering semantics; the memory semantics only serve the inform the compiler to not reorder memory

2 This second requirement seems like it would obviously always be the case, but some CPU architectures perform *load prediction*, where execution may proceed speculatively with a predicted value from a load, where the load's actualy value is only checked later. Compilers may generate code that uses *value speculation*, which may also reorder loads in this way.

accesses itself across them. (Other architectures do need additional instructions, though they remain reasonably lightweight.)

For accesses to `std::atomic` variables where there are no associated memory ordering requirements, the `std::memory_order_relaxed` semantic can be used.

1.4.2 HASH TABLE LOOKUPS

Onward to the `Lookup()` method. Most of the hash table lookup code is fairly standard, though note the explicit use of acquire semantics for the load from the `TileHashTable::table` array. Assuming the use of release semantics for writes to that array when tile pointers are added to it, `Lookup()` can access the member variables of `entry` without having to worry if the writes to initialize them were reordered after the write of `entry` into the hash table and thus possibly not yet visible to the thread running `Lookup()`.

⟨*TileHashTable Method Definitions*⟩ ≡

```
const char *TileHashTable::Lookup(TileId tileId) const {
    int hashOffset = tileId.hash() % size;
    int step = 1;
    for (;;) {
        const TextureTile *entry =
            table[hashOffset].load(std::memory_order_acquire);
        ⟨Process entry for hash table lookup⟩
    }
}
```

If `entry` is `nullptr`, then the tile isn't in the hash table and `Lookup()` can return. If there is an entry and the `TileId`s match, then we have a hit; after some bookkeeping related to freeing tiles that will be explained in Section 1.5.5, the pointer to the tile's texels can be returned. Otherwise, the next offset to try in the hash table is computed using quadratic probing.

⟨*Process* `entry` *for hash table lookup*⟩ ≡

```
if (entry == nullptr)
    return nullptr;
else if (entry->tileId == tileId) {
    ⟨Update entry's marked field after cache hit⟩
    return entry->texels;
} else {
    hashOffset += step * step;
    ++step;
    if (hashOffset >= size) hashOffset %= size;
}
```

1.4.3 HASH TABLE INSERTION

`TileHashTable::Insert()` adds a texture tile to the hash table. Like `Lookup()`, it is implemented in a lock-free manner, so that it's both safe for other threads to be adding other entries to the hash table as well as for other threads to be looking up tiles in the hash table concurrently.

The general approach is to first compute a candidate position in the hash table (stored in `hashOffset`) and then to attempt to store the given `tile` at that offset. If the entry at that offset is in use, then (as with lookups), quadratic probing is used to find the next candidate position until a free one is found. In practice, the first entry tried is usually free.

⟨*TileHashTable Method Definitions*⟩ ≡
```
void TileHashTable::Insert(TextureTile *tile) {
    int hashOffset = tile->tileId.hash() % size;
    int step = 1;
    for (;;) {
        ⟨Attempt to insert tile at hashOffset⟩
    }
}
```

The key to the lock-free implementation is the use of the atomic `compare_exchange_weak()` method to store the tile's pointer in the hash table. It takes an expected value in `cur` and a value to replace it with, `tile`. If the memory location at `table[hashOffset]` has the value stored in `cur`, then `tile` is stored there atomically and `true` is returned. Otherwise, `false` is returned and `cur` stores the current value at `table[hashOffset]`. Thus, the insertion attempt fails if another tile is already stored in that location. If two threads try to store a tile in the same `hashOffset` at the same time, only one will succeed.

⟨*Attempt to insert* `tile` *at* `hashOffset`⟩ ≡
```
TextureTile *cur = nullptr;
if (table[hashOffset].compare_exchange_weak(cur, tile,
        std::memory_order_release))
    return;
⟨Handle compare–exchange failure for hash table insertion⟩
```

There are two cases that may cause the compare–exchange to fail: as already mentioned, the current entry in the hash table may already have a different tile and thus not have the value `nullptr`. Another possibility is a *spurious failure*; `compare_exchange_weak()` is allowed to return `false` even if the first value passed to it matches the current value at that location.[3] In the first case, we update the `hashOffset` using quadratic probing and in the second, we try the exchange again at the same `hashOffset`.

3 There is a `compare_exchange_strong()` variant that will not fail spuriously, but it is much more expensive on some architectures, so the weak variant is generally preferable.

⟨*Handle compare–exchange failure for hash table insertion*⟩ ≡

```
if (cur != nullptr) {
    hashOffset += step * step;
    ++step;
    if (hashOffset >= size) hashOffset %= size;
}
```

1.5 TextureCache IMPLEMENTATION

We can now dig into the `TextureCache` implementation; it allocates a fixed number of `TextureTile`s at startup and uses the `TileHashTable` to maintain a cache of them. Tiles are read from disk on demand based on the texels needed for rendering.

⟨*TextureCache Declarations*⟩ ≡

```
class TextureCache {
public:
    ⟨TextureCache Public Methods⟩
private:
    ⟨TextureCache Private Methods⟩
    ⟨TextureCache Private Data⟩
};
```

We'll describe the part of the `TextureCache` constructor's implementation related to allocating memory for texture tiles now. Definitions of the last two fragments in the constructor will come later, closer to the code that they are related to.

⟨*TextureCache Method Definitions*⟩ ≡

```
TextureCache::TextureCache() {
    ⟨Allocate texture tiles and initialize free list⟩
    ⟨Allocate hash tables for texture tiles⟩
    ⟨Initialize markFreeCapacity⟩
    ⟨Allocate threadActiveFlags⟩
}
```

The number of texture tiles that can be resident in memory is easily computed given the total allowed texture cache size.

⟨*Allocate texture tiles and initialize free list*⟩ ≡

```
size_t maxTextureBytes = size_t(PbrtOptions.texCacheMB) * 1024 * 1024;
size_t nTiles = maxTextureBytes / TileAllocSize;
⟨Allocate tile memory for texture cache⟩
⟨Allocate TextureTiles and initialize free list⟩
```

A fixed amount of memory is allocated for all texture tiles, just enough to store 64×64 texels for the common 3-channel RGB, 8-bit texture format. Other formats use the largest power-of-two size that keeps them under the tile allocation size. (Exercise 1.1 at the end of this document discusses an alternative approach that uses memory more efficiently.)

⟨*TextureCache Private Data*⟩ ≡
```
static constexpr int TileAllocSize = 3 * 64 * 64;
```

Memory for all of the texels in all of the tiles is allocated in a single large allocation. Doing so is slightly more efficient in time (though that doesn't matter much since this is only done at startup time), and is likely to save a bit of memory, since the memory allocator doesn't need to store bookkeeping information for each tile's allocation.

⟨*Allocate tile memory for texture cache*⟩ ≡
```
tileMemAlloc.reset(new char[nTiles * TileAllocSize]);
char *tilePtr = tileMemAlloc.get();
```

⟨*TextureCache Private Data*⟩ ≡
```
std::unique_ptr<char[]> tileMemAlloc;
```

Now the `TextureTiles` are allocated and pointers offset throughout `tileMemAlloc` are used to initialize their `texels` pointers. The `TextureCache::freeTiles` member variable is used to store pointers to free tiles (initially, all of them).

⟨*Allocate* `TextureTile`*s and initialize free list*⟩ ≡
```
allTilesAlloc.reset(new TextureTile[nTiles]);
for (int i = 0; i < nTiles; ++i) {
    allTilesAlloc[i].texels = tilePtr + i * TileAllocSize;
    freeTiles.push_back(&allTilesAlloc[i]);
}
```

Other `TextureCache` methods that access `freeTiles` must hold `freeTilesMutex` when they do so. (There's no need in the constructor, however.)

⟨*TextureCache Private Data*⟩ ≡
```
std::unique_ptr<TextureTile[]> allTilesAlloc;
std::mutex freeTilesMutex;
std::vector<TextureTile *> freeTiles;
```

Not one, but two hash tables are allocated. The second one will be used when it's time to remove not-recently-used tiles from the cache and add them to the free list; we can ignore it until we discuss that part of the implementation.

⟨*Allocate hash tables for texture tiles*⟩ ≡
```
int hashSize = 8 * nTiles;
hashTable = new TileHashTable(hashSize);
freeHashTable = new TileHashTable(hashSize);
```

⟨*TextureCache Private Data*⟩ ≡
```
std::atomic<TileHashTable *> hashTable;
TileHashTable *freeHashTable;
```

The `AddTexture()` method allows callers to inform the texture cache of a texture for it to manage. It returns an integer id that identifies the texture, which should be passed back for the `texId` parameter of the `Texel()` method below to identify the texture. An error is indicated by a negative id return value.

Note that it's not safe to call this method concurrently while other threads are calling other `TextureCache` methods or for multiple threads to call it at the same time. Neither of these are problems in practice, as this method is only called during the (mostly) serial scene parsing and construction phase.

⟨*TextureCache Method Definitions*⟩ ≡

```
int TextureCache::AddTexture(const std::string &filename) {
    ⟨Return preexisting id if texture has already been added⟩
    TiledImagePyramid tex;
    if (!TiledImagePyramid::Read(filename, &tex)) return -1;
    textures.push_back(std::move(tex));
    return textures.size() - 1;
}
```

The returned id is actually an index into the `textures` vector, which holds the metadata for each texture managed by the cache.

⟨*TextureCache Private Data*⟩ ≡

```
std::vector<TiledImagePyramid> textures;
```

⟨*Return preexisting id if texture has already been added*⟩ ≡

```
auto iter = std::find_if(textures.begin(), textures.end(),
                         [&filename](const TiledImagePyramid &tex) {
                             return tex.filename == filename;
                         });
if (iter != textures.end()) return iter - textures.begin();
```

1.5.1 PARALLELISM STRATEGY

Before getting into the heart of the implementation of the texture cache—the `Texel()` and `GetTile()` methods—we'll describe the overall approach that allows threads that only need to lookup tiles to do so efficiently even in the presence of other threads adding new tiles or removing old ones. Going into this topic now will make it easier to understand some of the details of those methods.

The salient characteristics of the workload are the reading texels from tiles that are already in memory is much more frequent than loading tiles from disk and adding them to the hash table, which in turn is much more frequent than recycling not-recently-used tiles. (For example, in the performance measurements described in Section 1.6, typical ratios were that texel lookups were ∼3000× more frequent than adding tiles to the cache, which in turn was ∼8000× more frequent than recycling old tiles—thus, ∼24M texel lookups for each freeing operation.)

The "read-copy update" (RCU) technique provides an approach that can give good parallel scalability with workloads like these. Its key benefit is that it ensures that threads that are reading the data structure are never prevented from accessing it, even when it is being modified.

First, a few definitions: all reads of RCU-managed data are done with the reading thread inside a *read-side critical section*.[4] In turn, when not in a read-side critical section, a thread is in a *quiescent state* and must not access any RCU-managed data. A period of time where each thread has been in a quiescent state at least once is a *grace period*.

Given these definitions, there are two main ideas that underlie RCU.

- Modifications to a RCU-managed data structure must be done in a way such that the data structure is always in a consistent state and such that threads traversing it while it's being modified always see a valid data structure. (The compare-and-swap and memory ordering semantics used in `TileHashTable::Insert()` take care of this for inserting tiles; we will see how this is done for tile removal in a few pages.)
- If items are removed from the data structure, then the memory for the items can only be freed or reused after a grace period has passed; only then can we be sure that no thread is still accessing those entries.

There are many ways to detect the passage of a grace period. Our implementation is based on per-thread flags that indicate whether each thread is currently inside a read-side critical section. Each thread sets its flag to `true` before accessing RCU-managed data—the hash table, `TextureTiles`, and texels in those tiles—and only resets it to `false` when it is done.

A flag is allocated for each thread in the `TextureCache` constructor.

⟨*Allocate* `threadActiveFlags`⟩ ≡
```
threadActiveFlags = std::vector<ActiveFlag>(MaxThreadIndex());
```

⟨*TextureCache Private Data*⟩ ≡
```
std::vector<ActiveFlag> threadActiveFlags;
```

An `atomic bool` is all we need for each flag, but it's critical that each thread's flag be stored in a separate cache line; this is taken care of by the following `alignas` specifier. Without it, rendering was 5% slower on our test system due to false sharing and the resulting unnecessary cache coherence protocol overhead.

⟨*ActiveFlag Declarations*⟩ ≡
```
struct alignas(PBRT_L1_CACHE_LINE_SIZE) ActiveFlag {
    std::atomic<bool> flag{false};
};
```

When a thread enters a read-side critical section, it calls `RCUBegin()`, which sets its active flag to `true`. This is in general a highly efficient operation—normally, the thread's flag will be in its own CPU's cache and no other thread will have read from it, so no expensive CPU cache coherence operations are expected.

4 Being inside a read-side critical section doesn't require inter-thread communication or expensive synchronization mechanisms; for the moment, just think of it in terms of somehow delineated period of time for each thread when it's accessing shared data.

⟨*TextureCache Private Methods*⟩ ≡

```
void RCUBegin() {
    std::atomic<bool> &flag = threadActiveFlags[ThreadIndex].flag;
    flag.store(true, std::memory_order_acquire);
}
```

Upon exiting a read-side critical section, a thread should call `RCUEnd()`, which clears its flag. Note the memory ordering specifiers associated with these two operations, which ensure that no accesses of RCU-managed data are moved outside of the read-side critical section.

⟨*TextureCache Private Methods*⟩ ≡

```
void RCUEnd() {
    std::atomic<bool> &flag = threadActiveFlags[ThreadIndex].flag;
    flag.store(false, std::memory_order_release);
}
```

The `WaitForQuiescent()` method allows one thread to wait for another to enter a quiescent state. If the other thread isn't in a read-side critical section when the first one checks, the first will immediately see that the flag is `false`; otherwise it spins until the other thread finishes its read-side critical section. In any case, the other thread isn't inhibited from executing. After a thread calls this method for all of the other threads, it has ensured that a grace period has passed (as long as it isn't itself in a read-side critical section).

⟨*TextureCache Private Methods*⟩ ≡

```
void WaitForQuiescent(int thread) {
    std::atomic<bool> &flag = threadActiveFlags[thread].flag;
    while (flag.load(std::memory_order_acquire) == true)
        ; // spin
}
```

1.5.2 TEXEL LOOKUPS

With the RCU basics established, we can proceed to texel lookups, where the ideas are put into use. The `Texel()` method returns a single texel value from the given texture level. All texture cache management happens as a result of calls to this method. Note that `Texel()` is a template method; callers can use it to get both `Float` and `Spectrum` texel values, independent of the texture's underlying format.

⟨*TextureCache Method Definitions*⟩ ≡

```
template <typename T>
T TextureCache::Texel(int texId, int level, Point2i p) {
    const TiledImagePyramid &tex = textures[texId];
    ⟨Return texel from preloaded levels, if applicable⟩
    ⟨Get texel pointer from cache and return value⟩
}
```

`Texel()` starts by checking to see if the requested texel is one of the ones already in memory because it's in one of the very low resolution levels at the top of the pyramid. If so, we can avoid anything related to the cached tiles entirely.

⟨*Return texel from preloaded levels, if applicable*⟩ ≡

```
const char *texel = tex.GetTexel(level, p);
if (texel != nullptr) return ConvertTexel<T>(texel, tex.pixelFormat);
```

Otherwise, a call to `GetTile()` gives a pointer to the start of the texture tile that holds the texel for the coordinate `p`, loading the tile from disk if necessary. `TiledImagePyramid` methods give us the tile coordinates for `p` as well as its offset within the tile.

Because the pointer to the texels is RCU-managed data, `GetTile()` marks the start of a read-side critical section by calling `RCUBegin()` before accessing the hash table, leaving it active when returning the pointer. In turn, here we must not mark the critical section's end until after `ConvertTexel()` has converted the texel data to the value that `Texel()` will return.

⟨*Get texel pointer from cache and return value*⟩ ≡

```
TileId tileId(texId, level, tex.TileIndex(p));
texel = GetTile(tileId) + tex.TexelOffset(p);
T ret = ConvertTexel<T>(texel, tex.pixelFormat);
RCUEnd();
return ret;
```

`GetTile()` returns a pointer to the start of the texels for the given texture tile.

⟨*TextureCache Method Definitions*⟩ ≡

```
const char *TextureCache::GetTile(TileId tileId) {
    ⟨Return tile if it's present in the hash table⟩
    ⟨Check to see if another thread is already loading this tile⟩
    ⟨Load texture tile from disk⟩
    ⟨Add tile to hash table and return texel pointer⟩
}
```

The read-side critical section must start before the hash table is accessed at all, including loading the `hashTable` pointer. If the call to `Lookup()` is successful, the method returns without ending the read-side critical section. Otherwise, the critical section ends—this thread won't be accessing the hash table again for a while (and indeed, may end up performing disk I/O, which may take quite some time), and it's important that the reported time in read-side critical sections be as short as possible so that any other threads that may be waiting for a quiescent state don't wait any longer than necessary.

⟨*Return tile if it's present in the hash table*⟩ ≡

```
RCUBegin();
TileHashTable *t = hashTable.load(std::memory_order_acquire);
if (const char *texels = t->Lookup(tileId))
    return texels;
RCUEnd();
```

1.5.3 READING TILES FROM DISK

If the tile isn't in the cache, then it must be read from disk. However, if another thread has already started an I/O operation to load it, it does no good for the current thread to issue a redundant I/O operation. (Indeed, doing so is likely to delay I/O for other tiles, reducing overall performance.) Therefore, the texture cache tracks which tiles are currently being read from disk so that other threads that want the same tile can just wait for the thread already loading it to add it to the hash table.

This bookkeeping is handled by a vector that stores the `TileId`s for tiles currently being read; access to the vector is controlled with a mutex. (There's no need to worry about a more scalable approach since tile cache misses are so much less frequent than hits. Further, this mutex is never held by any thread for very long.)

⟨*Check to see if another thread is already loading this tile*⟩ ≡

```
outstandingReadsMutex.lock();
for (const TileId &readTileId : outstandingReads) {
    if (readTileId == tileId) {
        ⟨Wait for tileId to be read before retrying lookup⟩
    }
}
⟨Record that the current thread will read tileId⟩
```

⟨*TextureCache Private Data*⟩ ≡

```
std::mutex outstandingReadsMutex;
std::vector<TileId> outstandingReads;
```

If the tile is already being read from disk, the current thread waits on a condition variable, `outstandingReadsCondition`. Doing so requires slightly messy code, since a `std::unique_lock` must be passed to the condition variable's `wait()` method and we locked the mutex directly above. After the `wait()` call, the thread is suspended until another thread signals the condition variable.

After the thread is awakened, a recursive call to `GetTile()` tries to find the tile in the hash table. This lookup may not be successful, for example if tiles were freed from the hash table after the read finished but before this thread got a chance to call `Lookup()` again. (This case is unlikely, however, since the tile was presumably added very recently by another thread and thus shouldn't be chosen to be recycled when there are no more available tiles.) In any case, the regular lookup mechanisms just try again until the tile is found.

⟨*Wait for* `tileId` *to be read before retrying lookup*⟩ ≡

```
std::unique_lock<std::mutex> readsLock(outstandingReadsMutex,
    std::adopt_lock);
outstandingReadsCondition.wait(readsLock);
readsLock.unlock();
return GetTile(tileId);
```

⟨*TextureCache Private Data*⟩ ≡

```
std::condition_variable outstandingReadsCondition;
```

If the current thread is the one to read the tile, it records this fact and releases the mutex protecting `outstandingReads` so that other threads can access it while it is reading texels from disk.

⟨*Record that the current thread will read* `tileId`⟩ ≡

```
outstandingReads.push_back(tileId);
outstandingReadsMutex.unlock();
```

We can now load the tile's texels from disk. The `GetFreeTile()` method returns an available tile, freeing tiles if needed, and `ReadTile()` reads the specified tile from disk.

⟨*Load texture tile from disk*⟩ ≡

```
TextureTile *tile = GetFreeTile();
ReadTile(tileId, tile);
```

If there are no free tiles, `FreeTiles()` frees some of the entries in the hash table. Its implementation is the subject of the next section, 1.5.5. (The ⟨*Mark hash table entries if free-tile availability is low*⟩ fragment is discussed there as well.) For now we can proceed knowing that one way or another, entries will be available in `freeTiles` for the execution of the code fragment that concludes `GetFreeTile()`.

⟨*TextureCache Method Definitions*⟩ ≡

```
TextureTile *TextureCache::GetFreeTile() {
    std::lock_guard<std::mutex> lock(freeTilesMutex);
    if (freeTiles.size() == 0)
        FreeTiles();
    ⟨Mark hash table entries if free-tile availability is low⟩
    ⟨Return tile from freeTiles⟩
}
```

Given at least one entry in `freeTiles`, our task is straightforward; a tile is removed from the free list and reset using the `TextureTile::Clear()` method, not included here; it resets the tile's `tileId` field to the "invalid" value and clears some bookkeeping data used for freeing tiles.

⟨*Return tile from* `freeTiles`⟩ ≡

```
TextureTile *tile = freeTiles.back();
freeTiles.pop_back();
tile->Clear();
return tile;
```

`ReadTile()` reads the texels for the tile identified by `tileId` into the `texels` buffer of the provided `TextureTile`.

⟨*TextureCache Method Definitions*⟩ ≡
```
void TextureCache::ReadTile(TileId tileId, TextureTile *tile) {
    tile->tileId = tileId;
    const TiledImagePyramid &tex = textures[tileId.texId()];
    ⟨Get file descriptor and seek to start of texture tile⟩
    ⟨Read texel data and return file descriptor⟩
}
```

The `TiledImagePyramid` lets us know where in the file this tile's texels start; we seek to that offset using a file descriptor for the texture's file provided by the `FdCache`. `FdCache` maintains a cache of open file descriptors; it will be described momentarily.

⟨*Get file descriptor and seek to start of texture tile*⟩ ≡
```
int64_t offset = tex.FileOffset(tileId.level(), tileId.p());
FdEntry *fdEntry = fdCache.Lookup(tileId.texId(), tex.filename);
if (lseek(fdEntry->fd, offset, SEEK_SET) == -1)
    Error("%s: seek error %s", tex.filename.c_str(), strerror(errno));
```

⟨*TextureCache Private Data*⟩ ≡
```
FdCache fdCache;
```

After the seek, a call to the `read()` sysem call reads the texels from disk.

⟨*Read texel data and return file descriptor*⟩ ≡
```
int tileBytes = tex.TileBytes();
if (read(fdEntry->fd, tile->texels, tileBytes) != tileBytes)
    Error("%s: read error %s", tex.filename.c_str(), strerror(errno));
fdCache.Return(fdEntry);
```

File Descriptor Management

Opening and closing files can be somewhat expensive; given that we will generally want to read multiple tiles from each texture, it's worthwhile to leave files open between reads. However, most operating systems impose a limit on the number of open files; this limit may be insufficient for the number of textures in a scene (especially since multiple threads may want to load tiles from the same texture concurrently), so it's important to be able to limit the number of open files for texture lookups. The `FdCache` class takes care of keeping a fixed-size cache of open file descriptors, providing them when needed for texture lookups, and closing not-recently-used ones once too many are open.

Because tile reads are relatively rare (thousands of times less frequent than texel lookups), the `FdCache` implementation can be reasonably straightforward without a lot of attention to parallel scalability; a single mutex protects updates to the hash table it uses to store file descriptors.

The `Lookup()` method returns a `FdEntry *` for the given file. The file is specified by a unique identifier (`fileId`, which here corresponds to the texture id returned by `TextureCache::AddTexture()`) and its filename. Hash table lookups are done using just the identifier; the filename is only used when the file must be opened.

After its use for a read operation, the file descriptor should be returned to the cache via the `Return()` method.

⟨*FdCache Public Methods*⟩ ≡

```
FdEntry *Lookup(int id, const std::string &filename);
void Return(FdEntry *entry);
```

The `FdCache` uses the `FdEntry` structure to represent file descriptors. Callers can access the file descriptor (as would be returned by the `open()` system call) in the `fd` field.

⟨*FdEntry Declarations*⟩ ≡

```
class FdEntry {
public:
    int fd = -1;
private:
    ⟨FdEntry Private Data⟩
};
```

1.5.4 ADDING A TILE TO THE TEXTURE CACHE

Now the tile can be added to the cache. Before accessing the hash table to add the tile after it is in memory, another read-side critical section must be started. Note that a fresh load from `hashTable` is performed to get the hash table pointer—we must not use the pointer loaded earlier in `GetTile()`, since the read-side critical section where that pointer was loaded has ended. (The importance of this detail will become clear after we present the tile freeing algorithm.)

⟨*Add tile to hash table and return texel pointer*⟩ ≡

```
RCUBegin();
t = hashTable.load(std::memory_order_relaxed);
t->Insert(tile);
⟨Update outstandingReads for read tile⟩
return tile->texels;
```

Once the tile has been added to the hash table, we can remove the corresponding entry from `outstandingReads` and wake up any threads waiting on the condition variable. Because there is only a single condition variable rather than one for each outstanding tile read, any threads waiting for a different tile will be woken nonetheless. This is somewhat unfortunate, though such threads will just attempt to find the tile in the hash table again, not find it, check `outstandingReads`, find that the read they had been waiting on is still outstanding, and wait on the condition variable again.

⟨*Update* `outstandingReads` *for read tile*⟩ ≡
```
outstandingReadsMutex.lock();
for (auto iter = outstandingReads.begin(); iter != outstandingReads.end();
     ++iter) {
     if (*iter == tileId) {
         outstandingReads.erase(iter);
         break;
     }
}
outstandingReadsMutex.unlock();
outstandingReadsCondition.notify_all();
```

The very astute reader may have noticed that with the implementation we have described it's actually possible that two threads may add the same tile to the hash table: if one thread adds a tile to the hash table immediately after another finishes a failed lookup for the tile but before trying to acquire `outstandingReadsMutex`, then the second thread will not find the tile in `outstandingReads` and will proceed to add it itself. While this could be fixed by checking the hash table again with the mutex held, when we instrumented the implementation to check for this case, we never found an instance where it happened. Should it, the cost is only a tile's worth of unnecessary I/O and a single `TextureTile`, so we left the implementation as it stands.

1.5.5 FREEING TILES

In the implementation so far, we have enjoyed the benefits of RCU: both hash table lookups and insertion could proceed uninhibited, without any awareness of the fact that tiles are periodically recycled. Now comes the tricky part: we need to implement an algorithm that frees not-recently-used tiles once the tile free list is empty, and we need to do so in a way such that other threads always encounter a valid hash table during this process.

The approach we'll follow is to first construct a new private hash table that only stores the tiles we'd like to keep in the cache. Other threads can continue to use the old hash table as normal while the new hash table is being initialized. Once the new hash table is ready, it's atomically swapped with the old one. After this point, any thread that starts accessing the tile hash table will see the new one and will be unable to access tiles in the old one, including those that are to be freed.

However, at that point we can't yet start adding freed tiles to the free list: even after the swap, other threads may still be accessing the old hash table or texels in tiles it stores. Therefore, the next step is to wait for an RCU grace period to pass. Only then can we be sure that entries in the old hash table that weren't copied aren't being accessed by other threads and thus can be added to the free list.

Choosing Tiles to Free

In order to determine which tiles to free, our implementation marks tiles as the hash table gets full, clears those marks when tiles are accessed, and then later frees the tiles that are still marked. This approach works reasonably well to find tiles that haven't been accessed recently, with low runtime overhead. The ⟨*Mark hash table entries if free-tile availability is low*⟩ fragment from the `TextureCache::GetFreeTile()` method takes care of marking tiles; once the number of free tiles reaches a low watermark, all current hash table entries are marked.[5]

⟨*Mark hash table entries if free-tile availability is low*⟩ ≡
```
if (freeTiles.size() == markFreeCapacity)
    hashTable.load(std::memory_order_acquire)->MarkEntries();
```

The `TextureCache` constructor initializes a member variable, `markFreeCapacity`, that gives the point at which active tiles should be marked.

⟨*Initialize* `markFreeCapacity`⟩ ≡
```
markFreeCapacity = 1 + nTiles / 8;
```

⟨*TextureCache Private Data*⟩ ≡
```
int markFreeCapacity;
```

The `MarkEntries()` method goes through the hash table and sets each tile's `marked` field to `true`. Note that this is safe to do while other threads are concurrently looking up or adding entries to the hash table; marking an entry just involves an atomic store to a `TextureTile` member variable. If a new entry is added earlier in the table than the current loop index `i`, it will just be left unmarked. This is fine, since as a recent addition, we probably don't want to free it soon anyway. (Similarly for tiles added to the hash table after `MarkEntries()` returns.)

The store to `marked` doesn't carry along any restrictions on the ordering of other memory operations, so can be performed with relaxed ordering.

⟨*TileHashTable Method Definitions*⟩ ≡
```
void TileHashTable::MarkEntries() {
    for (size_t i = 0; i < size; ++i) {
        TextureTile *entry = table[i].load(std::memory_order_acquire);
        if (entry)
            entry->marked.store(true, std::memory_order_relaxed);
    }
}
```

⟨*TextureTile Public Data*⟩ ≡
```
mutable std::atomic<bool> marked;
```

5 Here we haven't bothered to bracket the access to the hash table with `TextureCache::RCUBegin()` and `TextureCache::RCUEnd()` calls; doing so is actually unnecessary in this case. In in our implementation the per-thread flags are only ever accessed to check for quiescence in `FreeTiles()`, but here the `freeTilesMutex` has been acquired at the start of the `GetFreeTile()` method, which in turn prevents any other thread from entering `FreeTiles()`.

We can now also implement the fragment in `TileHashTable::Lookup()` that clears `marked` after a tile is found in the hash table. A small detail is that `marked` is only written to if its current value is `true`; if it's already unmarked, then no redundant write is performed and a bit of unnecessary inter-core communication is saved.

As with setting `marked`, clearing it imposes no further memory ordering requirements. Note that it is possible that two threads may simultaneously clear `marked`; this is harmless, and the added overhead of a more expensive mechanism to avoid it (e.g. an atomic compare-exchange), would give no benefit.

⟨*Update* `entry`*'s* `marked` *field after cache hit*⟩ ≡

```
if (entry->marked.load(std::memory_order_relaxed))
    entry->marked.store(false, std::memory_order_relaxed);
```

Constructing the New Hash Table

We can now start to implement the `FreeTiles()` method. As you read through its implementation, recall that the calling code in `GetFreeTile()` is holding `freeTilesMutex`, which ensures that only one thread can be in `FreeTiles()` at once.

As described earlier, the idea is to construct a new hash table that stores only some of the tiles in the current hash table and to add the remaining tiles to the free list once it is certain that no other thread is accessing them.

⟨*TextureCache Method Definitions*⟩ ≡

```
void TextureCache::FreeTiles() {
    ⟨Copy unmarked tiles to freeHashTable⟩
    ⟨Swap texture cache hash tables⟩
    ⟨Ensure that no threads are accessing the old hash table⟩
    ⟨Add inactive tiles in freeHashTable to free list⟩
}
```

Copying active tiles to the new hash table are handled by `TileHashTable::CopyActive()`.

⟨*Copy unmarked tiles to* `freeHashTable`⟩ ≡

```
hashTable.load(std::memory_order_relaxed)->CopyActive(freeHashTable);
```

Usually, `CopyActive()` copies a subset of the tiles to the new hash table, leaving a number of others to be freed. However, if the texture cache is much too small to store the working set of the rendering threads, it may happen that all of the entries are unmarked and thus copied; in this case, the ⟨*Handle case of all entries copied to* `freeHashTable`⟩ fragment falls back to copying an arbitrary subset of the tiles to the destination hash table, at least ensuring that some free tiles will be available. (Since that fragment is both rarely invoked and not very interesting, we won't include its implementation here.)

⟨*TileHashTable Method Definitions*⟩ ≡

```
void TileHashTable::CopyActive(TileHashTable *dest) {
    int nCopied = 0, nActive = 0;
    ⟨Insert unmarked entries from hash table to dest⟩
    ⟨Handle case of all entries copied to freeHashTable⟩
}
```

Note that no mutual exclusion is used when copying from the active hash table: more entries may be added to it by other threads while `CopyActive()` is copying entries to `dest`. This poses no problem: if an entry is added after the position of the index `i`, then it will be copied over to `dest`. If added before, it will be recycled and added to the free list. While the latter case is a bit undesirable (since a recently-added tile is likely to see more use in the near future), we have found that this case happens incredibly rarely—usually never in our experiments. Because the cost when it does happen is a small amount of redundant I/O, we won't bother to do anything extra to avoid it here.

⟨*Insert unmarked entries from hash table to* `dest`⟩ ≡

```
for (size_t i = 0; i < size; ++i) {
    hashEntryCopied[i] = false;
    if (TextureTile *entry = table[i].load(std::memory_order_acquire)) {
        ⟨Add entry to dest if unmarked⟩
    }
}
```

`hashEntryCopied` is allocated in the `TileHashTable` constructor; it's the same size as the hash table and is used to record which entries were added to the new hash table.

⟨*TileHashTable Private Data*⟩ ≡

```
std::vector<bool> hashEntryCopied;
```

⟨*Add* `entry` *to* `dest` *if unmarked*⟩ ≡

```
++nActive;
if (entry->marked.load(std::memory_order_relaxed) == false) {
    hashEntryCopied[i] = true;
    ++nCopied;
    dest->Insert(entry);
}
```

After the unmarked tiles have been added to `freeHashTable`, an atomic swap via the `std::atomic exchange()` method replaces the pointer to the hash table stored in `TextureCache::hashTable` with `freeHashTable`, thus making the new hash table the one that is found by other threads. The exchange is performed with both acquire and release memory semantics, thus ensuring that memory operations before the exchange (e.g. initializing the new hash table) are all ordered before the exchange, and operations after (e.g. waiting for the RCU grace period) are all ordered after.

⟨*Swap texture cache hash tables*⟩ ≡

```
freeHashTable = hashTable.exchange(freeHashTable, std::memory_order_acq_rel);
```

Adding Tiles to the Free List

After `FreeTiles()` has exchanged the new hash table for the old one, but before the uncopied tiles can be added to the free list, it's necessary to be sure that no thread is still accessing the old hash table (or tiles that were stored in it). Waiting for an RCU grace period to pass suffices to ensure that this is the case.

⟨*Ensure that no threads are accessing the old hash table*⟩ ≡

```
for (size_t i = 0; i < threadActiveFlags.size(); ++i)
    WaitForQuiescent(i);
```

It's now safe to add the free tiles to `freeTiles`; this is handled by the `TileHashTable::ReclaimUncopied()` method. (Recall that `freeTilesMutex` is held when `FreeTiles()` executes, so it's safe to modify `freeTiles` without worrying about other threads trying to access it concurrently.) If, after freeing, the number of free tiles is under the low watermark, we immediately mark the tiles in the hash table so that the next tile free operation has a chance of making sensible choices about which tiles to reclaim.

⟨*Add inactive tiles in* `freeHashTable` *to free list*⟩ ≡

```
freeHashTable->ReclaimUncopied(&freeTiles);
if (freeTiles.size() < markFreeCapacity)
    hashTable.load(std::memory_order_acquire)->MarkEntries();
```

`ReclaimUncopied()` clears the hash table and adds any entries that weren't copied to the provided vector. All memory operations can be performed with relaxed semantics, since this method is only used with `freeHashTable`, which will only be visible to other threads in the future after a separate memory ordering operation.

⟨*TileHashTable Method Definitions*⟩ ≡

```
void TileHashTable::ReclaimUncopied(std::vector<TextureTile *> *returned) {
    for (size_t i = 0; i < size; ++i) {
        if (TextureTile *entry = table[i].load(std::memory_order_relaxed)) {
            if (!hashEntryCopied[i])
                returned->push_back(entry);
            table[i].store(nullptr, std::memory_order_relaxed);
        }
    }
}
```

1.6 EVALUATION

We have performed a series of experiments to evaluate our implementation. In the remainder of this section, we first investigate how well texture caching works in general for texture-heavy scenes rendered with path tracing and then investigate the effect of cache size on overall performance. Next, we examine the effect of ray differentials and disk read latency on these results. Finally, we evaluate the

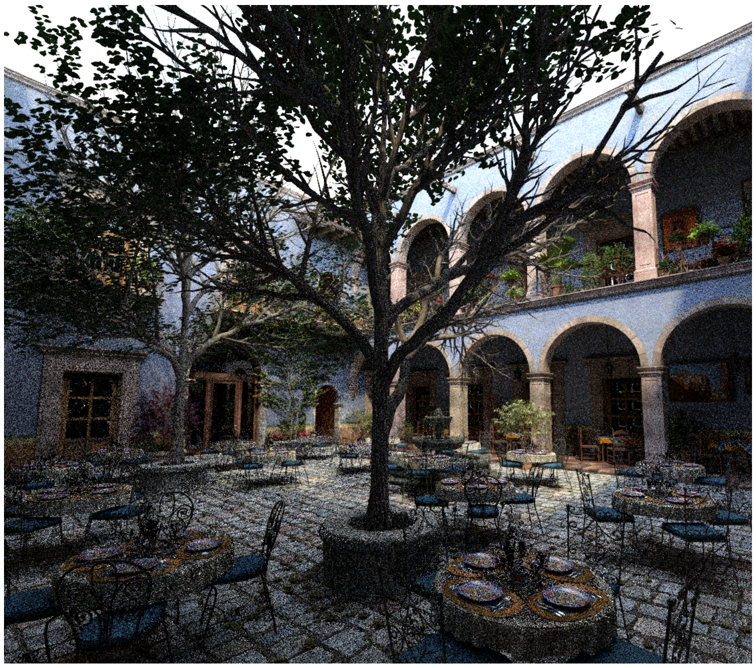

Figure 1.1: San Miguel Scene Used for Texture Cache Evaluation. This is `sanmiguel_cam14.pbrt` from the `pbrt` scenes distribution.

parallel scalability of our implementation and compare to a few alternate texture cache designs.

1.6.1 METHODOLOGY

Most of our measurements used the view of the San Miguel scene shown in Figure 1.1. All were rendered at 1600×1100 resolution using path tracing, with 32 samples per pixel. Bilinear filtering was used for all texture lookups (using the more detailed MIP map pyramid level closest to the level indicated by the filter footprint.) All of our timings are reported in terms of wall-clock time and are the minimum reported time over a few runs.

We ran our tests on a Google Compute Engine n1-highcpu-32 instance with a Haswell CPU, running Ubuntu 15.10 Linux. This system has 16 physical cores with hyperthreading and 28.8 GB of RAM.

When benchmarking a system like this one that uses caching to reduce I/O, it's important to be mindful of the operating system's buffer cache: operating systems generally use free memory to store the contents of files that have been read recently; if they are read again while still in the buffer cache, their contents can just be copied from RAM rather than read from disk (generally thousands of times more quickly).

Therefore, for evaluation purposes here, we have modified our implementation to optionally impose a minimum latency on disk reads, thus better modeling the actual overhead of reading from disk. (When not benchmarking, of course, one is happy to have the buffer cache help out when it is able to.) Unless specified otherwise, we have imposed a minimum 5ms overhead on all texture tile reads in the following; this is much more overhead than there would be from a local SSD, but is probably less than there would be if reading textures from a file server over the network. Section 1.6.4 examines the effect of varying the read latency on performance.

1.6.2 BENEFITS OF ON-DEMAND LOADING

It's instructive to measure the value of loading textures on demand, using a large enough cache such that texture tiles never need to be freed. For two scenes, we compared performance with loading all textures at startup time (as `pbrt-v3` does today) versus only loading tiles of them when needed for a texture access. For the preloading case, our implementation differs from `pbrt-v3` in that it stores texels from 8-bit texture formats in a single byte, rather than using a 4-byte `float`. (This matches how the texture cache stores tiles in memory.)

Before running these tests, we primed the OS buffer cache by reading the contents of all of the textures (`cat textures/* > /dev/null`) before running the benchmarks; if anything, this gives a slight advantage to the preloading case, since it loads more data from disk. (The 28 GB of RAM in the system was enough to comfortably store all of the textures in the buffer cache.)

Total run time for the San Miguel scene with 32 threads and preloaded textures was 157s; 684 MB of RAM was used for texture data. When rendered with textures loaded on demand using our texture cache, it took just 105s and only 141.8 MB of texture data was read from disk. Almost all of this time difference is thanks to saving 55s at startup from from not loading the textures and computing MIP maps then. If we subtract scene parsing and construction time and only measure time spent rendering the image, run time increased from 89.5s to 92.5s when lazy loading was used, though recall that in the lazy loading case, time spent reading textures from disk adds to rendering time, while it doesn't with preloading.

We also evaluated on-demand loading with a frame from the *Measure One* animation shown in Figure 1.2. (This corresponds to `frame52.pbrt` in the `pbrt` scenes distribution.) This scene was rendered at 1440×630 resolution with path tracing and 512 samples per pixel.

With preloaded textures, the scene renders in 655s and 470.2 MB is used to store textures in memory. At startup time, 17s is spent reading textures and generating MIP maps. With lazy loading, total run time is 612s (7% faster), and only 47.93 MB is needed to store all textures that are ever accessed.

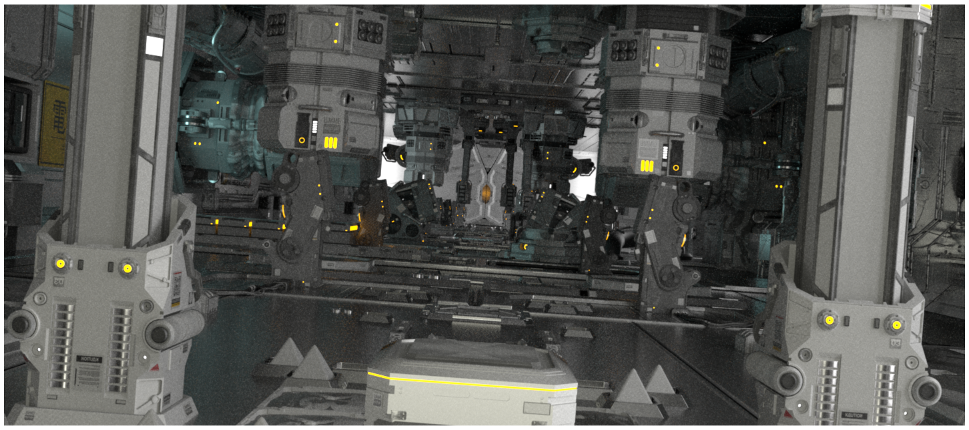

Figure 1.2: Frame 52 of the Measure One Animation. If all texture in the scene is loaded at startup time, 470 MB of texture is stored in memory. With lazy loading, only 48 MB of texture is read from disk.

Other than memory savings, the performance benefits from lazy texture loading largely depend on how long it takes to render the scene—for a multi-hour render, saving tens of seconds at startup time to load textures is no big deal. However, for quick previews and low sampling rates, this savings helps a lot.

1.6.3 EFFECT OF CACHE SIZE

Having established that on-demand texture loading works well, we'll now examine how texture cache size affects performance. We rendered the San Miguel scene with a range of texture cache sizes and measured three statistics: rendering time, the cache miss rate (the fraction of texel lookups that didn't find the corresponding tile in the cache), and the total number of bytes read from disk. As before, we used 32 threads and a minimum 5ms tile read latency for these experiments. Table 1.1 shows the results.

Down to an 80 MB cache, run time is just as good as the unlimited cache case; disk I/O has gone up by 31%, but averaged over the total rendering time, it's just over 1.8 MB/s, which is quite reasonable. At a 48 MB cache and smaller, the miss rate starts to increase rapidly and total I/O picks up accordingly, as the cache is too small to hold the texture working set; many tiles are discarded to make room for other ones, just to be read back into memory again soon afterward. Happily, 80 MB is a very small amount of memory in the grand scheme of things for rendering a complex scene with path tracing; it's just under 12% of the total amount of texture data for this scene.

One useful measure of the performance of a cache is the rate of *capacity misses*—the fraction of misses that are for items that were previously in the cache but were then evicted. This value can be computed from the amount of disk I/O: all bytes

Table 1.1: Performance as a Function of Texture Cache Size in MB. 160 MB is enough to store all accessed textures in memory; run time is unchanged down to an 80 MB cache.

Cache Size	Run time	Miss Rate	Total I/O
160 MB	102s	0.0018%	141.81 MB
128 MB	102s	0.0019%	146.52 MB
96 MB	102s	0.0020%	157.28 MB
80 MB	102s	0.0024%	185.50 MB
64 MB	105s	0.0050%	401.03 MB
48 MB	117s	0.0231%	1.75 GB
40 MB	140s	0.0519%	3.69 GB
32 MB	217s	0.0880%	9.48 GB

read beyond those needed for the total amount of texture accessed (141.81 MB) must be for items that were previously in the cache. Thus, for the 80 MB cache, $185.50 - 141.81 = 43.69$ MB of disk I/O was due to capacity misses, or 24% of the total. For a 48 MB cache, the capacity miss rate is much higher: 92%.

1.6.4 READ LATENCY

As discussed in Section 1.6.1, we have imposed a minimum disk read latency on all texture tile reads in order to counteract the operating system buffer cache. In order to understand how the texture cache would perform under a variety of conditions—from fast local SSDs to slower remote fileservers—we measured how performance changed as a function of disk read latency. Table 1.2 shows the results with the San Miguel scene, a 64 MB texture cache, and various amounts of tile read latency.[6]

We can see that disk latency indeed affects overall performance, though the latencies we would expect from a local SSD (< 1ms) or a local spinning disk ($\sim$10ms) don't slow down rendering too much. For higher latencies, as might be seen when reading from a file server, performance degrades significantly.

There is some hope, however—carefully examining the measurements provides an interesting clue. For this scene with a 64 MB texture cache, a total of 34,078 texture tiles were read from disk. With a 50ms read latency, this should in theory add 1704s of waiting time compared to no read latency; split over 32 threads, this is 53s of wall-clock time. Yet, we've only seen a 40s loss in performance. How can this be?

Recall that we're running on a system with hyperthreading: each CPU core has hardware contexts for two threads and normally switches between them frequently; doing so allows computation to continue when one of the threads

6 The actual average read latency was just 0.009ms in the case that we didn't impose a minimum latency, indicating that the buffer cache was indeed providing file data from RAM.

Table 1.2: Performance Versus Read Latency when Rendering the San Miguel Scene. These measurements were performed with 32 threads and a 64 MB texture cache. As the time to read a texture tile from disk increases, rendering time increases accordingly.

Read Latency	Run time
0ms	102s
1ms	104s
5ms	106s
10ms	108s
25ms	119s
50ms	142s

is stalled. When a hyperthreaded CPU has just one thread to run, that thread will run more quickly than if it was vying for time with another thread (though it generally won't get as much done as two threads on a single core would.) Thus, when a thread is stalled waiting for disk I/O, we don't fully pay the price for it being idle; the other thread running on its core can continue (if not stalled itself), and more quickly than it would have otherwise. This explains why overall performance didn't fall as much as we might have expected.

In the case of high read latency, adding more threads can improve performance. By giving the CPU more threads to run, there's a better chance that some of them won't be waiting for I/O.[7] With a 50ms read latency and 40 threads, the scene renders in 125s. With 48 threads, it takes 117s, and with 64 threads, it finishes in 108s—the same as the 10ms latency case with 32 threads. Adding more threads has its limits; having more rendering threads than hardware threads will cause more context switches and is likely to affect overall performance by increasing the CPU cache miss rate. However, in high-latency I/O situations, this may be a reasonable trade off.

1.6.5 THE IMPORTANCE OF RAY DIFFERENTIALS

As discussed in Section 1.2.3, ray differentials play an important role in texture cache performance: they improve the spatial locality of texture map lookups for both directly- and indirectly-visible surfaces, making it more likely that texture lookups for adjacent image will access adjacent texels in the texture map, even for texture lookups after the first intersection. In order to measure the importance of ray differentials, we rendered the San Miguel scene with indirect ray differentials disabled. (Specifically, ray differentials were still present for camera rays, but all indirect rays did not have ray differentials and thus always sampled the most detailed MIP level—equivalent to the current behavior of the path tracer in `pbrt-v3`.)

7 As it turns out, GPUs use this same mechanism to hide the effect of main memory latency from programs that they run.

Table 1.3: Performance When Indirect Rays Don't Have Differentials. Without ray differentials, indirect rays always access the most detailed level of the MIP map, the working set is much larger, and much larger cache sizes are needed than if differentials are maintained.

Cache Size	Run time	Disk I/O
448 MB	114s	410.36 MB
384 MB	115s	530.27 MB
320 MB	127s	1.66 GB
256 MB	184s	7.12 GB
192 MB	473s	27.67 GB

The difference was remarkable; without indirect ray differentials, the total amount of texture data accessed was 410.36 MB—2.9× more than with ray differentials! Table 1.3 shows performance in terms of runtime and bytes of texture data read as a function of texture cache size. Performance starts to degrade rapidly with a texture cache of 320 MB, which takes 127s to render and reads 1.66 GB of texture from disk. For comparison, recall from Section 1.6.3 that with ray differentials and a 64 MB texture cache, the scene rendered in 105s and read only 401 MB of texture from disk.

It's also interesting to compare results for the "all in cache" cases here and in Section 1.6.3. Performance without differentials is almost 12% worse than the case with differentials (114s versus 102s). The difference is largely reflected in difference in the time spent in `MIPMap::Lookup()` (from 11.5s to 5.7s), which is apparently worse without ray differentials due to less coherent memory accesses and worse CPU cache behavior.

1.6.6 SCALING WITH MORE THREADS

To evaluate how well the performance of our texture cache scales with increasing numbers of threads, we compared the parallel scalability of `pbrt` with the texture cache to its scalability with preloaded textures. We again rendered the San Miguel scene, comparing preloaded textures, a 160 MB texture cache (large enough to hold all accessed tiles), and a 40 MB texture cache (where there was a significant amount of tile freeing over the course of rendering).

For these tests, we subtracted system startup time (including parsing scene description files, BVH construction, etc.) and shutdown time, since these phases are largely serial. Thus, we can focus on the parallel scalability of the actual image rendering computation. (As before, note that this means that time spent on texture I/O is included in the texture caching measurements, but not in the preloaded textures measurements.) The results are shown in Table 1.4.

The results are quite good: with or without the texture cache, `pbrt` exhibits linear parallel scaling up to the number of physical cores in the system, 16, and consistently good scaling up to the number of hardware threads, 32. All of the work to use lock-free algorithms and RCU has paid off.

Table 1.4: Comparison of Parallel Scalability of `pbrt` with Preloaded Textures to `pbrt` with Texture Caching. We measured image rendering time (independently of scene construction and cleanup time) with a range of thread counts for both preloaded textures and two differently-sized texture caches. Scalability was just as good with a texture cache as without.

Threads	Preload (no cache)	160 MB Cache	40 MB Cache
1	2172s (1x)	2177s (1x)	3672s (1x)
4	522s (4.16x)	542s (4.02x)	903s (4.07x)
16	133s (16.33x)	129.5s (16.81x)	224.5s (16.35x)
32	85s (25.55x)	88.5s (24.60x)	125.5s (29.25x)

Table 1.5: Total Texture I/O as a Function of Thread Count. For a given texture cache size, increasing the number of threads will in general increase the amount of disk I/O, since the total texture working set will be larger. In practice, I/O grows very slowly unless the cache is quite small.

Threads	80 MB Cache	32 MB Cache
1	185.45 MB (1x)	7.29 GB (1x)
4	185.45 MB (1.00x)	7.32 GB (1.00x)
16	185.70 MB (1.00x)	8.12 GB (1.11x)
32	185.36 MB (1.00x)	9.46 GB (1.30x)

1.6.7 EFFECT OF THREADS ON I/O

In general, we expect that given a fixed size texture cache, adding more threads will make the cache less effective: since threads work on tiles of the image, having more threads running concurrently means that more tiles are being rendered concurrently, which in turn means that more distinct parts of the scene's texture maps are being accessed and the overall working set is larger. In order to understand how much this factor affects the texture cache performance, we measured how total I/O varied as the number of threads varied for a few cache sizes; see Table 1.5.

To our surprise, more threads generally had little effect on the amount of I/O required; it was only in the case of a very small 32 MB texture cache that I/O increased to a meaningful degree with a 32 threads.

1.6.8 COMPARISON TO OTHER CACHE DESIGNS

We have demonstrated that our implementation scales well; of course, one may ask whether the complexity was necessary. Would a more straightforward implementation based on mutexes have done as well? The experience of coming to our current implementation taught us that the answer is "no", but for completeness, we compared the performance of a few alternatives (some of which correspond to approaches we implemented before coming to the one described in this document).

Table 1.6: Performance with a Partitioned Texture Cache. The texture cache is split into partitions, each of which is protected by a reader–writer mutex. Measurements used 32 threads and are reported as a multiple of single-thread performance with our RCU-based implementation. Scalability is terrible and increasing the number of partitions doesn't help much.

Partitions	160 MB Cache	64 MB Cache
128	492s (4.37x)	486s (4.11x)
1024	352s (6.11x)	355s (5.62x)
8192	366s (5.87x)	353s (5.66x)

One simple way to address the issue of concurrent texture cache access by multiple threads would be to have a separate texture cache per thread. This approach has two big problems: excessive memory usage and increased I/O. A 40 MB texture cache for each of 32 threads adds up to 1.25 GB—more memory than would be used for preloading all textures for the San Miguel scene. Further, with separate caches, total I/O would increase substantially, since any tile that was accessed by multiple threads would be read from disk independently and redundantly. For anything more than a few threads, this approach isn't a good one.

A straightforward way to make a shared texture cache work well with multiple threads accessing it is to use a single reader-writer mutex to protect it: on access, threads acquire the reader lock and are free to lookup values in the cache. Multiple threads can hold a reader lock at the same time, and when a tile is to be added or tiles must be freed, the thread acquires a writer lock, which prohibits other threads from accessing the cache while its contents are being modified.

Not surprisingly, performance with this approach is terrible: with a 160 MB texture cache (thus completely avoiding tile frees) and 32 threads, rendering time was 14942s. (In other words, 14.3% the performance of rendering with a single thread, or 0.7% of the performance of rendering with our implementation with 32 threads.) Not only was 73% of the total execution time spent in the kernel, but the time spent running "user" code was 3.3× more than with our texture cache; all of this was presumably due to the highly contended mutex and the cost of cache coherence between the CPUs as they all competed for access to it.

We also implemented a partitioned texture cache, where tiles are stored in a hash table as in our implementation, but where the hash table is split into a fixed number of partitions, each of which has its own reader-writer mutex. In turn, threads acquire the mutex for the part of the hash table they're accessing. This approach has less mutex contention compared to a single mutex. Table 1.6 shows the results with 32 threads for two different cache sizes and four different degrees of partitioning, where speedup is measured with respect to a single thread.

Results are poor; $4 - 6\times$ speedups with 32 threads on 16 cores is not very good. From the performance logs, we noted that over 40% of execution time was spent in the `TextureCache::Texel()` method. (With our implementation, it's just a few percent.) Even at 8192-way partitioning, where mutex contention is relatively low, the overhead of cache-coherence operations as mutexes are acquired and released is significant.

In a last attempt to make a partition-based approach work well, we tried adding per-thread caches of a few tiles that were filled from the shared cache; the idea was that these would reduce contention for the shared mutexes and improve scalability. In practice, hit rates for those caches were only around 50% and there was no meaningful scalability improvement.

We note that the partitioning approach did perform well when used on a system with just 4 CPU cores; this was the first approach we tried, and results on a 4 core system led us to believe that it might be a reasonable one—an illusion that was quickly shattered when we started running tests on the 16 core system.

1.7 CONCLUSION

We have described a texture cache implementation based on a lock-free hash table that uses read-copy update to reclaim texture tiles. It scales well to at least 32 threads and significantly reduces the amount of memory needed for textures for rendering versus loading all textures at startup time.

The overall efficiency of the renderer affects the rate of texel requests that must be served by the texture cache, and in turn, how much disk read latency affects performance. `pbrt` is fairly efficient, but isn't as fast as highly-optimized commercial systems.[8] It would be interesting to measure the performance of our texture cache implementation with a faster rate of texture lookups.

It would also be interesting to measure our implementation's parallel scalability on systems with even more CPU cores, including multi-socket systems and "many core" processors like Xeon Phi. Given that there are no hints of scalability starting to fall off up to 16 cores and 32 threads, we have some optimism that scalability will continue to be good. On the other hand, our early optimism about the partitioned cache's viability based on benchmarks on a 4 core system taught us to be cautious. If readers have access to interesting systems with more cores, we'd be interested to help do these measurements.

Our choice of tile replacement algorithm probably merits further investigation; an approach that more accurately determines which tiles are least-recently-used may be able to pay back the costs of increased cache coherency overhead through less disk I/O. It would also be interesting to see if freeing only a single tile when a tile was needed was viable—such an approach might improve performance by always having as many entries as possible in the texture cache. (See Exercise 1.4 for further discussion of these topics.)

Finally, we note that operating systems provide the ability to "memory map" files, where a section of the virtual address space can be allocated for a file and

8 As one data point: we have done an experimental integration of the highly-optimized *Embree* ray intersection routines (Wald et al. 2014) with `pbrt`; intersection tests using their implementation were $2.5 - 3\times$ faster than ours.

where the file contents are read into RAM on demand when accessed. When memory gets low, pages are claimed. One advantage of this approach is that it naturally uses as much RAM as is free. It would be interesting to compare a texture cache built using this mechanism to ours as well.

1.8 FURTHER READING

Peachey's (1990) technical report on texture caching is an excellent read; it describes the texture cache implementation used in Pixar's RenderMan renderer through the 1980s and 1990s. Christensen et al. (2003) describe the use of indirect ray differentials for geometry and texture caching; they used per-thread texture caches on a system with just a few CPU cores.

Suykens and Willems (2001) discuss issues related to computing ray differentials for diffuse and glossy indirect rays. Recently, Belcour et al. (2017) developed a more principled approach for filtering texture with both forward and bidirectional path tracing. An interesting observatrion from their work was that when connecting two vertices with bidirectional path tracing, it's necessary to compute new filtered texture values for those two vertices to get good results.

The scalable approach we implemented for recycling texture tiles is based on ideas from the "read-copy update" technique used in the Linux kernel (and elsewhere). The paper by McKenney and Slingwine (1998) and McKenney and Walpole's article in *Linux Weekly News* (2007) are both good references about RCU. For more information about not only read-copy update but also general topics in scalable parallel programming, see McKenney's book (2017) and Jeff Preshing's blog (*http://preshing.com/*), both of which cover these topics well.

Michael (2002) describes a lock-free hash table that directly supports element deletion. See also Triplett et al. (2011), who describe a lock-free hash table that also supports resizing. An interesting approach to lock-free hash tables is *CPHash*, described by Metreveli et al. (2011), where the hash table is partitioned across CPUs (so that each CPU can access its partition without locking) and message passing between CPUs is used to communicate requests and results.

1.9 EXERCISES

1.1 With the current requirement of power-of-2 tile sizes, some texture formats make poor use of the fixed-size allocation for tiles. For example, single channel `Y8` textures are 64×64 texels, wasting $2/3$ of the available space in the fixed tile size. Modify the texture cache so that non-power-of-2 tile sizes can be used; for the `Y8` case, tiles could be 110×110 texels, wasting just 188 bytes. Test your change with one or more scenes. (Be sure to use scenes that don't just use texture formats that already fit the tile size exactly, like `RGB8`.) How does your change affect cache hit rates and overall rendering performance?

Next, try using an approach like `RadicalInverse()` does to ensure that the divides and modulus operations for texel addressing with your scheme are done with compile-time constants and can be performed efficiently. (Recall the discussion of this topic on p. 447 of the third edition.) How much performance benefit does that modification give?

1.2 In our implementation, found it worthwhile to special-case the low-resolution images at high levels of the image pyramid, thus not needing to use tile memory for them. Is it also worth having a more memory-efficient representation for texture tiles with the same texel value throughout? Investigate how often such tiles are found in practice and estimate whether this would be worthwhile. How does the size of tiles affect your conclusions?

1.3 Our implementation's fixed tile size of 12 kB (corresponding to, e.g., 64×64 8-bit RGB pixels) was an attempt to balance disk I/O overhead with memory efficiency. Smaller tiles would lead to more I/O operations, but would use a given amount of cache memory more effectively by reading in fewer texels that ended up never being accessed. (And for larger tiles, conversely on both fronts.) Investigate the performance impact of different-sized tiles. How does the latency for a disk read affect your results?

1.4 Our implementation approximates a least-recently-used tile replacement policy by marking all active tiles when the number of free tiles gets low, clearing marks when tiles are accessed, and freeing the tiles that are still marked later. Modify the implementation to more accurately track usage, for example by storing a "last accessed" timestamp in each texture tile. Update the tile freeing logic to use this information to decide which tiles to free. Do you see lower texture cache miss rates due to better choices of which tiles to free? Is any measurable overhead added to tile lookups?

One advantage of an approach like this one is that it's possible to decide ahead of time exactly how many tiles to free. On one hand, it's better to free few tiles each time to maximize the number of remaining tiles, in case any of them are needed again soon. On the other hand, there is some fixed overhead to freeing tiles, which means that we don't want to perform too many tile freeing operations. Experiment with varying the number of freed tiles and measure system performance. What is the right trade-off for your implementation?

1.5 In our evaluation, we exclusively used forward path tracing, where all paths start out fairly coherently. BDPT and MLT both have somewhat different access patterns to the scene. For example, while BDPT still creates eye paths in buckets, light paths from BDPT are fairly irregular (especially if well-distributed sampling patterns are used). MLT has its own characteristics: large step mutations by design lead to very incoherent paths, but on the other hand, small step mutations are likely to be friendly to texture caches. Investigate how well the texture cache performs with those approaches in comparison to its performance with path tracing. Do the parameters to the light transport algorithm (large step probability, etc.) affect texture cache performance?

1.6 Replace `pbrt`'s ray differential computations with the technique developed by Belcour et al. (2017). Measure the impact on the texture cache of using their approach, both for regular path tracing, where cache behavior should be similar, and bidirectional techniques, where their method should give much better cache behavior.

1.10 BIBLIOGRAPHY

Belcour, L., L.-Q. Yan, R. Ramamoorthi, and D. Nowrouzezahrai. 2017. Antialiasing complex global illumination effects in path-space. *ACM Transactions on Graphics*, 36 (1).

Christensen, P. H., D. M. Laur, J. Fong, W. L. Wooten, and D. Batali. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Computer Graphics Forum (Eurographics 2003 Conference Proceedings) 22*(3), 543–52.

McKenney, P. E., ed. 2017. Is parallel programming hard, and, if so, what can you do about it? *https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html*

McKenney, P. E. and J. D. Slingwine. 1998. Read-copy update: using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, 509–18.

McKenney, P. E. and J. Walpole. 2007. What is RCU, fundamentally? *Linux Weekly News*, December 17. *https://lwn.net/Articles/262464/.*

Metreveli, Z., N. Zeldovich, and F. Kaashoek. 2011. CPHash: a cache-partitioned hash table. *MIT CSAIL Technical Report*.

Michael, M. M. 2002. High performance dynamic lock-free hash tables and list-based sets. *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02).*

Peachey, D. 1990. Texture on demand. *Pixar Animation Studios Technical Memo 217. http://graphics.pixar.com/library/TOD/paper.pdf*

Stafford, D. 2011. Better bit mixing—improving on MurmurHash3's 64-bit finalizer. *http://zimbry.blogspot.ch/2011/09/better-bit-mixing-improving-on.html*

Suykens, F., and Y. Willems. 2001. Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering,* 257–68.

Triplett, J., P. E. McKenney, and J. Walpole. 2011. Resizable, scalable, concurrent hash tables via relativistic programming. *Proceedings of the 2011 USENIX Conference*.

Wald, I., S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014) 33*(4), 143:1–143:8.

1.11 REVISION HISTORY

5 March 2017: initial version posted.