APPENDIX A UTILITIES

In addition to all of the graphics-related code presented thus far, `pbrt` makes use of a number of general utility routines and classes. Although these are key to `pbrt`'s operation, it is not necessary to understand their implementation in detail in order to work with the rest of the system. This appendix describes the interfaces to these routines, including those that handle error reporting, memory management, the task sytem for parallel execution, and other basic infrastructure. The implementations of some of this functionality—the parts that are interesting enough to be worth delving into—are also discussed.

A.1 MAIN INCLUDE FILE

The `core/pbrt.h` file is included by all other source files in the system. It contains all global function declarations and inline functions, a few macros and numeric constants, and other globally accessible data. All files that include `pbrt.h` get a number of other included files in the process. This simplifies creation of new source files, almost all of which will want access to these extra headers. However, in the interest of compile time efficiency, we keep the number of these automatically included files to a minimum; the ones here are necessary for almost all modules.

⟨*Global Include Files*⟩ ≡

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

We also include files from the C++ standard library to get a few C++ classes that are used frequently. The `using` directive brings these classes into the default namespace.

⟨*Global Include Files*⟩ +≡
```
#include <string>
using std::string;
#include <vector>
using std::vector;
```

We will also define a macro that holds pbrt's current version number; this is used when printing the startup message in the main() function.

⟨*Global Constants*⟩ ≡
```
#define PBRT_VERSION "2.0.0"
```

A.1.1 UTILITY FUNCTIONS

A few short mathematical functions are useful throughout pbrt.

Linear Interpolation

Lerp() performs linear interpolation between two values v1 and v2, with the position given by the t parameter. When t is zero, the result is v1, and when t is one, the result is v2.

⟨*Global Inline Functions*⟩ +≡
```
inline float Lerp(float t, float v1, float v2) {
    return (1.f - t) * v1 + t * v2;
}
```

Notice that Lerp() is implemented as $(1-t)v_1 + tv_2$, rather than the more terse and potentially more computationally efficient form of $v_1 + t(v_2 - v_1)$. This is done to reduce floating-point error. If the magnitudes of v_1 and v_2 are substantially different, the difference $v_2 - v_1$ may have a substantial amount of floating-point roundoff error. When the resulting floating-point value is scaled by t and added to v_1, the result may be quite inaccurate. With the first formulation, not only is this problem avoided, but Lerp() returns *exactly* the values v_1 and v_2 when pos has values 0 and 1, respectively, and always returns a value in the range $[v_1, v_2]$ if t is in $[0, 1]$. These properties are not guaranteed by the second formulation, though other code usually at least implicitly assumes that linear interpolation will have these properties.

Clamping

Clamp() clamps the given value val to be between the values low and high:

⟨*Global Inline Functions*⟩ +≡
```
inline float Clamp(float val, float low, float high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

Clamp() 1000
Lerp() 1000

Modulus

`Mod()` computes the remainder of a/b. This function is handy since it behaves predictably and reasonably with negative numbers; the C and C++ standards leave the behavior of the `%` operator undefined in that case.

⟨*Global Inline Functions*⟩ +≡
```
inline int Mod(int a, int b) {
    int n = int(a/b);
    a -= n*b;
    if (a < 0) a += b;
    return a;
}
```

Converting between Angle Measures

Two simple functions convert from angles expressed in degrees to radians, and vice versa:

⟨*Global Inline Functions*⟩ +≡
```
inline float Radians(float deg) {
    return ((float)M_PI/180.f) * deg;
}
inline float Degrees(float rad) {
    return (180.f/(float)M_PI) * rad;
}
```

Base-2 Logarithms and Exponents

Because the math library doesn't provide a base-2 logarithm function, we provide one here, using the identity $\log_2(x) = \log x / \log 2$.

⟨*Global Inline Functions*⟩ +≡
```
inline float Log2(float x) {
    static float invLog2 = 1.f / logf(2.f);
    return logf(x) * invLog2;
}
```

Sometimes we need an integer-valued base-2 logarithm.

⟨*Global Inline Functions*⟩ +≡
```
inline int Log2Int(float v) {
    return Floor2Int(Log2(v));
}
```

Finally, bit manipulation techniques can be used to efficiently determine if a given integer is an exact power of two, or round an integer up to the next higher (or equal) power of two.

⟨*Global Inline Functions*⟩ +≡
```
inline bool IsPowerOf2(int v) {
    return (v & (v - 1)) == 0;
}
```

Mod() 1001
M_PI 1002

⟨*Global Inline Functions*⟩ +≡
```
inline uint32_t RoundUpPow2(uint32_t v) {
    v--;
    v |= v >> 1;    v |= v >> 2;
    v |= v >> 4;    v |= v >> 8;
    v |= v >> 16;
    return v+1;
}
```

Useful Constants

We explicitly redefine `M_PI` so that it is a 32-bit floating-point constant instead of using double precision and also define the useful constants $1/\pi$, $1/2\pi$, and $1/4\pi$.

⟨*Global Constants*⟩ +≡
```
#ifdef M_PI
#undef M_PI
#endif
#define M_PI       3.14159265358979323846f
#define INV_PI     0.31830988618379067154f
#define INV_TWOPI  0.15915494309189533577f
#define INV_FOURPI 0.07957747154594766788f
```

Finally, an `INFINITY` value is defined to be `FLT_MAX`, the largest representable floating-point number:

⟨*Global Constants*⟩ +≡
```
#ifndef INFINITY
#define INFINITY FLT_MAX
#endif
```

Floating-Point to Integer Conversion

A number of utility functions are provided for converting floating-point values to integers:

- `Float2Int(f)`: This is the same as the basic cast `(int)f`.
- `Round2Int(f)`: This rounds the floating-point value `f` to the nearest integer, returning the result as an `int`.
- `Floor2Int(f)`: The first integer value less than or equal to `f` is returned.
- `Ceil2Int(f)`: Similarly, the first integer value greater than or equal to `f` is returned.

Their default implementations are straightforward. On some architectures, these conversions can be surprisingly expensive, in which case specialized implementations can be worthwhile, especially because these functions are used in the inner loops of the system such as the texture filtering code and the film update code. See the "Further Reading" section for further details.

A.1.2 PSEUDO-RANDOM NUMBERS

`pbrt` uses a custom pseudo-random number generator rather than calling the one provided by the system. This is worth the extra effort for two reasons. First, it ensures that

INFINITY 1002
M_PI 1002

pbrt produces the same results regardless of machine architecture and C library implementation. In addition, many systems provide random number generation routines with poor statistical distributions.

The random number generator used in pbrt is the "Mersenne Twister" by Makoto Matsumoto and Takuji Nishimura. The code that implements the random number generator is both complex and subtle, and we will not attempt to explain it here. Nevertheless, it is one of the best random number generators known, can be implemented efficiently, and has a period of $2^{19937} - 1$ before it repeats the series again. A pointer to the paper describing its algorithm can be found in the "Further Reading" section.

The random number generator class, RNG, provides three methods in addition to its constructor:

- `RNG(uint32_t seed = 0)`: An optional *seed* value can be provided to the constructor; two RNGs seeded with different values will produce independent sequences of pseudo-random numbers.
- `void Seed(uint32_t seed)`: Alternatively, an RNG can be re-seeded after construction.
- `float RandomFloat() const`: A pseudo-random floating-point number in the range $[0, 1)$ is returned.
- `unsigned long RandomUInt() const`: A pseudo-random number in the range $[0, 2^{32})$ is returned.

It is defined in the files core/rng.h and core/rng.cpp.

A.2 IMAGE FILE INPUT AND OUTPUT

Many image file formats have been developed over the years, but for pbrt's purposes we are mainly interested in those that support imagery represented by floating-point pixel values. Because the images generated by pbrt will often have a large dynamic range, such formats are crucial for being able to store the computed radiance values directly; legacy image file formats, such as those that store 8 bits of data for red, green, and blue components to represent colors in the range $[0, 1]$, aren't a good fit for physically based rendering needs.

pbrt uses the OpenEXR and PFM standards as the principal file formats for images. OpenEXR is a floating-point file format designed by Industrial Light and Magic for use in their movie productions (Kainz, Bogart, and Hess 2002). A library that reads and writes this format is freely available, and support for it is available in a number of other tools. We chose this format because it has a clean design, is easy to use, and has first-class support for floating-point image data. If the preprocessor macro PBRT_HAS_OPENEXR is defined, then the OpenEXR image input and output routines are used by the system. PFM is a floating-point format based on the PPM file format; it is very easily read and written.

For convenience, pbrt also has support to read and write TGA format files. TGA is not a high-dynamic-range format like OpenEXR, but it is convenient, especially as an input format for low-dynamic-range texture maps.

pbrt provides a ReadImage() function that takes the filename to read from and pointers to two integers that will be initialized with the image resolution. It returns a freshly allocated array of RGBSpectrum objects. It will read the given file as an OpenEXR, PFM, or TGA file, depending on the suffix of the filename.

This function uses RGBSpectrum for the return values, not Spectrum. The primary client of this function is the image texture mapping code in pbrt, which stores texture maps as RGBSpectrum values, even when pbrt is compiled to do full-spectral rendering, so returning RGBSpectrum values is a natural approach. We also made this decision under the expectation that the image files being read would be in RGB or another three-channel format, so that returning RGB values wouldn't discard spectral information; if calling code wants to store full Spectrum values, then it can convert from RGB to the full-spectral representation itself, with no loss of information. If pbrt was extended to support a full-spectral image format for textures, then a variant of this function that did return Spectrum values would be necessary.

⟨*ImageIO Declarations*⟩ ≡
```
RGBSpectrum *ReadImage(const string &name, int *xSize, int *ySize);
```

The WriteImage() function takes a filename to be written, a pointer to the beginning of the pixel data, and information about the resolution of the image. The XRes and YRes variables hold the dimensions of the image to be written. If a portion of a larger image is being written, the dimensions of the larger image and the offset of the portion to be written are passed in the remaining variables. Note that these variables are not used to *save* a subimage, but rather to indicate that the pixels being passed are themselves part of a larger image. This information is written into the image header, so that the subimages can later be assembled into a single image by tools like the tools/exrassemble.cpp program in the pbrt distribution.

⟨*ImageIO Declarations*⟩ +≡
```
void WriteImage(const string &name, float *pixels, float *alpha,
    int XRes, int YRes, int totalXRes, int totalYRes, int xOffset,
    int yOffset);
```

We will not show the code that interfaces with the OpenEXR libraries or the code that implements PFM and TGA file I/O. This code can be found in the file core/imageio.cpp. The TGA implementation is based on GPL TGA code by Jaakko Keranen and Daniel Swanson; Jiawen "Kevin" Chen provided the PFM reader and writer; and the OpenEXR distribution is available from *www.openexr.com*.

A.3 COMMUNICATING WITH THE USER

The functions and classes in this section all communicate information to the user. In addition to consolidating functionality like printing progress bars, hiding user communication behind a small API like the one here also permits easy modification of the communication mechanisms. For example, if pbrt were embedded in an application that had a graphical user interface, errors might be reported via a dialog box or a routine provided by the parent application. If printf() calls were strewn throughout the system, it would be more difficult to make the two systems work together well.

ReadImage() 1004
RGBSpectrum 279
Spectrum 263
WriteImage() 1004

A.3.1 ERROR REPORTING

pbrt provides four functions for reporting anomalous conditions. In order of increasing severity, they are Info(), Warning(), Error(), and Severe(). These functions are defined in the files core/error.h and core/error.cpp. All of them take a formatting string as their first argument and a variable number of additional arguments providing values for the format. The syntax is identical to that used by the printf family of functions. For example, if the variable rayNum has type int, then the following call could be made:

```
Info("Now tracing ray number %d", rayNum);
```

core/pbrt.h includes this header file, as these functions are useful to have available in almost all parts of the system.

⟨*Global Include Files*⟩ +≡
```
#include "error.h"
```

We will not show the implementation of these functions here because they are a straightforward application of the C++ variable argument processing functions that in turn calls a common function to print the full error string. For sufficiently severe errors, the program aborts.

pbrt also has its own version of the standard assert() macro, named Assert(). It checks that the given expression's value evaluates to true; if not, Severe() is called with information about the location of the assertion failure. Assert() is used for basic sanity checks where failure indicates little possibility of recovery. In general, assertions should be used to detect internal bugs in the code, not expected error conditions (such as invalid scene file input), because the message printed will likely be cryptic to anyone other than the developer.

⟨*Global Inline Functions*⟩ +≡
```
#ifdef NDEBUG
#define Assert(expr) ((void)0)
#else
#define Assert(expr) \
    ((expr) ? (void)0 : \
        Severe("Assertion \"%s\" failed in %s, line %d", \
               #expr, __FILE__, __LINE__))
#endif // NDEBUG
```

A.3.2 REPORTING PROGRESS

Assert() 1005
ProgressReporter 1006
SamplerRenderer::Render() 27
Severe() 1005

The ProgressReporter class gives the user feedback about how much of a task has been completed and how much longer it is expected to take. For example, the implementation of the SamplerRenderer::Render() method uses a ProgressReporter to show how many of the camera rays have been traced. The current implementation prints a row of plus signs, the elapsed time, and the estimated remaining time. Its implementation is in the files core/progressreporter.h and core/progressreporter.cpp.

The constructor takes the total number of units of work to be done (e.g., the total number of camera rays that will be traced) and a short string describing the task being performed. It also takes an integer indicating the number of characters to use for the progress bar.

⟨*ProgressReporter Public Methods*⟩ ≡
```
ProgressReporter(int totalWork, const string &title,
                 int barLength = 58);
```

Once the `ProgressReporter` has been created, each call to its `Update()` method signifies that one unit of work has been completed. An optional integer value can be passed to indicate that multiple units have been done.

⟨*ProgressReporter Public Methods*⟩ +≡
```
void Update(int num = 1);
```

The `ProgressReporter::Done()` method lets the user know that the task is complete:

⟨*ProgressReporter Public Methods*⟩ +≡
```
void Done();
```

A.3.3 SIMPLE FLOAT FILE READER

A number of places in the `pbrt` code need to read simple text-format files with floating-point values. Examples include loading measured spectral data stored in text files and the measured BRDF material in Section 9.2.4. The `ReadFloatFile()` function parses simple text files of whitespace-separated numbers, returning the values found in the given `vector`. The parsing code ignores all text after a hash mark # to the end of its line to allow comments.

⟨*floatfile.h**⟩ ≡
```
bool ReadFloatFile(const char *filename, vector<float> *values);
```

A.4 PROBES AND STATISTICS

Collecting data about the run-time behavior of the system can provide a substantial amount of insight into its behavior and opportunities for improving its performance. For example, we might want to gather data to compute a histogram of how long each camera ray takes to compute a radiance value for. If some unexpectedly take much more time than others, then drilling down further may reveal a subtle bug in the system.

In support of this type of data gathering, we have defined a large number of *probes* and annotated code throughout `pbrt` to use them. The intent behind these probes is to easily make accessible relevant data from interesting points in the code's execution that we may want further information about. The programmer, in turn, can then enable a chosen set of probe points and write a small amount of code to gather the data of interest from them. Code related to these probes is defined in the files `core/probes.h` and `core/probes.cpp`.

ProgressReporter::
Done() 1006

Scene::Intersect() 23

As a concrete example, there are two probes in `Scene::Intersect()`; its actual implementation is:

```
bool Intersect(const Ray &ray, Intersection *isect) const {
    PBRT_STARTED_RAY_INTERSECTION(&ray);
    bool hit = aggregate->Intersect(ray, isect);
    PBRT_FINISHED_RAY_INTERSECTION(&ray, isect, int(hit));
    return hit;
}
```

(We have generally not included probes in the code presented in the book in the interest of brevity.) If we associate code with these probes, we can gather a number of different interesting types of information, including the total number of rays traced, the fraction of rays traced that found an intersection, the origins and directions of the rays traced (which could be useful to visualize), the average time to compute intersections for rays, and so forth.

Gathering this data can itself have a noticeable impact on performance. Therefore, `pbrt` provides a number of options for what is done with the probes in the system. In optimized builds (and more generally, when the `PBRT_PROBES_NONE` macro is defined), all of the probes disappear; all of the preprocessor macros are defined with an empty implementation. Thus, the probes have no performance impact.

⟨*Statistics Disabled Declarations*⟩ ≡

```
#define PBRT_STARTED_RAY_INTERSECTION(ray)
#define PBRT_FINISHED_RAY_INTERSECTION(ray, isect, hit)
#define PBRT_STARTED_RAY_INTERSECTIONP(ray)
#define PBRT_FINISHED_RAY_INTERSECTIONP(ray, hit)
⟨Remainder of disabled probes declarations⟩
```

A second option is provided when the `PBRT_PROBES_COUNTERS` macro is defined when the system is built. For this case, we have chosen a small number of generally interesting probes and implemented short functions that are called for them. For example, we track the total number of shapes and triangles created during rendering.

⟨*Statistics Counters Probe Definitions*⟩ ≡

```
void PBRT_CREATED_SHAPE(Shape *) {
    ++shapesMade;
}
void PBRT_CREATED_TRIANGLE(Triangle *) {
    ++trianglesMade;
}
```

These counters are tracked by simple statistics classes that associate descriptive text with the values being tracked and print out a report of results at the end of rendering. These classes also ensure that the statistics are collected in a safe manner given multithreaded execution; atomic operations are used to increment counters. Using them does cause a noticeable performance impact from having multiple threads frequently modifying the same memory location—see Section A.9.1 for discussion of this issue. A more scalable statistics system might store separate statistics for each thread in private per-thread memory and then compute aggregate results at the end of rendering.

Shape 108
Triangle 139

⟨*Statistics Counters Probe Declarations*⟩ ≡
```
static StatsCounter shapesMade("Shapes", "Total Shapes Created");
static StatsCounter trianglesMade("Shapes", "Total Triangles Created");
```

The final option for probes in `pbrt` is based on `dtrace`, a remarkable system developed by Cantrill, Shapiro, and Leventhal (2004).[1] When `pbrt` is compiled with `dtrace` support, each probe is initially converted to a small number of assembly language "no-op" instructions. When the program executes normally, these no-ops introduce a very small performance penalty—under 1% in our experience with `pbrt`'s instrumentation.

If the program is executed with data gathering enabled for some of the probes, `dtrace` modifies the executable's object code so that the no-ops for the enabled probes instead call out to data-gathering routines. These data-gathering routines are specified in a scripting language that was designed to be tailored to gathering and synthesizing performance information from complex software systems.

The `dtrace` probes corresponding to the probes above are defined as follows:

⟨*dtrace.d**⟩ ≡
```
provider PBRT {
probe started_ray_intersection(const struct Ray *);
probe finished_ray_intersection(const struct Ray *,
                                const struct Intersection *, int hit);
probe started_ray_intersectionp(const struct Ray *);
probe finished_ray_intersectionp(const struct Ray *, int hit);
⟨Remainder of dtrace probes⟩
};
```

Here is a very simple `dtrace` script; it attaches a snippet of code to the `started_ray_intersection` probe and provides another one to run when `pbrt` exits. This script counts the total number of rays that are traced and prints the total at the end.

```
uint64_t num_rays;

:::started_ray_intersection {
    ++num_rays;
}

dtrace:::END {
    printf("Total intersection tests: %d\n", num_rays);
}
```

When run on the TT car scene, this script produces the output

```
Total intersection tests: 2035434
```

Intersection 186
Ray 66

1 As of this writing, `dtrace` is available under the Mac OS X®, Solaris®, and FreeBSD® operating systems, and a port to Linux® is in progress.

dtrace transparently handles multithreaded execution, computing correct results when multiple threads simultaneously execute the code associated with a probe. dtrace scripts also provide a number of useful language features for aggregating data, producing histograms, only collecting data under certain circumstances, and so forth. For example, a relatively short script could be written to produce histograms of how many ray–triangle intersection tests were performed for rays traced for specular reflection versus those traced for shadows.

The available dtrace documentation has more information and ideas about how it can be used to investigate the behavior of complex systems. See also the example dtrace scripts provided with the pbrt distribution in the src/dtrace/ directory.

A.5 MEMORY MANAGEMENT

Memory management is often a complex issue in a system written in a language without garbage collection. The situation is mostly simple in pbrt, since most dynamic memory allocation is done as the scene description file is parsed, and most of this memory remains in use until rendering is finished. Nevertheless, there are a few issues related to memory management that warrant classes and utility routines to address them. Many of these issues are performance related, although an automatic reference-counting class is also provided to track the lifetimes of objects that may be referenced by multiple pointers in different parts of the system.

A.5.1 VARIABLE STACK ALLOCATION

Sometimes it is necessary to allocate a variable amount of memory that will be used temporarily in a single function but isn't needed after the function returns. If only a small amount of memory is needed, the overhead of new and delete (or malloc() and free()) may be high relative to the amount of actual computation being done. Instead, it is frequently more efficient to use alloca(), which efficiently allocates memory on the stack with just a few machine instructions. This memory is automatically deallocated when the function exits, which also saves bookkeeping work in the routine that uses it.

alloca() is an extremely useful tool, but there are two pitfalls to be aware of when using it. First, because the memory is deallocated when the calling routine returns, the pointer must not be returned from the routine or stored in a data structure with a longer lifetime than the function that allocated it. (However, the pointer may be passed to functions called by the allocating function.) Second, stack size is limited, and so alloca() shouldn't be used for more than a few kilobytes of storage. Unfortunately, there is no way to detect the error condition when more space is requested from alloca() than is available on the stack, so it's important to be conservative with its use.

pbrt provides a macro that makes it easy to allocate space for a given number of objects of a given type.

⟨*Global Macros*⟩ ≡

```
#define ALLOCA(TYPE, COUNT) (TYPE *)alloca((COUNT) * sizeof(TYPE))
```

A.5.2 REFERENCE-COUNTED OBJECTS

In programming languages that do not provide automatic memory management, tricky situations can arise when multiple pointers to some object exist. We would like to free an object as soon as it is no longer needed by any other object (but no sooner), so that both memory leaks and errors due to memory corruption are avoided.

As long as there aren't circular references (e.g., object A holds a reference to object B, which holds a reference to object A), a good solution to this problem is to use *reference counting*. Objects that may be pointed to in multiple places store a counter that is incremented when another object stores a reference to it and decremented when a reference goes away (e.g., due to the holding object being destroyed). When its reference count goes to zero, memory for the object can be safely freed.

We will define two classes to make it easy to use reference-counted objects in `pbrt`. An object should inherit from the `ReferenceCounted` class if it is to be managed via reference counting. The `ReferenceCounted` class adds an `nReferences` field, which is atomically incremented and decremented so that the class operates as expected when multiple threads are executing simultaneously. The count will be managed by the `Reference` class, defined in the following.

⟨*Memory Declarations*⟩ ≡
```
class ReferenceCounted {
public:
    ReferenceCounted() { nReferences = 0; }
    AtomicInt32 nReferences;
};
```

Rather than holding a pointer to a reference-counted object, an instance of the `Reference` template is used to hold the reference. The `Reference` template handles updates to the reference count as appropriate. For example, consider the following function:

```
void func() {
    Reference<Foo> r1 = new Foo;
    Reference<Foo> r2 = r1;
    r1 = new Foo;
    r2 = r1;
}
```

In the first line, a `Foo` object is allocated; `r1` holds a reference to it, and the object's `nReferences` count is one. A second reference to the object is made in the second line; `r1` and `r2` refer to the same `Foo` object, which now has a reference count of two. Next, a new `Foo` object is allocated. When a reference to it is assigned to `r1`, the reference count of the original object is decremented to one. Finally, in the last line, `r2` is assigned to refer to the newly allocated `Foo` object. The original `Foo` object now has zero references and is automatically deleted. At the end of the function, when both `r1` and `r2` go out of scope, the reference count for the second `Foo` object goes to zero, causing it to be freed as well. A few C++ language features make this all work transparently.

AtomicInt32 1036
Reference 1011
ReferenceCounted 1010

⟨*Memory Declarations*⟩ +≡
```
template <typename T> class Reference {
public:
    ⟨Reference Public Methods 1011⟩
private:
    T *ptr;
};
```

The constructors are straightforward. They just need to increment the reference count:

⟨*Reference Public Methods*⟩ ≡ 1011
```
Reference(T *p = NULL) {
    ptr = p;
    if (ptr) AtomicAdd(&ptr->nReferences, 1);
}
```

⟨*Reference Public Methods*⟩ +≡ 1011
```
Reference(const Reference<T> &r) {
    ptr = r.ptr;
    if (ptr) AtomicAdd(&ptr->nReferences, 1);
}
```

When a `Reference` is assigned to hold a different reference, it is necessary to decrement the old reference count and increment the count of the new object. The increments and decrements are ordered carefully in the following code so that an assignment like `r1 = r1` doesn't inadvertently delete the object `r1` is referring to if `r1` is the only reference.

⟨*Reference Public Methods*⟩ +≡ 1011
```
Reference &operator=(const Reference<T> &r) {
    if (r.ptr) AtomicAdd(&r.ptr->nReferences, 1);
    if (ptr && AtomicAdd(&ptr->nReferences, -1) == 0) delete ptr;
    ptr = r.ptr;
    return *this;
}
```

⟨*Reference Public Methods*⟩ +≡ 1011
```
Reference &operator=(T *p) {
    if (p) AtomicAdd(&p->nReferences, 1);
    if (ptr && AtomicAdd(&ptr->nReferences, -1) == 0) delete ptr;
    ptr = p;
    return *this;
}
```

⟨*Reference Public Methods*⟩ +≡ 1011
```
~Reference() {
    if (ptr && AtomicAdd(&ptr->nReferences, -1) == 0)
        delete ptr;
}
```

AtomicAdd() 1038
Reference 1011
Reference::ptr 1011
ReferenceCounted::
nReferences 1010

We would like to treat `References` as much like pointers as possible. For example, if the `Foo` class has a `Foo::bar()` method and `r1` is a `Reference<foo>` object, we'd like to be

able to write expressions like `r1->bar()`. The following methods take care of these details. Furthermore, the `operator bool` method makes it possible to check to see if a reference points to a `NULL` object with code like `if (!r) ...`.

⟨*Reference Public Methods*⟩ +≡ 1011

```
T *operator->() { return ptr; }
const T *operator->() const { return ptr; }
operator bool() const { return ptr != NULL; }
const T *GetPtr() const { return ptr; }
```

A.5.3 CACHE-FRIENDLY MEMORY USAGE

While the speed of modern CPUs has continued to increase at roughly the rate predicted by Moore's law (doubling every 18 to 24 months), modern memory technologies haven't been able to keep up with this growth. The speed at which memory can respond to read requests has been getting faster at a rate of roughly 10% per year. Today, a CPU will typically have to wait hundreds of execution cycles to read from main memory. The CPU is usually idle for much of this time, so a substantial amount of its computational potential may be lost.

One of the most effective techniques to address this problem is the judicious use of small, fast cache memory located either in the CPU itself or closer to it than main memory. The cache holds recently accessed data and is able to service memory requests much faster than main memory.

Because of the high penalty for accessing main memory, designing algorithms and data structures that make good use of the cache can substantially improve overall system performance. This section will discuss general programming techniques for improving cache performance. These techniques are used in many parts of `pbrt`, particularly the `KdTreeAccel`, `MIPMap`, and `ImageFilm`. We assume that the reader has a basic familiarity with computer architecture and caching technology; readers needing a review are directed to a computer architecture text such as Hennessy and Patterson (1997). In particular, the reader should be generally familiar with topics like cache lines, cache associativity, and the difference between compulsory, capacity, and conflict misses.

One easy way to reduce the number of cache misses incurred by `pbrt` is to make sure that some key memory allocations are aligned with the blocks of memory that the cache manages. Figure A.1 illustrates the basic technique. `pbrt`'s overall performance was improved by approximately 3% when allocation for the kd-tree accelerator in Section 4.5 was rewritten to use cache-aligned allocation. The `AllocAligned()` and `FreeAligned()` functions provide an interface to allocate and release cache-aligned memory blocks. If the preprocessor constant `PBRT_L1_CACHE_LINE_SIZE` is not set, a default cache line size of 64 bytes is used, which is representative of many current architectures.

⟨*Global Constants*⟩ +≡

```
#ifndef PBRT_L1_CACHE_LINE_SIZE
#define PBRT_L1_CACHE_LINE_SIZE 64
#endif
```

AllocAligned() 1013
FreeAligned() 1013
ImageFilm 404
KdTreeAccel 228
MIPMap 530
PBRT_L1_CACHE_LINE_SIZE 1012
Reference::ptr 1011

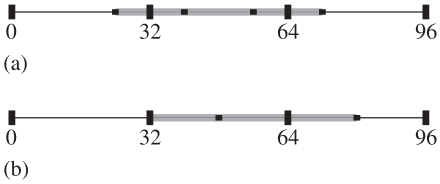

Figure A.1: The Layout of Three 16-Byte Objects in Memory on a System with 32-Byte Cache Lines. Cache-aligned memory allocation ensures that the address returned by the memory allocation routines are aligned with the start of a cache line. (a) The starting address is not cache aligned; the first and last of the three objects span two cache lines, such that two cache misses may be incurred when accessing their elements. (b) The starting address is cache aligned, guaranteeing that a maximum of one cache miss will be incurred per object.

Unfortunately there aren't portable methods to allocate memory aligned to a particular granularity. For the operating systems that do provide aligned memory allocation routines, `AllocAligned()` calls them. Otherwise, `AllocAligned()` allocates enough additional memory to ensure that the returned pointer is aligned to the desired granularity and to store the original pointer returned by `malloc()`. This pointer will be needed later to pass to `free()` when the memory is freed.

⟨*Memory Allocation Functions*⟩ ≡
```
void *AllocAligned(size_t size) {
#if defined(WIN32)
    return _aligned_malloc(size, PBRT_L1_CACHE_LINE_SIZE);
#elif defined (__OpenBSD__) || defined(__APPLE__)
    ⟨Allocate excess memory to ensure an aligned pointer can be returned⟩
#else
    return memalign(PBRT_L1_CACHE_LINE_SIZE, size);
#endif
}
```

A convenience routine is also provided for allocating a collection of objects so that code like `AllocAligned<Foo>(n)` can be written to allocate `n` instances of type `Foo`.

⟨*Memory Declarations*⟩ +≡
```
template <typename T> T *AllocAligned(uint32_t count) {
    return (T *)AllocAligned(count * sizeof(T));
}
```

The routine for freeing aligned memory either calls the operating-system-specific routine or retrieves the pointer originally returned by `malloc()` and passes that to `free()`. We won't include its implementation here.

⟨*Memory Declarations*⟩ +≡
```
void FreeAligned(void *);
```

PBRT_L1_CACHE_LINE_SIZE 1012

Another family of techniques for improving cache performance is based on reorganizing data structures themselves. For example, using bit fields to reduce the size of a frequently

used data structure can be helpful. This approach improves the *spatial locality* of memory access at run time, since code that accesses multiple packed values won't incur more than one cache miss to get them all. Furthermore, by reducing the overall size of the structure, this technique can reduce capacity misses if fewer cache lines are consequently needed to store the structure.

If not all of the elements of a structure are frequently accessed, there are a few possible strategies to improve cache performance. For example, if the structure has a size of 128 bytes and the computer has 64-byte cache lines, two cache misses may be needed to access it. If the commonly used fields are collected into the first 64 bytes rather than being spread throughout, then no more than one cache miss will be incurred when only those fields are needed (Truong, Bodin, and Seznec 1998).

A related technique is *splitting*, where data structures are split into "hot" and "cold" parts, each stored in separate regions of memory. For example, given an array of some structure type, we can split it into two arrays, one for the more frequently accessed (or "hot") portions and one for the less frequently accessed (or "cold") portions. This way, cold data doesn't displace useful information in the cache except when it is actually needed.

Cache-friendly programming is a complex engineering task, and we will not cover all the variations here. Readers are directed to the "Further Reading" section of this appendix for more information.

A.5.4 ARENA-BASED ALLOCATION

Conventional wisdom says that the system's memory allocation routines (e.g., `malloc()` and `new()`) are slow, and that custom allocation routines for objects that are frequently allocated or freed can provide a measurable performance gain. However, this conventional wisdom seems to be wrong. Wilson et al. (1995), Johnstone and Wilson (1999), and Berger, Zorn, and McKinley (2001, 2002) all investigated the performance impact of memory allocation in real-world applications and found that custom allocators almost always result in *worse* performance than a well-tuned generic system memory allocation, in both execution time and memory use.

One type of custom allocation technique that has proved to be useful in some cases is *arena-based allocation*, which allows the user to quickly allocate objects from a large contiguous region of memory. In this scheme, individual objects are never explicitly freed; the entire region of memory is released when the lifetime of all of the allocated objects ends. This type of memory allocator is a natural fit for many of the objects in `pbrt`.

There are two main advantages to arena-based allocation. First, allocation is extremely fast, usually just requiring a pointer increment. Second, it can improve locality of reference and lead to fewer cache misses, since the allocated objects are contiguous in memory. A more general dynamic memory allocator will typically prepend a bookkeeping structure to each block it returns, which adversely affects locality of reference.

`pbrt` provides the `MemoryArena` class to implement this approach; it supports variable-sized allocation from the arena.

`MemoryArena` 1015

The MemoryArena quickly allocates memory for objects of variable size by handing out pointers into a preallocated block. It does not support freeing of individual blocks of memory, only freeing of all of the memory in the zone at once. Thus, it is useful when a number of allocations need to be done quickly and all of the allocated objects have similar lifetimes.

⟨*Memory Declarations*⟩ +≡
```
class MemoryArena {
public:
    ⟨MemoryArena Public Methods 1015⟩
private:
    ⟨MemoryArena Private Data 1015⟩
};
```

MemoryArena allocates memory in chunks of size MemoryArena::blockSize, the value of which is set by a parameter passed to the constructor. It maintains a pointer to the current block of memory and the offset of the first free location in the block.

⟨*MemoryArena Public Methods*⟩ ≡ 1015
```
MemoryArena(uint32_t bs = 32768) {
    blockSize = bs;
    curBlockPos = 0;
    currentBlock = AllocAligned<char>(blockSize);
}
```

MemoryArena also uses two vectors to hold pointers to blocks of memory that have been fully used as well as available blocks that were previously allocated but aren't currently in use.

⟨*MemoryArena Private Data*⟩ ≡ 1015
```
uint32_t curBlockPos, blockSize;
char *currentBlock;
vector<char *> usedBlocks, availableBlocks;
```

To service an allocation request, the allocation routine first rounds the requested amount of memory up so that it meets the computer's word alignment requirements.[2] The routine then checks to see if the current block has enough space to handle the request, allocating a new block if necessary. Finally, it returns the pointer and updates the current block offset.

AllocAligned() 1013
MemoryArena 1015
MemoryArena::blockSize 1015
MemoryArena::curBlockPos 1015
MemoryArena::currentBlock 1015

2 Some systems (such as those based on Intel® processors) can handle non-word-aligned memory accesses, but this is usually substantially times slower than word-aligned memory reads or writes. Other architectures do not support this at all and will generate a bus error if a nonaligned access is performed.

⟨*MemoryArena Public Methods*⟩ +≡ 1015
```
void *Alloc(uint32_t sz) {
    ⟨Round up sz to minimum machine alignment 1016⟩
    if (curBlockPos + sz > blockSize) {
        ⟨Get new block of memory for MemoryArena 1016⟩
        curBlockPos = 0;
    }
    void *ret = currentBlock + curBlockPos;
    curBlockPos += sz;
    return ret;
}
```

⟨*MemoryArena Public Methods*⟩ +≡ 1015
```
template<typename T> T *Alloc(uint32_t count = 1) {
    T *ret = (T *)Alloc(count * sizeof(T));
    for (uint32_t i = 0; i < count; ++i)
        new (&ret[i]) T();
    return ret;
}
```

Most modern computer architectures impose alignment requirements on the positioning of objects in memory. For example, it is frequently a requirement that `float` values be stored at memory locations that are word aligned. To be safe, the implementation always hands out 16-byte-aligned pointers (i.e., their address is a multiple of 16).

⟨*Round up* sz *to minimum machine alignment*⟩ ≡ 1016
```
sz = ((sz + 15) & (~15));
```

If a new block of memory is needed to service request, the `MemoryArena` stores the pointer to the current block of memory on the `usedBlocks` list so that it is not lost. Later, when `MemoryArena::FreeAll()` is called, it will be able to reuse the block for the next series of allocations. The allocation routine then checks to see if there are any already allocated free blocks in the `availableBlocks` list before calling the system allocation routine to allocate a brand-new block.

⟨*Get new block of memory for* MemoryArena⟩ ≡ 1016
```
usedBlocks.push_back(currentBlock);
if (availableBlocks.size() && sz <= blockSize) {
    currentBlock = availableBlocks.back();
    availableBlocks.pop_back();
}
else
    currentBlock = AllocAligned<char>(max(sz, blockSize));
```

When the user is done with all of the memory, the arena just resets its offset in the current block and moves all of the memory from the `usedBlocks` list onto the `availableBlocks` list.

AllocAligned() 1013
MemoryArena 1015
MemoryArena::availableBlocks 1015
MemoryArena::blockSize 1015
MemoryArena::curBlockPos 1015
MemoryArena::currentBlock 1015
MemoryArena::FreeAll() 1017
MemoryArena::usedBlocks 1015

⟨*MemoryArena Public Methods*⟩ +≡ 1015

```
void FreeAll() {
    curBlockPos = 0;
    while (usedBlocks.size()) {
        availableBlocks.push_back(usedBlocks.back());
        usedBlocks.pop_back();
    }
}
```

A.5.5 BLOCKED 2D ARRAYS

In C++, 2D arrays are arranged in memory so that entire rows of values are contiguous in memory, as shown in Figure A.2(a). This is not always an optimal layout, however; for such an array indexed by (u, v), nearby (u, v) array positions will often map to distant memory locations. For all but the smallest arrays, the adjacent values in the v direction will be on different cache lines; thus, if the cost of a cache miss is incurred to reference a value at a particular location (u, v), there is no chance that handling that miss will also load into memory the data for values $(u, v + 1)$, $(u, v - 1)$, and so on. Thus, spatially coherent array indices in (u, v) do not necessarily lead to the spatially coherent memory access patterns that modern memory caches depend on.

To address this problem, the `BlockedArray` template implements a generic 2D array of values, with the items ordered in memory using a *blocked* memory layout, as shown in Figure A.2(b). The array is subdivided into square blocks of a small fixed size that is a power of two, `BLOCK_SIZE`. Each block is laid out row by row, as if it were a separate 2D C++ array. This organization substantially improves the memory coherence of 2D array references in practice and requires only a small amount of additional computation to determine the memory address for a particular position (Lam, Rothberg, and Wolf 1991).

To ensure that the block size is a power of two, the caller specifies its logarithm (base 2), which is given by the template parameter `logBlockSize`.

⟨*Memory Declarations*⟩ +≡

```
template <typename T, int logBlockSize> class BlockedArray {
public:
    ⟨BlockedArray Public Methods 1018⟩
private:
    ⟨BlockedArray Private Data 1018⟩
};
```

`BlockedArray` 1017
`BlockedArray::RoundUp()` 1019
`MemoryArena::availableBlocks` 1015
`MemoryArena::curBlockPos` 1015
`MemoryArena::usedBlocks` 1015

The constructor allocates space for the array and optionally initializes its values from a pointer to a standard C++ array. Because the array size may not be an exact multiple of the block size, it may be necessary to round up the size in one or both directions to find the total amount of memory needed for the blocked array. The `BlockedArray::RoundUp()` method rounds both dimensions up to be a multiple of the block size.

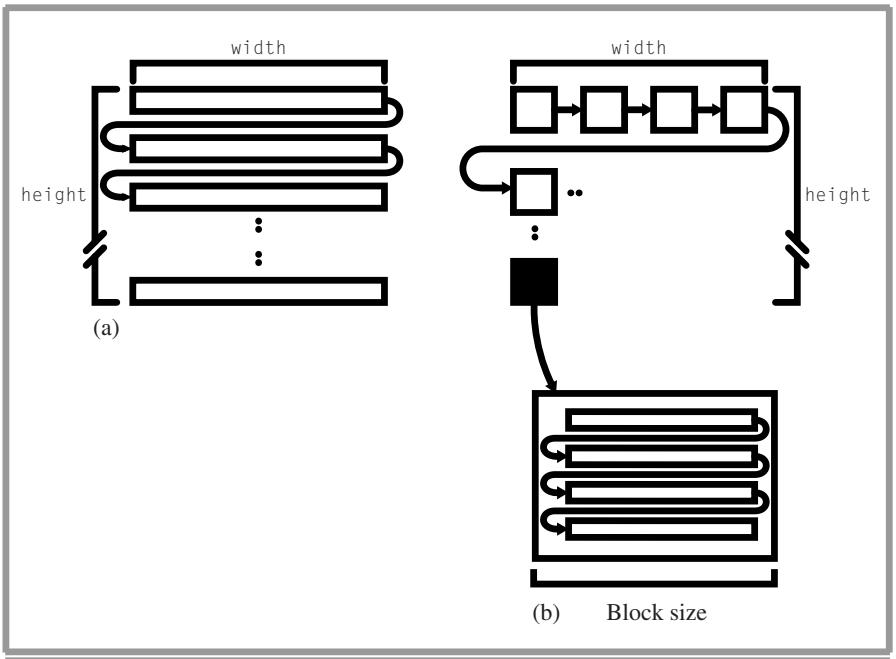

Figure A.2: (a) In C++, the natural layout for a 2D array of size `width*height` is a block of `width*height` entries, where the (u, v) array element is at the `u+v*width` offset. (b) A blocked array has been split into smaller square blocks, each of which is laid out linearly. Although it is slightly more complex to find the memory location associated with a given (u, v) array position in the blocked scheme, the improvement in cache performance due to more coherent memory access patterns often more than makes up for this in overall faster performance.

⟨*BlockedArray Public Methods*⟩ ≡ 1017

```
BlockedArray(uint32_t nu, uint32_t nv, const T *d = NULL) {
    uRes = nu;
    vRes = nv;
    uBlocks = RoundUp(uRes) >> logBlockSize;
    uint32_t nAlloc = RoundUp(uRes) * RoundUp(vRes);
    data = AllocAligned<T>(nAlloc);
    for (uint32_t i = 0; i < nAlloc; ++i)
        new (&data[i]) T();
    if (d)
        for (uint32_t v = 0; v < vRes; ++v)
            for (uint32_t u = 0; u < uRes; ++u)
                (*this)(u, v) = d[v * uRes + u];
}
```

AllocAligned() 1013
BlockedArray 1017
BlockedArray::RoundUp() 1019

⟨*BlockedArray Private Data*⟩ ≡ 1017

```
T *data;
uint32_t uRes, vRes, uBlocks;
```

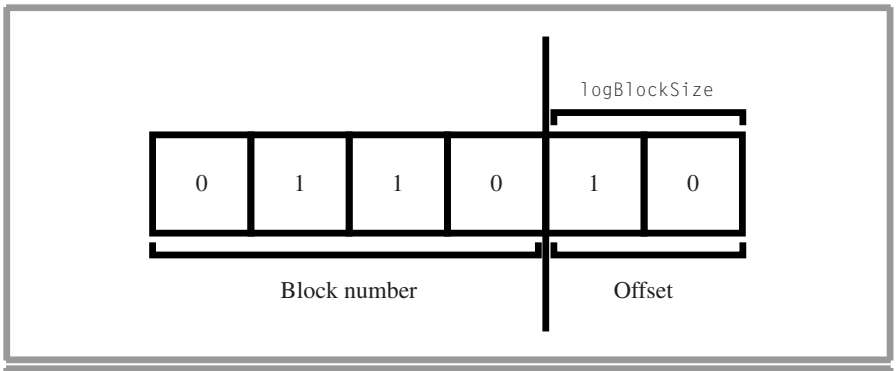

Figure A.3: Given an array coordinate, the (u, v) block number that it is in can be found by shifting off the `logBlockSize` low-order bits for both u and v. For example, with a `logBlockSize` of 2 and thus a block size of 4, we can see that this correctly maps 1D array positions from 0 to 3 to block 0, 4 to 7 to block 1, and so on. To find the offset within the particular block, it is just necessary to mask off the high-order bits, leaving the `logBlockSize` low-order bits. Because the block size is a power of two, these computations can all be done with efficient bit operations.

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
uint32_t BlockSize() const { return 1 << logBlockSize; }
uint32_t RoundUp(uint32_t x) const {
    return (x + BlockSize() - 1) & ~(BlockSize() - 1);
}
```

For convenience, the `BlockedArray` can also report its size in each dimension:

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
uint32_t uSize() const { return uRes; }
uint32_t vSize() const { return vRes; }
```

Looking up a value from a particular (u, v) position in the array requires some indexing work to find the memory location for that value. There are two steps to this process: finding which block the value is in and finding its offset within that block. Because the block sizes are always powers of two, the `logBlockSize` low-order bits in each of the u and v array positions give the offset within the block, and the high-order bits give the block number (Figure A.3).

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
uint32_t Block(uint32_t a) const { return a >> logBlockSize; }
uint32_t Offset(uint32_t a) const { return (a & (BlockSize() - 1)); }
```

BlockedArray::
BlockSize() 1019
BlockedArray::
logBlockSize 1017
BlockedArray::uRes 1018
BlockedArray::vRes 1018

Then, given the block number (b_u, b_v) and the offset within the block (o_u, o_v), it is necessary to compute what memory location this maps to in the blocked array layout. First consider the task of finding the starting address of the block; since the blocks are laid out row by row, this corresponds to the block number `bu + bv * uBlocks`, where `uBlocks` is the number of blocks in the u direction. Because each block has `BlockSize()*BlockSize()` values in it, the product of the block number and this value gives us the offset to the start

of the block. We then just need to account for the additional offset from the start of the block, which is `ou + ov * BlockSize()`.

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
T &operator()(uint32_t u, uint32_t v) {
    uint32_t bu = Block(u), bv = Block(v);
    uint32_t ou = Offset(u), ov = Offset(v);
    uint32_t offset = BlockSize() * BlockSize() * (uBlocks * bv + bu);
    offset += BlockSize() * ov + ou;
    return data[offset];
}
```

A.6 MATHEMATICAL ROUTINES

This section describes a number of useful mathematical functions and classes that support basic operations in `pbrt`, such as solving small linear systems, manipulating matrices, and linear interpolation.

A.6.1 2 × 2 LINEAR SYSTEMS

There are a number of places throughout `pbrt` where we need to solve a 2×2 linear system $Ax = B$ of the form

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$$

for values x_0 and x_1. The `SolveLinearSystem2x2()` routine finds the closed-form solution to such a system. It returns `true` if it was successful, and `false` if the determinant of A is very small, indicating that the system is numerically ill-conditioned and either not solvable or likely to have unacceptable floating-point errors. In this case, no solution is returned.

⟨*Matrix4x4 Method Definitions*⟩ ≡
```
bool SolveLinearSystem2x2(const float A[2][2],
        const float B[2], float *x0, float *x1) {
    float det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
    if (fabsf(det) < 1e-10f)
        return false;
    *x0 = (A[1][1]*B[0] - A[0][1]*B[1]) / det;
    *x1 = (A[0][0]*B[1] - A[1][0]*B[0]) / det;
    if (isnan(*x0) || isnan(*x1))
        return false;
    return true;
}
```

BlockedArray::Block() 1019
BlockedArray::BlockSize() 1019
BlockedArray::data 1018
BlockedArray::operator() 1020
BlockedArray::uBlocks 1018
Matrix4x4 1021
Transform 76

A.6.2 4 × 4 MATRICES

The `Matrix4x4` structure provides a low-level representation of 4×4 matrices. It is an integral part of the `Transform` class.

⟨*Matrix4x4 Declarations*⟩ ≡
```
struct Matrix4x4 {
    ⟨Matrix4x4 Public Methods 1021⟩
    float m[4][4];
};
```

The default constructor, not shown here, sets the matrix to the identity matrix. It also provides constructors that allow the user to pass an array of floats or 16 individual floats to initialize a `Matrix4x4`:

⟨*Matrix4x4 Public Methods*⟩ ≡ 1021
```
Matrix4x4(float mat[4][4]);
Matrix4x4(float t00, float t01, float t02, float t03,
          float t10, float t11, float t12, float t13,
          float t20, float t21, float t22, float t23,
          float t30, float t31, float t32, float t33);
```

The implementations of operators that test for equality and inequality are straightforward and not included in the text here.

The `Matrix4x4` class supports a few low-level matrix operations. For example, `Transpose()` returns a new matrix that is the transpose of the original matrix.

⟨*Matrix4x4 Method Definitions*⟩ +≡
```
Matrix4x4 Transpose(const Matrix4x4 &m) {
   return Matrix4x4(m.m[0][0], m.m[1][0], m.m[2][0], m.m[3][0],
                    m.m[0][1], m.m[1][1], m.m[2][1], m.m[3][1],
                    m.m[0][2], m.m[1][2], m.m[2][2], m.m[3][2],
                    m.m[0][3], m.m[1][3], m.m[2][3], m.m[3][3]);
}
```

The product of two matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ is computed by setting the (i, j)th element of the result to the inner product of the ith row of $\mathbf{M}_1$ with the jth column of $\mathbf{M}_2$.

⟨*Matrix4x4 Public Methods*⟩ +≡ 1021
```
static Matrix4x4 Mul(const Matrix4x4 &m1, const Matrix4x4 &m2) {
    Matrix4x4 r;
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            r.m[i][j] = m1.m[i][0] * m2.m[0][j] +
                        m1.m[i][1] * m2.m[1][j] +
                        m1.m[i][2] * m2.m[2][j] +
                        m1.m[i][3] * m2.m[3][j];
    return r;
}
```

Inverse() 1021
Matrix4x4 1021
Matrix4x4::m 1021
Transpose() 1021

Finally, `Inverse()` returns the inverse of the matrix. The implementation (not shown here) uses a numerically stable Gauss–Jordan elimination routine to compute the inverse.

⟨*Matrix4x4 Public Methods*⟩ +≡ 1021
```
friend Matrix4x4 Inverse(const Matrix4x4 &);
```

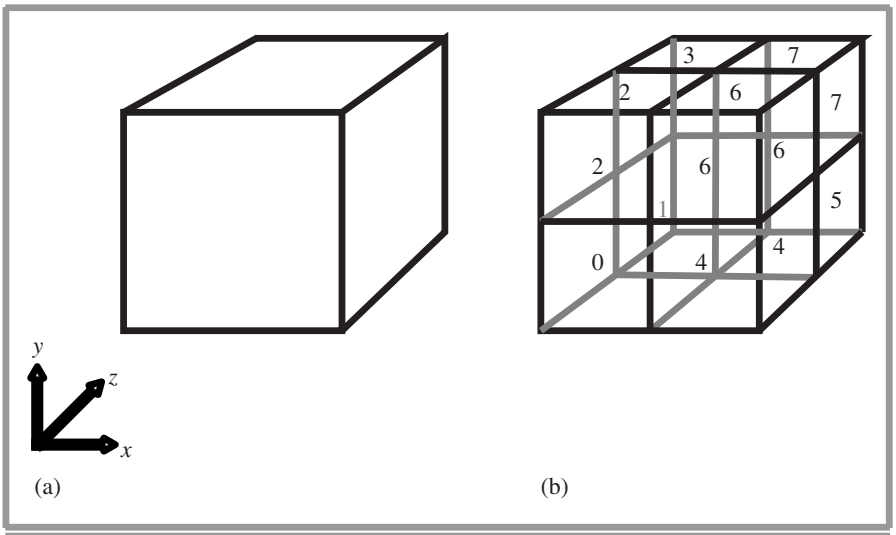

Figure A.4: Basic Octree Refinement. Starting with (a) an axis-aligned bounding box, the octree is defined by (b) progressively splitting each node into eight equal-sized child nodes. The order in which the child nodes are assigned numbers 0 to 7 is significant; details will be explained later in this section. Different subtrees may be refined to different depths, giving an adaptive discretization of 3D space.

A.7 OCTREES

An octree is a three-dimensional data structure that recursively splits a region of space into eight axis-aligned boxes (Figure A.4). The octree implementation defined in this section is in the file `core/octree.h`.

Octrees have many applications, including acceleration structures for ray tracing. The implementation in this section is specifically designed to determine which of a set of axis-aligned bounding boxes overlaps a given point. Using an octree for these queries can be substantially faster than looping over all of the objects directly. `pbrt` currently uses these octrees to store the irradiance estimates computed by the `IrradianceCacheIntegrator` and the `DipoleSubsurfaceIntegrator`. The values in these cases each have a bounding box associated with them that gives a spatial region the estimate represents.

First, we will define the `OctNode` structure, which represents a node of the tree. It holds pointers to the eight children of the node (some or all of which may be `NULL`) and a `vector` of `NodeData` objects. `NodeData` is a template argument that gives the type to be stored in the tree; for the `IrradianceCacheIntegrator`, it's the `IrradProcess` structure, which records a single irradiance estimate. The constructor and destructor of the `OctNode` just initialize the `children` to `NULL` and delete them, respectively.

DipoleSubsurfaceIntegrator 887
IrradianceCacheIntegrator 786
IrradProcess 792
NodeData 1023
OctNode 1023

⟨*Octree Declarations*⟩ ≡
```
template <typename NodeData> struct OctNode {
    OctNode() {
        for (int i = 0; i < 8; ++i)
            children[i] = NULL;
    }
    ~OctNode() {
        for (int i = 0; i < 8; ++i)
            delete children[i];
    }
    OctNode *children[8];
    vector<NodeData> data;
};
```

The `Octree` template is also parameterized by the `NodeData` type.

⟨*Octree Declarations*⟩ +≡
```
template <typename NodeData> class Octree {
public:
    ⟨Octree Public Methods 1023⟩
private:
    ⟨Octree Private Methods⟩
    ⟨Octree Private Data 1023⟩
};
```

The constructor takes the overall bounds of the tree and a maximum recursion depth beyond which nodes should not be refined:

⟨*Octree Public Methods*⟩ ≡ 1023
```
Octree(const BBox &b, int md = 16)
    : maxDepth(md), bound(b) { }
```

⟨*Octree Private Data*⟩ ≡ 1023
```
int maxDepth;
BBox bound;
OctNode<NodeData> root;
```

To add an item to the tree, the octree recurses through the tree, creating new nodes as needed, until termination criteria are met, at which point the item is added to the node it overlaps. Similar to the `KdTreeAccel` of Section 4.5, performance is highly dependent on the specific termination criteria. For example, we could decide never to refine the tree and add all items to the root node. This would be a valid octree, although it would perform poorly for large numbers of objects. However, if the tree is refined too much, items may span many nodes, leading to excessive memory use.

The `Octree::Add()` method for adding an item forwards the request to a private `Octree::addPrivate()` method with a few additional parameters, including the current node being considered, the bounding box of the node, and the squared length of the diagonal of the data item's bounding box. This method calls itself recursively as it works down the octree to the nodes where the item is stored.

BBox 70
KdTreeAccel 228
NodeData 1023
OctNode 1023
Octree 1023
Octree::Add() 1024
Octree::addPrivate() 1024
Octree::maxDepth 1023

⟨*Octree Public Methods*⟩ +≡ 1023
```
void Add(const NodeData &dataItem, const BBox &dataBound) {
    addPrivate(&root, bound, dataItem, dataBound,
               DistanceSquared(dataBound.pMin, dataBound.pMax));
}
```

The internal `Octree::addPrivate()` method either adds the item to the current node or determines which child nodes the item overlaps, allocates them if necessary, and recursively calls itself to allow the children to decide whether to add the item to their lists.

⟨*Octree Method Definitions*⟩ ≡
```
template <typename NodeData>
void Octree<NodeData>::addPrivate(
        OctNode<NodeData> *node, const BBox &nodeBound,
        const NodeData &dataItem, const BBox &dataBound,
        float diag2, int depth) {
    ⟨Possibly add data item to current octree node 1024⟩
    ⟨Otherwise add data item to octree children 1024⟩
}
```

The item is added to the current node once the maximum tree depth is reached or when the length of the diagonal of the node is less than the length of the diagonal of the item's bounds. This ensures that the item overlaps a relatively small number of tree nodes while not being too small relative to the extent of the nodes that it's added to. Figure A.5 shows the basic operation of the algorithm in two dimensions (where the corresponding data structure is known as a *quadtree*).

⟨*Possibly add data item to current octree node*⟩ ≡ 1024
```
if (depth == maxDepth ||
    DistanceSquared(nodeBound.pMin, nodeBound.pMax) < diag2) {
    node->data.push_back(dataItem);
    return;
}
```

If the method continues down the tree, it needs to determine which of the child nodes the item's bounding box overlaps. The fragment ⟨*Determine which children the item overlaps*⟩ initializes an array of Boolean values, `over[]`, such that the ith element is true only if the bounds of the data item being added overlap the ith child of the current node. The method can then loop over the eight children and recursively call `addPrivate()` for the ones that the object overlaps.

⟨*Otherwise add data item to octree children*⟩ ≡ 1024
```
Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;
⟨Determine which children the item overlaps 1026⟩
for (int child = 0; child < 8; ++child) {
    if (!over[child]) continue;
    ⟨Allocate octree node if needed and continue recursive traversal 1025⟩
}
```

BBox 70
BBox::pMax 71
BBox::pMin 71
DistanceSquared() 65
NodeData 1023
OctNode 1023
Octree 1023
Octree::addPrivate() 1024
Octree::bound 1023
Octree::maxDepth 1023
Octree::root 1023
Point 63

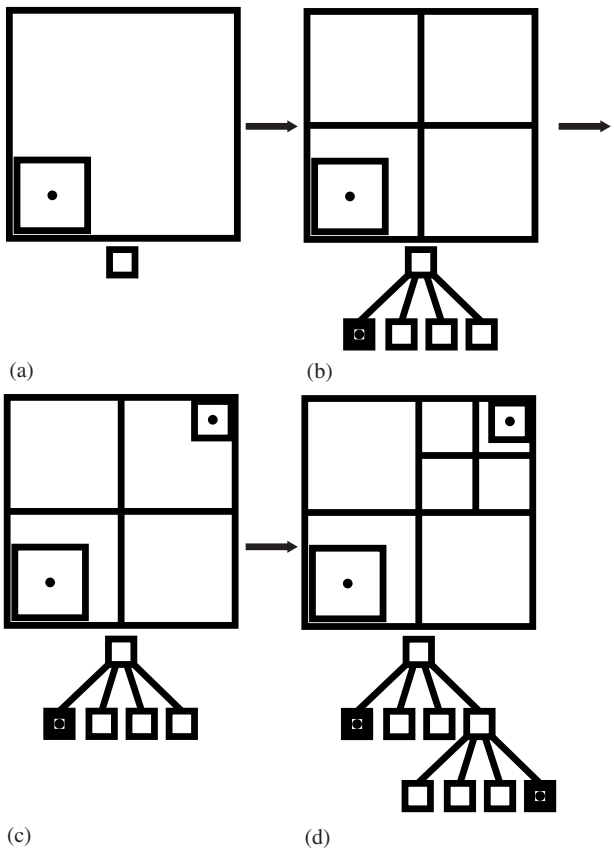

Figure A.5: Creation of a Quadtree (the 2D Analog of an Octree). (a) The tree contains just the root node, and an object with bounds around a given point is being added. The tree's topology is illustrated with a small box beneath it, corresponding to the root node with no children. (b) The tree is refined one level and the object is added to the single child node that it overlaps (again shown schematically underneath the tree). (c) Another new object with a smaller bounding box than the first is being added. (d) The tree is refined a second time before the item is added. In general, items may be stored in multiple nodes of the tree.

⟨*Allocate octree node if needed and continue recursive traversal*⟩ ≡ 1024

```
if (!node->children[child])
    node->children[child] = new OctNode<NodeData>;
BBox childBound = octreeChildBound(child, nodeBound, pMid);
addPrivate(node->children[child], childBound,
           dataItem, dataBound, diag2, depth+1);
```

BBox 70
NodeData 1023
OctNode 1023
OctNode::children 1023
Octree::addPrivate() 1024
octreeChildBound() 1026

Rather than computing the bounds of each child and doing a bounding box overlap test, it is possible to save work by taking advantage of symmetries of the situation. For example, if the x range of the object's bounding box is entirely on the left side of the plane that splits the tree node in the x direction, it cannot overlap any of the four child nodes

on the right side. Careful selection of the child node numbering scheme in Figure A.4 is the key to the success of this approach. The child nodes are numbered such that the low bit of a child's index is zero if its z component is on the low side of the z splitting plane and one if it is on the high side. Similarly, the second bit is set based on which side of the y plane the child is on, and the third bit is set based on its position with respect to the x plane. Given Boolean variables that classify a child node with respect to the splitting planes (`true` if it is above the plane), the child number of a given node is equal to

```
4 * (xHigh ? 1 : 0) + 2 * (yHigh ? 1 : 0) + 1 * (zHigh ? 1 : 0).
```

It is possible to quickly determine which child nodes a given bounding box overlaps by classifying its extent with respect to the center point of the node. For example, if the bounding box's starting x value is less than the midpoint, then the node potentially overlaps children numbers 0, 1, 2, and 3. If its ending x value is greater than the midpoint, it potentially overlaps 4, 5, 6, and 7. The following fragment checks the three dimensions in turn, computing the logical AND of the results; the item only overlaps a child node if it overlaps its extent in all three dimensions.

⟨*Determine which children the item overlaps*⟩ ≡ 1024

```
bool x[2] = { dataBound.pMin.x <= pMid.x, dataBound.pMax.x > pMid.x };
bool y[2] = { dataBound.pMin.y <= pMid.y, dataBound.pMax.y > pMid.y };
bool z[2] = { dataBound.pMin.z <= pMid.z, dataBound.pMax.z > pMid.z };
bool over[8] = { x[0] & y[0] & z[0], x[0] & y[0] & z[1],
                 x[0] & y[1] & z[0], x[0] & y[1] & z[1],
                 x[1] & y[0] & z[0], x[1] & y[0] & z[1],
                 x[1] & y[1] & z[0], x[1] & y[1] & z[1] };
```

The child node numbering scheme also makes it possible to easily find the bounding box of a particular child based on the child number and the parent node's bound:

⟨*Octree Declarations*⟩ +≡

```
inline BBox octreeChildBound(int child, const BBox &nodeBound,
                             const Point &pMid) {
    BBox childBound;
    childBound.pMin.x = (child & 4) ? pMid.x : nodeBound.pMin.x;
    childBound.pMax.x = (child & 4) ? nodeBound.pMax.x : pMid.x;
    childBound.pMin.y = (child & 2) ? pMid.y : nodeBound.pMin.y;
    childBound.pMax.y = (child & 2) ? nodeBound.pMax.y : pMid.y;
    childBound.pMin.z = (child & 1) ? pMid.z : nodeBound.pMin.z;
    childBound.pMax.z = (child & 1) ? nodeBound.pMax.z : pMid.z;
    return childBound;
}
```

After items have been added to the tree, the user can use the tree to find the items that have bounds that overlap a given point. The `Lookup()` method walks down the tree, processing the nodes that the given point overlaps. The user-supplied callback, `process`, is called for each `NodeData` item that overlaps the given point. As with the `Add()` method, the main lookup function directly calls a private version that takes a pointer to the current node and the current node's bounds.

BBox 70
NodeData 1023
Point 63

⟨*Octree Public Methods*⟩ +≡ 1023

```
template <typename LookupProc> void Lookup(const Point &p,
                                           LookupProc &process) {
    if (!bound.Inside(p)) return;
    lookupPrivate(&root, bound, p, process);
}
```

If the private lookup function has been called with a given node, the point `p` must be inside the node. The user-supplied callback is called for each `NodeData` item that is stored in the octree node, allowing the user to do whatever processing is appropriate.[3]

The callback must be either a pointer to a function that takes a position and a `NodeData` object or a class that has an `operator()` method that takes those arguments. If the callback returns `false`, traversal is halted. After the items are processed, this method continues down the tree into the single child node that `p` is inside until the bottom is reached.

⟨*Octree Method Definitions*⟩ +≡

```
template <typename NodeData> template <typename LookupProc>
bool Octree<NodeData>::lookupPrivate(OctNode<NodeData> *node,
        const BBox &nodeBound, const Point &p, LookupProc &process) {
    for (uint32_t i = 0; i < node->data.size(); ++i)
        if (!process(node->data[i]))
            return false;
    ⟨Determine which octree child node p is inside 1027⟩
    if (!node->children[child])
        return true;
    BBox childBound = octreeChildBound(child, nodeBound, pMid);
    return lookupPrivate(node->children[child], childBound, p, process);
}
```

Again taking advantage of the child numbering scheme, it is possible to quickly determine which child a point overlaps by classifying it with respect to the center of the parent node in each direction:

⟨*Determine which octree child node* p *is inside*⟩ ≡ 1027

```
Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
int child = (p.x > pMid.x ? 4 : 0) + (p.y > pMid.y ? 2 : 0) +
            (p.z > pMid.z ? 1 : 0);
```

BBox 70
BBox::Inside() 72
BBox::pMax 71
BBox::pMin 71
NodeData 1023
OctNode 1023
OctNode::children 1023
OctNode::data 1023
Octree 1023
Octree::bound 1023
Octree::lookupPrivate() 1027
octreeChildBound() 1026
Point 63

A.8 KD-TREES

Like the octree, the kd-tree is another data structure that accelerates the processing of spatial data. In contrast to the octree, where the data items had a known bounding box and the caller wanted to find all items that overlap a given point, the generic kd-tree

3 Note that the `Octree` actually passes all of the data items in the node to the callback, not just the subset of them that `p` is inside the bounds of. This isn't too much of a problem in practice, since the code using the `Octree` can always store a `BBox` in the `NodeData` and do the check itself. When it doesn't matter if a few extra `NodeData` items are passed back, writing the implementation in this way saves a substantial amount of `BBox` storage space.

presented in this section is useful for handling data items that are just single points in space with no associated bound, but where the caller wants to find all such points within a user-supplied distance of a given point. It is a key component of the implementation of the PhotonIntegrator.

The KdTree class described here is similar to the KdTreeAccel of Section 4.5 in that 3D space is progressively split in half by planes. There are two main differences here:

- Each tree node in the KdTree class stores a single data item. Therefore, there is exactly one kd-tree node for each data item stored in the tree.
- Because each item being stored is just a single point, items never straddle the splitting plane. Therefore, they never need to be stored on both sides of a split.

One result of these differences is that it is possible to build a perfectly balanced tree, which can improve the efficiency of lookups.

Like the KdTreeAccel, the implementation here stores all of the nodes of the tree in a single contiguous array. If a node has a left child, it will immediately follow the node in the array, and the rightChild member of KdNode gives the offset to the right child of the node, if any. rightChild will be set to a very large number if there is no right child.

To further improve the cache efficiency of the kd-tree, we will apply the cache optimization described previously of separating "hot" and "cold" data. Hot data is data that is frequently accessed while the tree is being traversed, while cold data is less frequently accessed. By splitting the kd-tree node data structure into two pieces in this way, it is possible to pack hot data close together in contiguous memory, which improves performance since more tree nodes can be packed into a single cache line. Hot data is stored in KdNode structures, which record information about a node's splitting plane and its children. The additional cold data that the user wants to associate with each node is stored in a separate array, indexed identically to the KdNode array.

⟨*KdTree Declarations*⟩ ≡

```
struct KdNode {
    void init(float p, uint32_t a) {
        splitPos = p;
        splitAxis = a;
        rightChild = (1<<29)-1;
        hasLeftChild = 0;
    }
    void initLeaf() {
        splitAxis = 3;
        rightChild = (1<<29)-1;
        hasLeftChild = 0;
    }
    ⟨KdNode Data 1029⟩
};
```

KdNode 1028
KdTree 1029
KdTreeAccel 228
PhotonIntegrator 802

⟨*KdNode Data*⟩ ≡ 1028
```
float splitPos;
uint32_t splitAxis:2;
uint32_t hasLeftChild:1, rightChild:29;
```

The KdTree is a template class that is parameterized by the type of object stored in the nodes (NodeData) and the type of callback function that is used for reporting which nodes are within a given search radius of the lookup position. The KdTree implementation requires that the NodeData class has a Point member variable called NodeData::p that gives its position.

⟨*KdTree Declarations*⟩ +≡
```
template <typename NodeData> class KdTree {
public:
    ⟨KdTree Public Methods⟩
private:
    ⟨KdTree Private Methods⟩
    ⟨KdTree Private Data 1029⟩
};
```

⟨*KdTree Private Data*⟩ ≡ 1029
```
KdNode *nodes;
NodeData *nodeData;
uint32_t nNodes, nextFreeNode;
```

Because incremental addition or removal of kd-tree nodes isn't needed in pbrt, the implementation of the KdTree is made more straightforward by having all of the data to be stored passed to the KdTree constructor. The constructor allocates all of the memory needed for the tree and the data and calls the recursive tree construction function.

⟨*KdTree Method Definitions*⟩ ≡
```
template <typename NodeData>
KdTree<NodeData>::KdTree(const vector<NodeData> &d) {
    nNodes = d.size();
    nextFreeNode = 1;
    nodes = AllocAligned<KdNode>(nNodes);
    nodeData = AllocAligned<NodeData>(nNodes);
    vector<const NodeData *> buildNodes(nNodes, NULL);
    for (uint32_t i = 0; i < nNodes; ++i)
        buildNodes[i] = &d[i];
    ⟨Begin the KdTree building process 1029⟩
}
```

AllocAligned() 1013
KdNode 1028
KdTree 1029
KdTree::nextFreeNode 1029
KdTree::nNodes 1029
KdTree::nodeData 1029
KdTree::recursiveBuild() 1030
NodeData 1023
Point 63

Tree construction is handled by the recursiveBuild() method. It takes the node number of the current node to be initialized and offsets into the array data, indicating the subset of data items [start, end) from the buildNodes array to be stored beneath this node.

⟨*Begin the KdTree building process*⟩ ≡ 1029
```
recursiveBuild(0, 0, nNodes, &buildNodes[0]);
```

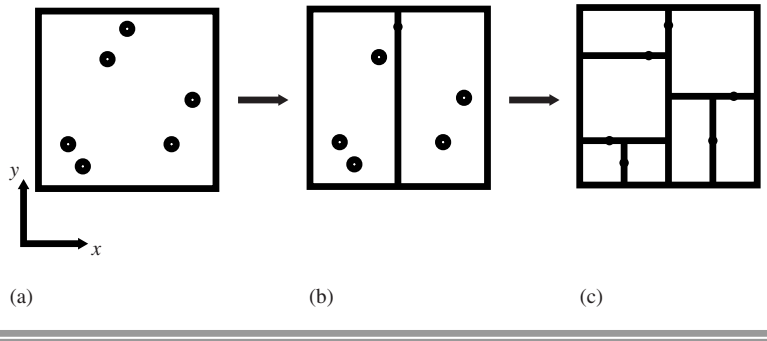

Figure A.6: Creation of a Kd-Tree to Store a Set of Points. (a) A collection of points. (b) A split direction and position are chosen. Here, we have decided to split in the x direction. We choose the point in the middle of the list sorted by x and split along the plane that goes through the point. Roughly half of points are to the left of the splitting plane and half are to the right. (c) We then continue recursively in each half, allocating new tree nodes, splitting and partitioning, until all data points have been processed.

The tree building process selects the "middle" element of the user-supplied data (what this means will soon be defined precisely) and partitions the data so that all items below the middle are in the first half of the array and all items above the middle are in the second half. It constructs a node with the middle element as its data item and then recursively initializes the two children of the node by processing the first and second halves of the array (Figure A.6).

⟨*KdTree Method Definitions*⟩ +≡
```
template <typename NodeData> void
KdTree<NodeData>::recursiveBuild(uint32_t nodeNum, int start, int end,
        const NodeData **buildNodes) {
    ⟨Create leaf node of kd-tree if we've reached the bottom 1030⟩
    ⟨Choose split direction and partition data 1031⟩
    ⟨Allocate kd-tree node and continue recursively 1031⟩
}
```

When there is just a single item to be processed, the bottom of the tree has been reached, so the node is flagged as a leaf and the `nodeData` array item at the appropriate offset is initialized from the appropriate user data.

⟨*Create leaf node of kd-tree if we've reached the bottom*⟩ ≡ 1030
```
if (start + 1 == end) {
    nodes[nodeNum].initLeaf();
    nodeData[nodeNum] = *buildNodes[start];
    return;
}
```

KdNode::initLeaf() 1028
KdTree 1029
KdTree::nodeData 1029
NodeData 1023

Otherwise, the data is partitioned into two halves, and a nonleaf node is initialized. The longest edge of the remaining data's bounding box is used to choose which axis to split along. The standard library `nth_element()` function then finds the middle node along that axis. It takes three pointers into a sequence, `start`, `mid`, and `end`, and partitions the

sequence such that the `mid`th element is in the position it would be in if the sequence were sorted. It also rearranges the array so that all elements from `start` to `mid-1` are less than `mid`, and all elements from `mid+1` to `end` are greater than `mid`. This can all be done more quickly than sorting the entire range—in $O(n)$ time rather than $O(n \log n)$.

⟨*Choose split direction and partition data*⟩ ≡ 1030
```
⟨Compute bounds of data from start to end 1031⟩
int splitAxis = bound.MaximumExtent();
int splitPos = (start+end)/2;
std::nth_element(&buildNodes[start], &buildNodes[splitPos],
                 &buildNodes[end], CompareNode<NodeData>(splitAxis));
```

It is easy to find the bounds of the entire array range that we're interested in using the `Union()` method:

⟨*Compute bounds of data from* `start` *to* `end`⟩ ≡ 1031
```
BBox bound;
for (int i = start; i < end; ++i)
    bound = Union(bound, buildNodes[i]->p);
```

The `nth_element()` function needs a "comparison object" that determines the ordering between two data elements. `CompareNode` compares positions along the chosen axis.

⟨*KdTree Declarations*⟩ +≡
```
template <typename NodeData> struct CompareNode {
    CompareNode(int a) { axis = a; }
    int axis;
    bool operator()(const NodeData *d1, const NodeData *d2) const {
        return d1->p[axis] == d2->p[axis] ? (d1 < d2) :
                                            d1->p[axis] < d2->p[axis];
    }
};
```

Once the data have been partitioned, the current node is initialized to store the middle item, and its two children are recursively initialized with the two sets of remaining items:

⟨*Allocate kd-tree node and continue recursively*⟩ ≡ 1030
```
nodes[nodeNum].init(buildNodes[splitPos]->p[splitAxis], splitAxis);
nodeData[nodeNum] = *buildNodes[splitPos];
if (start < splitPos) {
    nodes[nodeNum].hasLeftChild = 1;
    uint32_t childNum = nextFreeNode++;
    recursiveBuild(childNum, start, splitPos, buildNodes);
}
if (splitPos+1 < end) {
    nodes[nodeNum].rightChild = nextFreeNode++;
    recursiveBuild(nodes[nodeNum].rightChild, splitPos+1,
                   end, buildNodes);
}
```

When another part of the system wants to look up items from the tree, it provides a point `p`, a callback procedure (similar to the one used above), and a maximum squared

BBox 70
BBox::MaximumExtent() 73
BBox::Union() 71
CompareNode 1031
KdNode::hasLeftChild 1029
KdNode::rightChild 1029
KdNode::splitAxis 1029
KdNode::splitPos 1029
KdTree::
recursiveBuild() 1030
NodeData 1023
Union() 72

search radius. Using the squared radius rather than the radius directly leads to some optimizations in the following traversal code. All data items within that radius will be passed back to the caller.

Rather than being passed by value, the squared search radius is passed into the lookup function by reference. This allows the lookup routine to pass it to the callback procedure by reference, so that the callback can reduce the search radius as the search goes on. This can speed up lookups when the callback routine can determine that a smaller search radius was appropriate after all. As usual, the lookup method immediately calls a private lookup procedure, passing in a pointer to the current node to be processed.

⟨*KdTree Method Definitions*⟩ +≡
```
template <typename NodeData> template <typename LookupProc>
void KdTree<NodeData>::Lookup(const Point &p, LookupProc &proc,
                              float &maxDistSquared) const {
    privateLookup(0, p, proc, maxDistSquared);
}
```

The lookup function has two responsibilities (Figure A.7): it needs to recursively process each of the children of the current node if they overlap the search region, and it needs to invoke the callback routine, passing it the data item in the current node if it is inside the search radius.

⟨*KdTree Method Definitions*⟩ +≡
```
template <typename NodeData> template <typename LookupProc>
void KdTree<NodeData>::privateLookup(uint32_t nodeNum, const Point &p,
        LookupProc &process, float &maxDistSquared) const {
    KdNode *node = &nodes[nodeNum];
    ⟨Process kd-tree node's children 1033⟩
    ⟨Hand kd-tree node to processing function 1033⟩
}
```

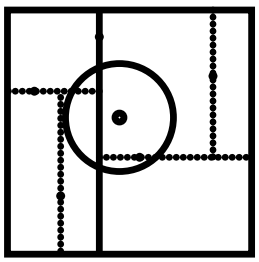

Figure A.7: Basic Process of Kd-Tree Lookups. The point marked with an open circle is the lookup position, and the region of interest is denoted by the circular region around it. At the root node of the tree (indicated by a bold splitting line), the data item is outside the region of interest, so it is not handed to the callback function. However, the region overlaps both children of the node, so we have to recursively consider each of them. In this case, the implementation here considers the right child (child number one) first in order to examine the nearby data items before examining the ones farther away.

KdNode 1028
KdTree 1029
KdTree::nodes 1029
KdTree::privateLookup() 1032
NodeData 1023
Point 63

The tree is traversed depth first, examining the leaf nodes that are close to the lookup point p first. This approach ensures that data points are passed to the callback function in a generally near-to-far order. If the caller is only interested in finding a fixed number of points around the lookup point, after which it will end the search, this is a more efficient order than going far to near.

Therefore, this method first recurses down the side of the tree on which the current point lies. After that lookup has returned, it checks to see if the search radius indicates that the region covers both sides of the tree. Leaf nodes are denoted by a value of 3 in the node's splitAxis field, in which case these steps are skipped.

⟨*Process kd-tree node's children*⟩ ≡ 1032

```
int axis = node->splitAxis;
if (axis != 3) {
    float dist2 = (p[axis] - node->splitPos) * (p[axis] - node->splitPos);
    if (p[axis] <= node->splitPos) {
        if (node->hasLeftChild)
            privateLookup(nodeNum+1, p, process, maxDistSquared);
        if (dist2 < maxDistSquared && node->rightChild < nNodes)
            privateLookup(node->rightChild, p, process, maxDistSquared);
    }
    else {
        if (node->rightChild < nNodes)
            privateLookup(node->rightChild, p, process, maxDistSquared);
        if (dist2 < maxDistSquared && node->hasLeftChild)
            privateLookup(nodeNum+1, p, process, maxDistSquared);
    }
}
```

Finally, at the end of the lookup function, a check is made to see if the point stored in the node is inside the search radius, passing it to the callback function if so. In addition to doing whatever processing it needs to do based on the item, the callback function may decrease maxDistSquared in order to reduce the region of space searched for the remainder of the processing.

⟨*Hand kd-tree node to processing function*⟩ ≡ 1032

```
float dist2 = DistanceSquared(nodeData[nodeNum].p, p);
if (dist2 < maxDistSquared)
    process(p, nodeData[nodeNum], dist2, maxDistSquared);
```

A.9 PARALLELISM

Section 1.3.5 introduced some basic principles of parallel programming and described their application to pbrt. Here, we'll provide some additional information on performance issues related to multithreading as well as the implementations of utility routines and classes for atomic memory operations, synchronization, and pbrt's task system. The implementations of these routines are in the files core/parallel.h and core/parallel.cpp.

DistanceSquared() 65
KdTree::nodeData 1029

Most of this code is highly platform specific; the implementations of the atomic operations require assembly language. The forthcoming update to the C++ standard will provide thread, mutex, and atomic operations as part of the standard library, which will eliminate the need for custom implementations here.[4]

A.9.1 MEMORY COHERENCE MODELS AND PERFORMANCE

Cache coherency is a feature of all modern multicore CPUs; with it, memory writes by one processor are automatically visible to other processors. This is an incredibly useful feature; being able to assume it in the implementation of a system like `pbrt` is extremely helpful to the programmer. Understanding the subtleties and the performance characteristics of this feature is important to keep in mind, however.

One important issue is that other processors may not see writes to memory in the same order that the processor that performed the writes issued them. This can happen for two main reasons: the compiler's optimizer may have reordered write operations to improve performance, and the CPU hardware may write values to memory in a different order than the stream of executed machine instructions. In the single-threaded case, both of these are innocuous; by design, the compiler and hardware, respectively, ensure that it's impossible for a single thread of execution to detect when these cases happen. This guarantee is not provided for multithreaded code, however; doing so would impose a significant performance penalty, so hardware architectures leave handling this problem, when it matters, to software.

Memory barrier instructions can be used to ensure that all write instructions before the barrier are visible in memory before any subsequent instructions execute. In practice, we generally don't need to issue memory barrier instructions explicitly, since operating system thread synchronization calls take care of this; they are defined to do this so that if we are coordinating execution between multiple threads using these calls, then they have a consistent view of memory after synchronization points.

Although cache coherence is helpful to the programmer, it can sometimes impose a substantial performance penalty for data that is frequently modified and accessed by multiple processors. Read-only data has little penalty; copies of it can be stored in the local caches of all of the processors that are accessing it, allowing all of them the same performance benefits from the caches as in the single-threaded case. To understand the downside of taking too much advantage of cache coherence for read–write data, it's useful to understand how cache coherence is typically implemented on processors.

CPUs implement a *cache coherence protocol*, which is responsible for tracking the memory transactions issued by all of the processors in order to provide cache coherence. A widely used protocol is *MESI*, where the acronym represents the four states that each cache line can be in. Each processor stores the current state for each cache line in its local caches:

4 The Boost libraries (*www.boost.org*) provide implementations of many of these components of the new C++ standard library. We have chosen not to make `pbrt` dependent on Boost, however, in order to make it easier to build, by not requiring the user to install a number of additional libraries.

- *Modified*—The current processor has written to the memory location, but the result is only stored in the cache—it's *dirty* and hasn't been written to main memory. No other processor has the location in its cache.
- *Exclusive*—The current processor is the only one with the data from the corresponding memory location in its cache. The value in the cache matches the value in memory.
- *Shared*—Multiple processors have the corresponding memory location in their caches, but they have only performed read operations.
- *Invalid*—The cache line doesn't hold valid data.

At system startup time, the caches are empty and all cache lines are in the invalid state. If multiple threads read a memory location, the data for that location may be replicated in multiple caches, in the "exclusive" or "shared" state. If another processor performs a memory read from a location that is in the "exclusive" state in another cache, then both caches record the state for the corresponding memory location to instead be "shared."

When a processor writes to a memory location, the performance impact depends on the state of the corresponding cache line. If it's in the "exclusive" state and already in the writing processor's cache, then the write is cheap; the data is modified in the cache and the cache line's state is changed to "modified." (If it was already in the "modified" state, then the write is similarly efficient.) In these cases, the value will eventually be written to main memory, at which point the corresponding cache line returns to the "exclusive" state.

However, if a processor writes to a memory location that's in the "shared" state in its cache or is in the "modified" or "exclusive" state in another processor's cache, then expensive communication between the cores is required. All of this is handled transparently by the hardware, though it still has a performance impact. In this case, the writing processor must issue a *read for ownership* (RFO), which marks the memory location as invalid in the caches of any other processors; RFOs can cause stalls of tens or hundreds of cycles—a substantial penalty for a single memory write.

In general, we'd therefore like to avoid the situation of multiple processors concurrently writing to the same memory location as well as unnecessarily reading memory that another processor is writing to. An important case to be aware of is "false sharing," where a single cache line holds some read-only data and some data that is frequently modified. In this case, even if only a single processor is writing to the part of the cache line that is modified but many are reading from the read-only part, the overhead of frequent RFO operations will be unnecessarily incurred.

Another situation where many processors may be writing to the same memory concurrently can happen when precomputing data for rendering. For example, during the photon shooting phase in Section 15.6.3, multiple threads are concurrently tracing paths and depositing photons in the scene. Rather than simultaneously updating a single shared data structure, each thread stores photons in its own private data structure and only periodically merges its results with the global photon data structure that will actually be used for rendering. This approach allows the overhead of RFO operations to be amortized over a smaller number of larger updates.

A.9.2 ATOMIC OPERATIONS

Recall from Section 1.3.5 that mutexes can be used to safely update memory locations that are shared by multiple threads. However, when it's possible to use built-in hardware instructions for atomic updates, doing so is often more efficient than acquiring a mutex, updating the location, and releasing the mutex. Hardware atomic update instructions operate only on 4 to 8 bytes of memory at a time, however, and support only a few operations (addition, exchange, etc.). If atomic updates to more data are required, mutexes must generally be used instead.

`pbrt` defines special types for 32-bit and, if available, 64-bit values to be updated atomically. Declaring these types with the `volatile` qualifier directs the compiler to always fetch their values from memory and to not cache them in registers. This ensures that updates made by one processor are visible to others.

⟨*Parallel Declarations*⟩ ≡
```
typedef volatile int32_t AtomicInt32;
#ifdef PBRT_HAS_64_BIT_ATOMICS
typedef volatile int64_t AtomicInt64;
#endif
```

The implementation of the atomic functions are highly platform specific; the 32-bit atomic add is below. For some hardware architectures and operating systems, a library call is available to perform the atomic operation. For others, we must use assembly language.

⟨*Parallel Declarations*⟩ +≡
```
inline int32_t AtomicAdd(AtomicInt32 *v, int32_t delta) {
#ifdef WIN32
    ⟨Do atomic add with MSVC inline assembly⟩
#elif defined(__APPLE__) && !(defined(__i386__) || defined(__amd64__))
    return OSAtomicAdd32Barrier(delta, v);
#else
    ⟨Do atomic add with gcc x86 inline assembly 1037⟩
#endif
}
```

In addition to different CPU architectures having different instruction sets, we need to deal with the fact that different compilers have different syntax for specifying assembly language within C and C++ code. The fragment below uses the `xadd` x86 opcode with the `lock` modifier. (The `l` at the end of `xadd` here indicates that the value is long-word sized, which for x86 means 32 bits.)

The `lock` prefix ensures that the instruction executes as an atomic operation and `xadd` adds the parameter (here, `delta`) to the destination (the address pointed to by `v`) and returns the old value pointed to by `v`. Here, we add the value `delta` to the value returned by `xadd` and return this value to the caller, thus returning the new value after the atomic add. Note that it would be an error to instead have the return statement be `return *v;` this would introduce a subtle race condition, since `*v` would need to be reloaded from memory and could have a different value if another processor had executed an atomic add between the add and the load here.

AtomicInt32 1036

⟨*Do atomic add with gcc x86 inline assembly*⟩ ≡ 1036

```
int32_t origValue;
__asm__ __volatile__("lock\n"
                     "xaddl %0,%1"
                     : "=r"(origValue), "=m"(*v) : "0"(delta)
                     : "memory");
return origValue + delta;
```

We won't include other assembly language implementations of atomic primitives in the book text here.

Another useful atomic operation is "compare and swap." It takes a memory location and the value that the caller believes that the location stores (`oldValue`). If the memory location still holds that value when the atomic compare and swap executes, then the value `newValue` is stored; otherwise, memory is left unchanged. In either case, the current value stored at the location is returned.

⟨*Parallel Declarations*⟩ +≡

```
inline int32_t AtomicCompareAndSwap(AtomicInt32 *v, int32_t newValue,
                                    int32_t oldValue);
```

Compare and swap is a building block that can be used to build many other atomic operations. For example, the code below could be executed by multiple threads to compute the maximum of values computed by the threads:

```
int32_t oldval;
do {
    oldval = *ptr;
    newval = max(oldval, newval);
} while(AtomicCompareAndSwap(ptr, newval, oldval) != oldval);
```

If only a single thread is trying to update the memory location, the loop is successful the first time through; the value loaded into `oldval` is still the value at `*ptr` when `AtomicCompareAndSwap()` executes and so `newval` is successfully stored and `oldval` is returned. If multiple threads are executing concurrently, then another thread may update the value at `*ptr` between the thread's read into `oldval` and execution of `AtomicCompareAndSwap()`. In that case, the compare and swap fails, memory isn't updated, and another pass is taken through the loop to try again.

Another application of atomic compare and swap is lazy construction of data structures. Consider a tree data structure where each node has child node pointers, initially set to `NULL`. If code traversing the tree wants to create a new child at a node, code could be written like:

```
if (node->firstChild == NULL) {
    Type *newChild = new Type ...
    if (AtomicCompareAndSwap(&node->firstChild, newChild, NULL) != NULL)
        delete newChild;
}
/* node->firstChild != NULL now */
```

AtomicInt32 1036

The idea is that if the child is NULL, the thread speculatively creates and fully initializes the child node. Atomic compare and swap is then used to try to initialize the child pointer in the tree; if it's still NULL, then the new child is stored and made available to all threads. If the child pointer is no longer NULL, then another thread has initialized the child in the time between the current thread first seeing that it was NULL and the current thread trying to update it. In this case, the work done in the current thread turns out to have been wasted, but it can delete the locally created child node and continue execution, using the node created by the other thread.

This is a simple example of a *lock-free* algorithm. This approach has a few advantages compared to, for example, using a reader–writer mutex to manage updating the tree. First, there's no overhead of acquiring the reader mutex for regular tree traversal; doing so may in general require an expensive system call. Second, multiple threads can naturally concurrently update different parts of the tree. With a single reader–writer mutex, if one thread acquires the mutex to update one node in the tree, other threads won't be able to update other nodes. The "Further Reading" section at the end of the appendix has more pointers to lock-free algorithms.

Atomic add of floating-point values isn't directly available in most instruction sets, though it can be constructed with atomic compare and swap.

⟨*Parallel Declarations*⟩ +≡
```
inline float AtomicAdd(volatile float *val, float delta) {
    union bits { float f; int32_t i; };
    bits oldVal, newVal;
    do {
        oldVal.f = *val;
        newVal.f = oldVal.f + delta;
    } while (AtomicCompareAndSwap(((AtomicInt32 *)val),
                                  newVal.i, oldVal.i) != oldVal.i);
    return newVal.f;
}
```

A.9.3 MUTEXES

Instances of the Mutex class can't be allocated on the stack or directly as members of classes or structures but must be dynamically allocated. This allows us to ensure that they are allocated to start at the beginning of cache lines and that other objects are not allocated in their cache line, eliminating the risk of false sharing. The Create() and Destroy() methods here handle these operations.

⟨*Parallel Declarations*⟩ +≡
```
class Mutex {
public:
    static Mutex *Create();
    static void Destroy(Mutex *m);
private:
    ⟨Mutex Private Methods⟩
    ⟨System-dependent mutex implementation⟩
};
```

AtomicInt32 1036
Mutex 1038

The MutexLock structure manages acquisition and release of mutex locks; it waits in its constructor until the mutex is available, and when it goes out of scope and its destructor executes, the lock is released. Using a helper structure in this manner helps ensure that locks are released when no longer needed. If a mutex only needs to be held during the execution of some function, but the function has multiple return statements, this approach is more convenient than having to release the mutex before each return.

⟨*Parallel Declarations*⟩ +≡
```
struct MutexLock {
    MutexLock(Mutex &m);
    ~MutexLock();
private:
    Mutex &mutex;
};
```

RWMutex provides reader–writer mutexes, which allow multiple threads to acquire a lock for reading but only one thread at a time to acquire a lock for writing (and only when no other thread holds a reader lock).

⟨*Parallel Declarations*⟩ +≡
```
class RWMutex {
public:
    static RWMutex *Create();
    static void Destroy(RWMutex *m);
private:
    ⟨RWMutex Private Methods⟩
    ⟨System-dependent rw mutex implementation⟩
};
```

Similar to MutexLock, the RWMutexLock helper structure acquires and releases reader–writer mutexes. A parameter to the constructor indicates whether a read-only or a write lock is wanted. Furthermore, after a lock of one type has been acquired, methods allow requests to change to the other type. (These methods may of course need to block until other threads release their locks.)

⟨*Parallel Declarations*⟩ +≡
```
enum RWMutexLockType { READ, WRITE };
```

⟨*Parallel Declarations*⟩ +≡
```
struct RWMutexLock {
    RWMutexLock(RWMutex &m, RWMutexLockType t);
    ~RWMutexLock();
    void UpgradeToWrite();
    void DowngradeToRead();
private:
    RWMutexLockType type;
    RWMutex &mutex;
};
```

Mutex 1038
MutexLock 1039
RWMutex 1039
RWMutexLock 1039
RWMutexLockType 1039

A.9.4 TASK SYSTEM

`pbrt` is parallelized with a *task system* that executes independent *tasks* on the processing cores of the system. When a part of the system has work that can be done in parallel, it decomposes the work into some number of tasks and provides them to the task system for scheduling and execution. This style of parallelism has become widely and successfully used in game engines, for example. The 10.6 version of Mac OS® X provides a task system, *Grand Central Dispatch*, as a fundamental part of the system libraries, and the 2010 release of Microsoft® Visual Studio® provides a task system as part of the *Concurrency Runtime*. (The task system in this section is built on top of these system-level task systems if available.)

A strict definition of a task is that it is a computational job that is immediately ready to execute, independent of all other tasks, and able to run to completion. (In other words, a task won't go partway through its work and then ask the task scheduler to suspend it and resume it later, pending availability of some other resource.) This expectation implies that tasks aren't expected to hold mutexes for long periods of time. Given these constraints, the task system's job is relatively simple: it launches one hardware execution thread on each processor in the system where each thread runs code that removes a task from a task queue, executes it, and repeats, until no more tasks remain. Tasks can be launched in any order. Because persistent threads run on the processors (versus launching a new thread for each unit of work), tasks can be fine-grained—they don't need to perform large amounts of computation to amortize the operating system's thread launching overhead.

This style of expressing parallelism in software systems often leads to clean code implementations and good parallel scalability. The requirements (like independence from other tasks and ability to run to completion) lead to high performance when the tasks execute, and the code that implements the work that tasks do tends to have few parallel synchronization calls in it; most of this is isolated in the task system itself. The programmer's main task is to decompose the computation into as many fine-grained tasks as possible and then not worry about the details of how they run. Furthermore, because the programmer is encouraged to create many fine-grained tasks (rather than decompose the work into a number of units equal to the number of processing cores present), the system can load-balance well when tasks have variable amounts of computation and scale well as more cores are available in the future.

`TasksInit()` is called at system startup time. It launches a task worker thread for each of the cores in the system (or, it launches the number of threads manually specified by the user), where each thread calls a function that waits for work on the task queue and takes and runs jobs as they become available. Thus, we have the main thread of execution continuing to run and independent of the task execution threads. In `pbrt`, usually either the task worker threads will be active running tasks and the main thread will be waiting for the tasks to all finish before continuing, or the main thread will be running while the worker threads are waiting for new tasks to be created. In other task-based software systems, it's often the case that the main thread and some tasks execute concurrently. This simpler approach makes `pbrt`'s parallelization easier to understand, at a small cost to performance.

⟨*Parallel Declarations*⟩ +≡
```
void TasksInit();
```

`TasksCleanup()` cleans up the resources allocated for the threads. It is an error (detected by an `Assert()` test) to call this function while tasks are still running.

⟨*Parallel Declarations*⟩ +≡
```
void TasksCleanup();
```

Tasks are implemented by inheriting from the abstract `Task` base-class. Implementations of this class gather the data needed to execute the task and implement the `Run()` method. See the `SamplerRendererTask` implementation in Section 1.3.4 as an example.

⟨*Parallel Declarations*⟩ +≡
```
class Task {
public:
    virtual ~Task();
    virtual void Run() = 0;
};
```

The `EnqueueTasks()` function adds the given tasks to the work queue and returns immediately. In `pbrt`, the main thread generally will want to wait for all of the tasks to finish before continuing execution; the `WaitForAllTasks()` function does just this. Alternatively, if the results from the tasks weren't needed immediately, the main thread could continue execution and call `WaitForAllTasks()` later.

The implementation of `pbrt`'s task system is relatively simple: there's a single task queue, protected by a mutex, shared by all task worker threads. More sophisticated task system implementations have multiple task queues, each with different priorities, and per-thread local task queues, allowing worker threads to quickly retrieve the next task to run without contention at a centralized task queue. In this approach, when a thread's local task queue is empty, it steals work from another thread's queue.

More sophisticated task systems also support finer-grained control to wait for task completion, making it possible to mark a set of tasks as a group and wait for their completion only, while other tasks continue to run. `pbrt` doesn't currently need this type of functionality, so it's not currently available in the task system.

⟨*Parallel Declarations*⟩ +≡
```
void EnqueueTasks(const vector<Task *> &tasks);
void WaitForAllTasks();
```

The last function in this section is `NumSystemCores()`, which returns the number of processing cores in the system. Its implementation is highly operating system dependent.

SamplerRendererTask 29
Task 1041
WaitForAllTasks() 1041

⟨*Parallel Declarations*⟩ +≡
```
int NumSystemCores();
```

FURTHER READING

Hacker's Delight (Warren 2006), is a delightful and thought-provoking exploration of the bit-twiddling algorithms like those used in some of the utility routines in this appendix. Sean Anderson has a Web page filled with a collection of bit-twiddling techniques like the ones in the `IsPowerOf2()` and `RoundUpPow2()` functions at *graphics.stanford.edu/~seander/bithacks.html*. See Kotay's Web page at *www.stereopsis.com/sree/fpu2006.html* for an extensive discussion of efficient floating-point to integer conversions.

Numerical Recipes, by Press et al. (1992), and Atkinson's book (1993) on numerical analysis both discuss algorithms for matrix inversion and solving linear systems.

Detailed information about the random number generator used in `pbrt`, including the original paper that describes its derivation, is available at *www.math.keio.ac.jp/~matumoto/emt.html*.

Many papers have been written on cache-friendly programming techniques; only a few are surveyed here. Ericson's chapter on high-performance programming techniques has very good coverage of this topic (Ericson 2004). Lam, Rothberg, and Wolf (1991) investigated blocking (tiling) for improving cache performance and developed techniques for selecting appropriate block sizes, given the size of the arrays and the cache size. Grunwald, Zorn, and Henderson (1993) were one of the first groups of researchers to investigate the interplay between memory allocation algorithms and the cache behavior of applications.

In `pbrt`, we only worry about cache layout issues for dynamically allocated data. However, Calder et al. (1998) show a profile-driven system that optimizes memory layout of global variables, constant values, data on the stack, and dynamically allocated data from the heap in order to reduce cache conflicts among them all, giving an average 30% reduction in data cache misses for the applications they studied.

Blocking for tree data structures was investigated by Chilimbi et al. (1999b); they ensured that nodes of the tree and a few levels of its children were allocated contiguously. Among other applications, they applied their tool to the layout of the acceleration octree in the *Radiance* renderer and reported a 42% speedup in run time. Chilimbi et al. (1999a) also evaluated the effectiveness of reordering fields inside structures to improve locality.

Drepper's paper (2007) is a useful resource for understanding performance issues related to caches, cache coherence, and main memory access, particularly in multicore systems.

Samet's book on octrees is the canonical reference on that data structure (Samet 1990), and de Berg et al.'s book on computational geometry has extensive information about kd-trees (de Berg et al. 2000).

Given the computational cost of ray tracing, there has been interest in parallel algorithms for ray tracing since shortly after the algorithm was first introduced (Cleary et al. 1983; Green and Paddon 1989; Badouel and Priol 1989). With the computational capabilities available in modern CPUs, researchers have started to demonstrate interactive ray tracing using tens of processors. For example, Parker et al. (1999) developed an interactive ray tracer on a shared-memory computer with 64 processors, and Wald and collaborators ray-traced complex scenes at interactive rates on a cluster of PCs (Wald et al. 2001a, 2002; Wald, Slusallek, and Benthin 2001; Wald, Benthin, and Slusallek 2003).

`IsPowerOf2()` 1001
`RoundUpPow2()` 1002

Task systems like the one in pbrt have been widely used in recent years. Blumofe et al. (1996) described the task scheduler in Cilk, and Blumofe and Leiserson (1999) describe the work-stealing algorithm that is the mainstay of current high-performance task systems. Boehm's paper *Threads Cannot Be Implemented as a Library* (2005) makes the remarkable (and disconcerting) observation that multithreading cannot be reliably implemented without the compiler having expicit knowledge of the fact that multithreaded execution is expected. Boehm presented a number of examples that demonstrate the danger in current compilers and language standards like C and C++. Fortunately, the forthcoming C++0x standard addresses the issues he has identified.

EXERCISES

❶ A.1 Modify MemoryArena so that it just calls new for each memory allocation. Render images of a few scenes and measure how much slower pbrt runs. Can you quantify how much of this is due to different cache behavior and how much is due to overhead in the dynamic memory management routines?

❶ A.2 Change the BlockedArray class so that it doesn't do any blocking and just uses a linear addressing scheme for the array. Measure the change in pbrt's performance as a result. (Scenes with many image map textures are most likely to show any differences, since the MIPMap class is a key user of BlockedArray.)

❷ A.3 The KdNode for the KdTree can be brought down to use just four bytes of storage; making this change may further improve its memory performance. Modify the KdNode to just store the split position and split axis in four bytes, using low two bits of the floating-point mantissa to store the split position. Then, modify the tree construction routine to build a *left-balanced* kd-tree, where the tree's topology is organized such that for the node at position i in the array of nodes, the left child is at 2*i, the right child is at 2*i+1, and the tree is balanced such that only the first nNodes elements of the array are used. How does this change affect performance for scenes that use photon mapping? Discuss why this change makes the performance difference that it did.

BlockedArray 1017
KdTree 1029
MemoryArena 1015
MIPMap 530