04 PRIMITIVES AND INTERSECTION ACCELERATION

The classes described in the last chapter focus exclusively on representing geometric properties of 3D objects. Although the `Shape` class provides a convenient abstraction for geometric operations such as intersection and bounding, it doesn't contain enough information to fully describe an object in a scene. For example, it is necessary to bind material properties to each shape in order to specify its appearance. To accomplish these goals, this chapter introduces the `Primitive` class and provides a number of implementations.

Shapes to be rendered directly are represented by the `GeometricPrimitive` class. This class combines a `Shape` with a description of its appearance properties. So that the geometric and shading portions of `pbrt` can be cleanly separated, these appearance properties are encapsulated in the `Material` class, which is described in Chapter 9.

The `TransformedPrimitive` class handles two more general uses of `Shapes` in the scene: shapes with animated transformation matrices and object instancing, which can greatly reduce the memory requirements for scenes that contain many instances of the same geometry at different locations (such as the one in Figure 4.1). Implementing each of these features essentially requires injecting an additional transformation matrix between the `Shape`'s notion of world space and the actual scene world space. Therefore, both are handled by a single class.

This chapter also introduces the `Aggregate` base class, which represents a container that can hold many `Primitives`. `pbrt` uses this class to implement *acceleration structures*—data structures that help reduce the otherwise $O(n)$ complexity of testing a ray for intersection with all n objects in a scene. Most rays will intersect only a few primitives and miss the others by a large distance. If an intersection acceleration technique can reject

Figure 4.1: This ecosystem scene makes heavy use of instancing as a mechanism for compressing the scene's description. There are only 1.1 million unique triangles in the scene, although, thanks to object reuse through instancing, the total geometric complexity is 19.5 million triangles.

whole groups of primitives at once, there will be a substantial performance improvement compared to simply testing each ray against each primitive in turn. One benefit from reusing the `Primitive` interface for these acceleration structures is that `pbrt` can support hybrid approaches where an accelerator of one type holds accelerators of other types.

This chapter describes the implementation of three accelerators, one (`GridAccel`) based on overlaying a uniform grid over the scene, one (`BVHAccel`) based on building a hierarchy of bounding boxes around objects in the scene, and the last (`KdTreeAccel`) based on adaptive recursive spatial subdivision.

4.1 PRIMITIVE INTERFACE AND GEOMETRIC PRIMITIVES

`BVHAccel` 209
`GridAccel` 196
`KdTreeAccel` 228
`Primitive` 185
`ReferenceCounted` 1010

The abstract `Primitive` base class is the bridge between the geometry processing and shading subsystems of `pbrt`. It inherits from the `ReferenceCounted` base class, which automatically tracks how many references there are to an object, freeing its storage when the last reference goes out of scope. Other classes that store `Primitives` shouldn't store pointers to them, but instead hold a `Reference<Primitive>`, which ensures that reference

counts are computed correctly. The `Reference<Primitive>` class otherwise behaves as if it was a pointer to a `Primitive`.

⟨*Primitive Declarations*⟩ ≡
```
class Primitive : public ReferenceCounted {
public:
    ⟨Primitive Interface 185⟩
    ⟨Primitive Public Data 185⟩
protected:
    ⟨Primitive Protected Data 185⟩
};
```

Like `Shapes`, `Primitives` also each have a unique 32-bit identifier. Because a single `Shape` may be represented in the scene multiple times due to object instancing, both the shape and primitive ids are necessary to uniquely identify an instance of a shape in the scene.

⟨*Primitive Interface*⟩ ≡ 185
```
Primitive() : primitiveId(nextprimitiveId++) { }
```

⟨*Primitive Public Data*⟩ ≡ 185
```
const uint32_t primitiveId;
```

⟨*Primitive Protected Data*⟩ ≡ 185
```
static uint32_t nextprimitiveId;
```

Similarly to shape ids, the first primitive id value handed out is one, so that zero can be reserved to indicate "no primitive."

⟨*Primitive Method Definitions*⟩ ≡
```
uint32_t Primitive::nextprimitiveId = 1;
```

Because the `Primitive` class connects geometry and shading, its interface contains methods related to both. There are five geometric routines in the `Primitive` interface, all of which are similar to a corresponding `Shape` method. The first, `Primitive::WorldBound()`, returns a box that encloses the primitive's geometry in world space. There are many uses for such a bound; one of the most important is to place the `Primitive` in the acceleration data structures.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual BBox WorldBound() const = 0;
```

Similarly to the `Shape` class, all primitives must be able to either determine if a given ray intersects their geometry or else refine themselves into one or more new primitives. Like the `Shape` interface, `Primitive` has a `Primitive::CanIntersect()` method so that `pbrt` can determine whether the underlying geometry is intersectable or not.

One difference from the `Shape` interface is that the `Primitive` intersection methods return `Intersection` structures rather than `DifferentialGeometry`. These `Intersection` structures hold more information about the intersection than just the local geometric information, such as information about the material properties at the hit point.

BBox 70
DifferentialGeometry 102
Intersection 186
Primitive 185
Primitive::
CanIntersect() 186
Primitive::WorldBound() 185
ReferenceCounted 1010
Shape 108

Another difference is that `Shape::Intersect()` returns the parametric distance along the ray to the intersection in a `float *` output variable, while `Primitive::Intersect()` is responsible for updating `Ray::maxt` with this value if an intersection is found.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual bool CanIntersect() const;
virtual bool Intersect(const Ray &r, Intersection *in) const = 0;
virtual bool IntersectP(const Ray &r) const = 0;
virtual void Refine(vector<Reference<Primitive> > &refined) const;
```

The `Intersection` structure holds information about a ray–primitive intersection, including information about the differential geometry of the point on the surface, a pointer to the `Primitive` that the ray hit, and its world-to-object-space transformation. It is defined in the files `core/intersection.h` and `core/intersection.cpp`.

⟨*Intersection Declarations*⟩ ≡
```
struct Intersection {
    ⟨Intersection Public Methods 484⟩
    ⟨Intersection Public Data 186⟩
};
```

⟨*Intersection Public Data*⟩ ≡ 186
```
DifferentialGeometry dg;
const Primitive *primitive;
Transform WorldToObject, ObjectToWorld;
uint32_t shapeId, primitiveId;
float rayEpsilon;
```

It may be necessary to repeatedly refine a primitive until all of the primitives it has returned are themselves intersectable. The `Primitive::FullyRefine()` utility method handles this task. Its implementation is straightforward. It maintains a queue of primitives to be refined (called `todo` in the following code) and invokes the `Primitive::Refine()` method repeatedly on entries in that queue. Intersectable `Primitives` returned by `Primitive::Refine()` are placed in the `refined` array, while nonintersectable ones are placed back on the `todo` list by the `Refine()` routine.

⟨*Primitive Method Definitions*⟩ +≡
```
void
Primitive::FullyRefine(vector<Reference<Primitive> > &refined) const {
    vector<Reference<Primitive> > todo;
    todo.push_back(const_cast<Primitive *>(this));
    while (todo.size()) {
        ⟨Refine last primitive in todo list 187⟩
    }
}
```

DifferentialGeometry 102
Intersection 186
Primitive 185
Primitive::FullyRefine() 186
Primitive::Intersect() 186
Primitive::Refine() 186
Ray 66
Ray::maxt 67
Reference 1011
Shape::Intersect() 111
Transform 76

⟨*Refine last primitive in todo list*⟩ ≡ 186
```
Reference<Primitive> prim = todo.back();
todo.pop_back();
if (prim->CanIntersect())
    refined.push_back(prim);
else
    prim->Refine(todo);
```

In addition to the geometric methods, a `Primitive` object has three methods related to its material properties. The first, `Primitive::GetAreaLight()`, returns a pointer to the `AreaLight` that describes the primitive's emission distribution, if the primitive is itself a light source. If the primitive is not emissive, this method should return `NULL`.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual const AreaLight *GetAreaLight() const = 0;
```

The other two methods return representations of the light-scattering properties of the material at the given point on the surface. The first, `Primitive::GetBSDF()`, returns a `BSDF` object (introduced in Section 9.1) that describes local light-scattering properties at the intersection point. In addition to the differential geometry at the hit point, this method takes the object-to-world-space transformation and a `MemoryArena` to allocate memory for the returned `BSDF`. Section 9.1.1 discusses the use of the `MemoryArena` for `BSDF` memory allocation in more detail.

The second method, `Primitive::GetBSSRDF()`, returns a `BSSRDF`, which describes subsurface scattering inside the primitive—light that enters the surface at points far from where it exits. While subsurface light transport has little effect on the appearance of objects like metal, cloth, or plastic, it is the dominant light-scattering mechanism for biological materials like skin, thick liquids like milk, etc. The `BSSRDF` is used by the subsurface light transport integrator defined in Section 16.5.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual BSDF *GetBSDF(const DifferentialGeometry &dg,
    const Transform &ObjectToWorld, MemoryArena &arena) const = 0;
virtual BSSRDF *GetBSSRDF(const DifferentialGeometry &dg,
    const Transform &ObjectToWorld, MemoryArena &arena) const = 0;
```

AreaLight 623
BSDF 478
BSSRDF 598
DifferentialGeometry 102
GeometricPrimitive 188
MemoryArena 1015
Primitive 185
Primitive::
  CanIntersect() 186
Primitive::
  GetAreaLight() 187
Primitive::GetBSDF() 187
Primitive::GetBSSRDF() 187
Primitive::Refine() 186
Reference 1011
Transform 76

4.1.1 GEOMETRIC PRIMITIVES

The `GeometricPrimitive` class represents a single shape (e.g., a sphere) in the scene. One `GeometricPrimitive` is allocated for each shape in the scene description provided by the user. It is implemented in the files `core/primitive.h` and `core/primitive.cpp`.

⟨*GeometricPrimitive Declarations*⟩ ≡
```
class GeometricPrimitive : public Primitive {
public:
    ⟨GeometricPrimitive Public Methods 188⟩
private:
    ⟨GeometricPrimitive Private Data 188⟩
};
```

Each GeometricPrimitive holds a reference to a Shape and its Material. In addition, because primitives in pbrt may be area light sources, it stores a pointer to an AreaLight object that describes its emission characteristics (this pointer is set to NULL if the primitive does not emit light).

⟨*GeometricPrimitive Private Data*⟩ ≡ 187
```
Reference<Shape> shape;
Reference<Material> material;
AreaLight *areaLight;
```

The GeometricPrimitive constructor initializes these variables from the parameters passed to it. Its implementation is omitted.

⟨*GeometricPrimitive Public Methods*⟩ ≡ 187
```
GeometricPrimitive(const Reference<Shape> &s,
                   const Reference<Material> &m, AreaLight *a);
```

Most of the methods of the Primitive interface related to geometric processing are simply forwarded to the corresponding Shape method. For example, GeometricPrimitive::Intersect() calls the Shape::Intersect() method of its enclosed Shape to do the actual geometric intersection and initializes an Intersection object to describe the intersection, if any. It also uses the returned parametric hit distance to update the Ray::maxt member. The primary advantage of storing the distance to the closest hit in Ray::maxt is that this makes it easy to avoid performing intersection tests with any primitives that lie farther along the ray than any already-found intersections.

⟨*GeometricPrimitive Method Definitions*⟩ ≡
```
bool GeometricPrimitive::Intersect(const Ray &r,
                                   Intersection *isect) const {
    float thit, rayEpsilon;
    if (!shape->Intersect(r, &thit, &rayEpsilon, &isect->dg))
        return false;
    isect->primitive = this;
    isect->WorldToObject = *shape->WorldToObject;
    isect->ObjectToWorld = *shape->ObjectToWorld;
    isect->shapeId = shape->shapeId;
    isect->primitiveId = primitiveId;
    isect->rayEpsilon = rayEpsilon;
    r.maxt = thit;
    return true;
}
```

We won't include the implementations of the GeometricPrimitive's WorldBound(), IntersectP(), CanIntersect(), or Refine() methods here; they just forward these requests on to the Shape in a similar manner. Similarly, GetAreaLight() just returns the GeometricPrimitive::areaLight member.

The GetBSDF() method uses the Primitive's Shape to find the shading geometry at the point and forwards the request on to the Material.

AreaLight 623
GeometricPrimitive 188
GeometricPrimitive::areaLight 188
GeometricPrimitive::Intersect() 188
Intersection 186
Intersection::dg 186
Intersection::ObjectToWorld 186
Intersection::primitive 186
Intersection::primitiveId 186
Intersection::rayEpsilon 186
Intersection::shapeId 186
Intersection::WorldToObject 186
Material 483
Primitive 185
Primitive::primitiveId 185
Ray 66
Ray::maxt 67
Reference 1011
Shape 108
Shape::Intersect() 111
Shape::shapeId 109
Shape::WorldToObject 108

⟨*GeometricPrimitive Method Definitions*⟩ +≡

```
BSDF *GeometricPrimitive::GetBSDF(const DifferentialGeometry &dg,
                                  const Transform &ObjectToWorld,
                                  MemoryArena &arena) const {
    DifferentialGeometry dgs;
    shape->GetShadingGeometry(ObjectToWorld, dg, &dgs);
    return material->GetBSDF(dg, dgs, arena);
}
```

The `GeometricPrimitive::GetBSSRDF()` method is similar; it also computes the shading geometry and returns the `BSSRDF` returned by the `Material::GetBSSRDF()` method.

4.1.2 `TransformedPrimitive`: OBJECT INSTANCING AND ANIMATED PRIMITIVES

`TransformedPrimitive` holds a single `Primitive` and also includes an `AnimatedTransform` that is essentially injected in between the underlying primitive and its representation in the scene. This extra transformation enables two useful features: object instancing and primitives with animated transformations.

Object instancing is a classic technique in rendering that reuses transformed copies of a single collection of geometry at multiple positions in a scene. For example, in a model of a concert hall with thousands of identical seats, the scene description can be compressed substantially if all of the seats refer to a shared geometric representation of a single seat. The ecosystem scene in Figure 4.1 has over 4000 individual plants of various types, although only 61 unique plant models. Because each plant model is instanced multiple times with a different transformation for each instance, the complete scene has a total of 19.5 million triangles, although only 1.1 million triangles are stored in memory, thanks to primitive reuse through object instancing. `pbrt` uses approximately 600 MB of memory when rendering this scene with object instancing, but would need upwards of 11 GB to render it without instancing.

Animated transformations enable rigid-body animation of primitives in the scene via the `AnimatedTransform` class. See Figure 2.14 for an image that exhibits motion blur due to animated transformations.

Recall that the `Shapes` of Chapter 3 themselves had object-to-world transformations applied to them to place them in the scene. If a shape is held by a `TransformedPrimitive`, then the shape's notion of world space isn't the actual scene world space—only after the `TransformedPrimitive`'s transformation is also applied is the shape actually in world space. For the applications here, it makes sense for the shape to not be at all aware of the additional transformations being applied. For animated shapes, it's simpler to isolate all of the handling of animated transformations to a single class here, rather than require all `Shapes` to support `AnimatedTransforms`. Similarly, for instanced primitives, letting `Shapes` know all of the instance transforms is of limited utility: we wouldn't want the `TriangleMesh` to make a copy of its vertex positions for each instance transformation and transform them all the way to world space, since this would negate the memory savings of object instancing.

The `TransformedPrimitive` constructor takes a reference to the `Primitive` that represents the model, and the transformation that places it in the scene. If the geometry is described

AnimatedTransform 96
BSDF 478
BSSRDF 598
DifferentialGeometry 102
GeometricPrimitive 188
GeometricPrimitive::
material 188
GeometricPrimitive::
shape 188
Material::GetBSDF() 483
Material::GetBSSRDF() 484
MemoryArena 1015
Primitive 185
Shape 108
Shape::
GetShadingGeometry() 113
Transform 76
TransformedPrimitive 190
TriangleMesh 135

by multiple Primitives, the calling code is responsible for placing them in an Aggregate class so that only a single Primitive needs to be stored here.

The TransformedPrimitive also requires that the primitive be intersectable. (In the presence of object instancing, it would be a waste of both time and memory for all of the instances to individually refine the primitive.) For the code that refines shapes and creates aggregates as needed, see the pbrtObjectInstance() function in Section B.3.6 of Appendix B for the code that creates primitive instances, and see the pbrtShape() function in Section B.3.5 for the corresponding code for animated shapes.

⟨*TransformedPrimitive Declarations*⟩ ≡
```
class TransformedPrimitive : public Primitive {
public:
    ⟨TransformedPrimitive Public Methods 190⟩
private:
    ⟨TransformedPrimitive Private Data 190⟩
};
```

⟨*TransformedPrimitive Public Methods*⟩ ≡ 190
```
TransformedPrimitive(Reference<Primitive> &prim,
                     const AnimatedTransform &w2p)
    : primitive(prim), WorldToPrimitive(w2p) { }
```

⟨*TransformedPrimitive Private Data*⟩ ≡ 190
```
Reference<Primitive> primitive;
const AnimatedTransform WorldToPrimitive;
```

The key task of the TransformedPrimitive is to bridge the Primitive interface that it implements and the Primitive that it holds a reference to, accounting for the effects of the additional transformation matrix that it holds. The TransformedPrimitive's WorldToPrimitive transformation defines the transformation from world space to the coordinate system of this particular instance of the geometry. The primitive member has its own transformation that should be interpreted as the transformation from a TransformedPrimitive's coordinate system to object space. The complete transformation to world space requires both of these transformations together.

Thus, the TransformedPrimitive::Intersect() method transforms the given ray to the primitive's coordinate system and passes the transformed ray to its Intersect() routine. If a hit is found, the maxt value from the transformed ray needs to be copied into the ray r originally passed to the Intersect() routine and the Intersection's primitiveId member is overridden with the primitive id of this TransformedPrimitive.

⟨*TransformedPrimitive Method Definitions*⟩ ≡
```
bool TransformedPrimitive::Intersect(const Ray &r,
                                     Intersection *isect) const {
    Transform w2p;
    WorldToPrimitive.Interpolate(r.time, &w2p);
    Ray ray = w2p(r);
    if (!primitive->Intersect(ray, isect))
        return false;
```

Aggregate 192
AnimatedTransform 96
AnimatedTransform::Interpolate() 99
Intersection 186
Intersection::primitiveId 186
pbrtObjectInstance() 1070
pbrtShape() 1065
Primitive 185
Primitive::Intersect() 186
Primitive::primitiveId 185
Ray 66
Ray::maxt 67
Ray::time 67
Reference 1011
Transform 76
Transform::IsIdentity() 77
Transform::operator() 86
TransformedPrimitive 190
TransformedPrimitive::Intersect() 190
TransformedPrimitive::primitive 190
TransformedPrimitive::WorldToPrimitive 190

```
        r.maxt = ray.maxt;
        isect->primitiveId = primitiveId;
        if (!w2p.IsIdentity()) {
            ⟨Compute world-to-object transformation for instance 191⟩
            ⟨Transform instance's differential geometry to world space 191⟩
        }
        return true;
    }
```

The Transforms in the Intersection must be set properly as well; we need to compute the full transformation all the way from the primitive's object space to the actual world space, multiplying both of the relevant Transforms together.

⟨*Compute world-to-object transformation for instance*⟩ ≡ 190

```
isect->WorldToObject = isect->WorldToObject * w2p;
isect->ObjectToWorld = Inverse(isect->WorldToObject);
```

Finally, the DifferentialGeometry at the intersection point needs to be transformed to world space; the primitive member will already have transformed the differential geometry information to its notion of world space, so here we only need to apply the effect of the additional transformation held here.

⟨*Transform instance's differential geometry to world space*⟩ ≡ 190

```
Transform PrimitiveToWorld = Inverse(w2p);
isect->dg.p = PrimitiveToWorld(isect->dg.p);
isect->dg.nn = Normalize(PrimitiveToWorld(isect->dg.nn));
isect->dg.dpdu = PrimitiveToWorld(isect->dg.dpdu);
isect->dg.dpdv = PrimitiveToWorld(isect->dg.dpdv);
isect->dg.dndu = PrimitiveToWorld(isect->dg.dndu);
isect->dg.dndv = PrimitiveToWorld(isect->dg.dndv);
```

The rest of the geometric Primitive methods are forwarded on to the shared instance, with the results similarly transformed as needed by the TransformedPrimitive's transformation.

⟨*TransformedPrimitive Public Methods*⟩ +≡ 190

```
BBox WorldBound() const {
    return WorldToPrimitive.MotionBounds(primitive->WorldBound(), true);
}
```

The TransformedPrimitive::GetAreaLight(), TransformedPrimitive::GetBSDF(), and TransformedPrimitive::GetBSSRDF() methods should never be called. The corresponding methods in the primitive that the ray actually hit will be called instead. Calling the TransformedPrimitive implementations (not shown here) results in a run time error.

AnimatedTransform::
MotionBounds() 100
BBox 70
DifferentialGeometry 102
Intersection 186
Intersection::
WorldToObject 186
Primitive 185
Primitive::WorldBound() 185
Transform 76
Transform::Inverse() 77
TransformedPrimitive 190
TransformedPrimitive::
GetAreaLight() 191
TransformedPrimitive::
GetBSDF() 191
TransformedPrimitive::
GetBSSRDF() 191
TransformedPrimitive::
primitive 190
TransformedPrimitive::
WorldToPrimitive 190
Vector::Normalize() 63

4.2 AGGREGATES

Acceleration structures are one of the components at the heart of any ray tracer. Without algorithms to reduce the number of unnecessary ray intersection tests, tracing a single ray through a scene would take time linear in the number of primitives in the scene, since the

ray would need to be tested against each primitive in turn to find the closest intersection. However, doing so is extremely wasteful in most scenes, since the ray passes nowhere near the vast majority of primitives. The goal of acceleration structures is to allow the quick, simultaneous rejection of *groups* of primitives and also to order the search process so that nearby intersections are likely to be found first and farther away ones can potentially be ignored.

Because ray–object intersections can account for the bulk of execution time in ray tracers, there has been a substantial amount of research into algorithms for ray intersection acceleration. We will not try to explore all of this work here, but refer the interested reader to references in the "Further Reading" section at the end of this chapter and in particular Arvo and Kirk's chapter in *An Introduction to Ray Tracing* (Glassner 1989a), which has a useful taxonomy for classifying different approaches to ray-tracing acceleration.

Broadly speaking, there are two main approaches to this problem: spatial subdivision and object subdivision. Spatial subdivision algorithms decompose 3D space into regions (e.g., by superimposing a grid of axis-aligned boxes on the scene) and record which primitives overlap which regions. In some algorithms, the regions may also be adaptively subdivided based on the number of primitives that overlap them. When a ray intersection needs to be found, the sequence of these regions that the ray passes through is computed and only the primitives in the overlapping regions are tested for intersection.

In contrast, object subdivision is based on progressively breaking the objects in the scene down into smaller sets of constituent objects. For example, a model of a room might be broken down into four walls, a ceiling, and a chair. If a ray doesn't intersect the room's bounding volume, then all of its primitives can be culled. Otherwise, the ray is tested against each of them. If it hits the chair's bounding volume, for example, then it might be tested against each of its legs, the seat, and the back. Otherwise, the chair is culled.

Both of these approaches have been quite successful at solving the general problem of ray intersection computational requirements; there's no fundamental reason to prefer one over the other. The `GridAccel` and `KdTreeAccel` in this chapter are both based on the spatial subdivision approach, and the `BVHAccel` is based on object subdivision.

The `Aggregate` class provides an interface for grouping multiple `Primitive` objects together. Because `Aggregates` themselves implement the `Primitive` interface, no special support is required elsewhere in `pbrt` for intersection acceleration. `Integrators` can be written as if there was just a single `Primitive` in the scene, checking for intersections without needing to be concerned about how they're actually found. Furthermore, by implementing acceleration in this way, it is easy to experiment with new acceleration techniques by simply adding a new `Aggregate` primitive to `pbrt`.

⟨*Aggregate Declarations*⟩ ≡
```
class Aggregate : public Primitive {
public:
    ⟨Aggregate Public Methods⟩
};
```

Like `TransformedPrimitives`, `Aggregate` intersection routines leave the `Intersection::primitive` pointer set to the primitive that the ray actually hit, not the aggregate that

Aggregate 192
BVHAccel 209
GridAccel 196
Integrator 740
Intersection::primitive 186
KdTreeAccel 228
Primitive 185
TransformedPrimitive 190

holds the primitive. Because pbrt uses this pointer to obtain information about the primitive being hit (its reflection and emission properties), the Aggregate::GetAreaLight(), Aggregate::GetBSDF(), and Aggregate::GetBSSRDF() methods should never be called, so the implementations of those methods (not shown here) report a run time error.

4.2.1 RAY–BOX INTERSECTIONS

All of the accelerators in this chapter store a BBox that surrounds all of their primitives. This box can be used to quickly determine if a ray doesn't intersect any of the primitives; if the ray misses the box, it also must miss all of the primitives inside it. Furthermore, some accelerators use the point at which the ray enters the bounding box and the point at which it exits as part of the input to their traversal algorithms. Therefore, we will add a BBox method, BBox::IntersectP(), that checks for a ray–box intersection and returns the two parametric t values of the intersection, if any.

One way to think of bounding boxes is as the intersection of three slabs, where a slab is the region of space between two parallel planes. To intersect a ray against a box, we intersect the ray against each of the box's three slabs in turn. Because the slabs are aligned with the three coordinate axes, a number of optimizations can be made in the ray–slab tests.

The basic ray-bounding box intersection algorithm works as follows: We start with a parametric interval that covers that range of positions t along the ray where we're interested in finding intersections; typically, this is $[0, \infty)$. We will then successively compute the two parametric t positions where the ray intersects each axis-aligned slab. We compute the set intersection of the per-slab intersection interval with the current BBox intersection interval, returning failure if we find that the resulting interval is degenerate. If, after checking all three slabs, the interval is nondegenerate, we have the parametric range of the ray that is inside the box. Figure 4.2 illustrates this process, and Figure 4.3 shows the basic geometry of a ray and a slab.

If the BBox::IntersectP() method returns true, the intersection's parametric range is returned in the optional arguments hitt0 and hitt1. Intersections outside of the

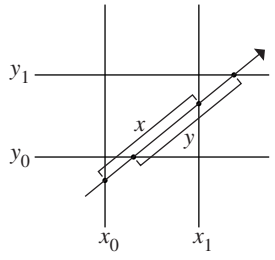

Figure 4.2: Intersecting a Ray with an Axis-Aligned Bounding Box. We compute intersection points with each slab in turn, progressively narrowing the parametric interval. Here, in 2D, the intersection of the x and y extents along the ray gives the extent where the ray is inside the box.

BBox 70
BBox::IntersectP() 194

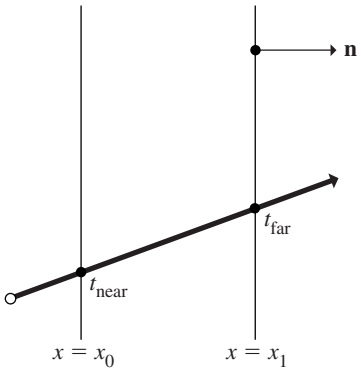

Figure 4.3: Intersecting a Ray with an Axis-Aligned Slab. The two planes shown here are described by $x = c$ for some constant value c. The normal of each plane is $(1, 0, 0)$. Unless the ray is parallel to the planes, it will intersect the slab twice, at parametric positions t_{near} and t_{far}.

`[Ray::mint, Ray::maxt]` range of the ray are ignored. If the ray's starting point, `ray(ray.mint)`, is inside the box, `ray.mint` is returned for `hitt0`.

⟨*BBox Method Definitions*⟩ +≡

```
bool BBox::IntersectP(const Ray &ray, float *hitt0,
                      float *hitt1) const {
    float t0 = ray.mint, t1 = ray.maxt;
    for (int i = 0; i < 3; ++i) {
        ⟨Update interval for ith bounding box slab 195⟩
    }
    if (hitt0) *hitt0 = t0;
    if (hitt1) *hitt1 = t1;
    return true;
}
```

For each pair of planes, this routine needs to compute two ray–plane intersections, giving the parametric t values where the intersections occur. Consider the slab along the x axis: it can be described by the two planes through the points $(x_1, 0, 0)$ and $(x_2, 0, 0)$, each with normal $(1, 0, 0)$. Consider the first t value for a plane intersection, t_1. The general form of the intersection between a ray with origin o and direction $\mathbf{d}$ and a plane $ax + by + cz + d = 0$ can be shown to be

$$t = \frac{-d - (\mathrm{o} \cdot (a, b, c))}{(\mathbf{d} \cdot (a, b, c))}.$$

Because the y and z components of the plane's normal are zero, b and c are zero, and a is one. The plane's d coefficient is $-x_1$. We can use this information and the definition of the dot product to simplify this substantially:

$$t_1 = \frac{x_1 - \mathrm{o}_x}{\mathbf{d}_x}.$$

BBox 70
Ray 66
Ray::maxt 67
Ray::mint 67

The code to compute these values starts by computing the reciprocal of the corresponding component of the ray direction so that it can multiply by this factor instead of performing multiple divisions. Note that, although it divides by this component, it is not necessary to verify that it is nonzero. If it is zero, then `invRayDir` will hold an infinite value, either $-\infty$ or ∞, and the rest of the algorithm still works correctly.[1]

⟨*Update interval for* `i`*th bounding box slab*⟩ ≡ 194
```
float invRayDir = 1.f / ray.d[i];
float tNear = (pMin[i] - ray.o[i]) * invRayDir;
float tFar  = (pMax[i] - ray.o[i]) * invRayDir;
⟨Update parametric interval from slab intersection ts 195⟩
```

The two distances are reordered so that `tNear` holds the closer intersection and `tFar` the farther one. This gives a parametric range [`tNear`, `tFar`], which is used to compute the set intersection with the current range [`t0`, `t1`] to compute a new range. If this new range is empty (i.e., `t0` > `t1`), then the code can immediately return failure. There is another floating-point-related subtlety here: in the case where the ray origin is in the plane of one of the bounding box slabs and the ray lies in the plane of the slab, it is possible that `tNear` or `tFar` will be computed by an expression of the form 0/0, which results in an IEEE floating-point "not a number" (NaN) value. Like infinity values, NaNs have well-specified semantics: for example, any logical comparison involving a NaN always evaluates to false. Therefore, the code that updates the values of `t0` and `t1` is carefully written so that if `tNear` or `tFar` is NaN, then `t0` or `t1` won't ever take on a NaN value but will always remain unchanged.

⟨*Update parametric interval from slab intersection ts*⟩ ≡ 195
```
if (tNear > tFar) swap(tNear, tFar);
t0 = tNear > t0 ? tNear : t0;
t1 = tFar  < t1 ? tFar  : t1;
if (t0 > t1) return false;
```

4.3 GRID ACCELERATOR

`GridAccel` is an accelerator that divides an axis-aligned region of space into equal-sized box-shaped chunks (called *voxels*). Each voxel stores references to the primitives that overlap it (Figure 4.4). Given a ray, the grid steps through each of the voxels that the ray passes through in order, checking for intersections with only the primitives in each voxel. Useless ray intersection tests are reduced substantially because primitives far away from the ray aren't considered at all. Furthermore, because the voxels are considered from near to far along the ray, it is possible to stop performing intersection tests once an intersection has been found and it is certain that it is not possible for there to be any closer intersections.

BBox::pMax 71
BBox::pMin 71
GridAccel 196
Ray::d 67
Ray::o 67

1 This assumes that the architecture being used supports IEEE floating-point arithmetic (Institute of Electrical and Electronic Engineers 1985), which is universal on modern systems. The relevant properties of IEEE floating-point arithmetic are that for all $v > 0$, $v/0 = \infty$ and for all $w < 0$, $w/0 = -\infty$, where ∞ is a special value such that any positive number multiplied by ∞ gives ∞, any negative number multiplied by ∞ gives $-\infty$, and so on.

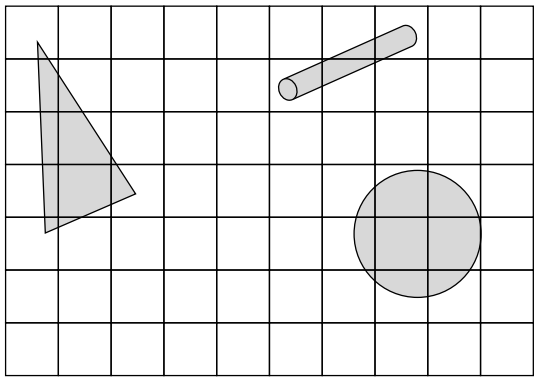

Figure 4.4: The regular grid accelerator divides space into regularly sized cells. Each one stores a reference to the `Primitives` that overlap it.

The `GridAccel` structure can be initialized quickly, and a simple computation determines the sequence of voxels through which a given ray passes. However, this simplicity is a doubled-edged sword. `GridAccel` can suffer from poor performance when the primitives in the scene aren't distributed evenly throughout space. If there's a small region of space with a lot of geometry in it, all that geometry might fall in a single voxel, and performance will suffer when a ray passes through that voxel, as many intersection tests will be performed. This is sometimes referred to as the "teapot in a stadium" problem; it is not unusual to have such a variable distribution of geometry in realistic scenes.

The root problem is that the data structure cannot adapt well to the distribution of the data that it is storing: if a very fine grid is used, too much time is spent stepping through empty space, and if the grid is too coarse, there is little benefit from the grid at all. The `BVHAccel` and the `KdTreeAccel` in the next two sections adapt to the distribution of geometry such that they don't suffer from this problem.

The implementation of `pbrt`'s grid accelerator is defined in `accelerators/grid.h` and `accelerators/grid.cpp`.

⟨*GridAccel Declarations*⟩ ≡

```
class GridAccel : public Aggregate {
public:
    ⟨GridAccel Public Methods 208⟩
private:
    ⟨GridAccel Private Methods 200⟩
    ⟨GridAccel Private Data 198⟩
};
```

Aggregate 192
BVHAccel 209
GridAccel 196
KdTreeAccel 228
Primitive 185

4.3.1 CREATION

The GridAccel constructor takes a vector of Primitives to be stored in the grid. It automatically determines the number of voxels to store in the grid based on the number of primitives.

One factor that adds to the complexity of the grid's implementation is the fact that some of these primitives may not be directly intersectable (they may return false from Primitive::CanIntersect()) and need to refine themselves into subprimitives before intersection tests can be performed. This is a problem because when the grid is being built we might have a scene with a single primitive in it and choose to build a coarse grid with few voxels. However, if the primitive is later refined for intersection tests, it might turn into millions of primitives, and the original grid resolution would be far too small to efficiently find intersections. pbrt addresses this problem in one of two ways:

- If the refineImmediately flag to the grid constructor is true, all of the Primitives are refined until they have turned into intersectable primitives. This may waste time and memory for scenes where some of the primitives wouldn't ever need to be refined since no rays approached them.
- Otherwise, primitives are refined only when a ray enters one of the voxels they are stored in. If they create multiple Primitives when refined, the new primitives are stored in a new instance of a GridAccel that replaces the original Primitive in the top-level grid. This allows the implementation to handle primitive refinement without needing to rebuild the entire grid each time another primitive is refined.

Lazy refinement of primitives in the grid introduces some issues related to multithreaded synchronization (recall the discussion of this topic in Section 1.3.5); the issue is that if one thread is traversing the grid while another thread is modifying its contents by refining primitives in the grid, we need to ensure that one thread's modification of the shared data doesn't cause the other thread to access a partially updated or an otherwise inconsistent representation of the scene. We will discuss these issues (and a solution to them) further later in this section.

⟨*GridAccel Method Definitions*⟩ ≡

```
GridAccel::GridAccel(const vector<Reference<Primitive> > &p,
                     bool refineImmediately) {
    ⟨Initialize primitives with primitives for grid 198⟩
    ⟨Compute bounds and choose grid resolution 198⟩
    ⟨Compute voxel widths and allocate voxels 199⟩
    ⟨Add primitives to grid voxels 199⟩
    ⟨Create reader-writer mutex for grid 205⟩
}
```

GridAccel 196
Primitive 185
Primitive::CanIntersect() 186
Reference 1011

First, the constructor determines the set of Primitives to store in the grid, either directly using the primitives passed in or refining all of them until they are intersectable.

⟨*Initialize* primitives *with primitives for grid*⟩ ≡ 197
```
if (refineImmediately)
    for (uint32_t i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(primitives);
else
    primitives = p;
```

⟨*GridAccel Private Data*⟩ ≡ 196
```
vector<Reference<Primitive> > primitives;
```

The constructor next computes the overall bounds of the primitives and determines how many voxels to create along each of the x, y, and z axes. The `voxelsPerUnitDist` value, computed in a later fragment, gives the average number of voxels that should be created per unit distance in each of the three directions. Given that value, multiplication by the grid's extent in each direction gives the number of voxels to make. The number of voxels in any direction is capped at 64 to avoid creating enormous data structures for complex scenes.

⟨*Compute bounds and choose grid resolution*⟩ ≡ 197
```
for (uint32_t i = 0; i < primitives.size(); ++i)
    bounds = Union(bounds, primitives[i]->WorldBound());
Vector delta = bounds.pMax - bounds.pMin;
⟨Find voxelsPerUnitDist for grid 199⟩
for (int axis = 0; axis < 3; ++axis) {
    nVoxels[axis] = Round2Int(delta[axis] * voxelsPerUnitDist);
    nVoxels[axis] = Clamp(nVoxels[axis], 1, 64);
}
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
int nVoxels[3];
BBox bounds;
```

As a first approximation to choosing a grid size, the total number of voxels should be roughly proportional to the total number of primitives. If the primitives were uniformly distributed, this would mean that a constant number of primitives were in each voxel. While increasing the number of voxels improves efficiency by reducing the average number of primitives per voxel (and thus reducing the number of ray–object intersection tests that need to be performed), doing so also increases memory use, hurts cache performance, and increases the time spent tracing the ray's path through the greater number of voxels it overlaps. On the other hand, too few voxels obviously leads to poor performance, due to an increased number of ray–primitive intersection tests to be performed.

Given the goal of having the number of voxels be proportional to the number of primitives, the cube root of the number of objects is an appropriate starting point for the grid resolution in each direction. In practice, this value is typically scaled by an empirically chosen factor; in `pbrt` we use a scale of three. Whichever of the x, y, or z dimensions has the largest extent will have exactly $3\sqrt[3]{N}$ voxels for a scene with N primitives. The number of voxels in the other two directions are set in an effort to create voxels that are as close to cubes as possible. The `voxelsPerUnitDist` variable is the foundation of these

BBox 70
BBox::pMax 71
BBox::pMin 71
Clamp() 1000
GridAccel::nVoxels 198
GridAccel::primitives 198
Primitive 185
Primitive::FullyRefine() 186
Primitive::WorldBound() 185
Reference 1011
Round2Int() 1002
Union() 72
Vector 57

computations; it gives the number of voxels to create per unit distance. Its value is set such that cubeRoot voxels will be created along the axis with the largest extent.

⟨*Find* voxelsPerUnitDist *for grid*⟩ ≡ 198
```
int maxAxis = bounds.MaximumExtent();
float invMaxWidth = 1.f / delta[maxAxis];
float cubeRoot = 3.f * powf(float(primitives.size()), 1.f/3.f);
float voxelsPerUnitDist = cubeRoot * invMaxWidth;
```

Given the number of voxels in each dimension, the constructor sets GridAccel::width, which holds the world space widths of the voxels in each direction. It also precomputes the GridAccel::invWidth values, so that routines that would otherwise divide by the width value can perform a multiplication rather than dividing. Finally, it allocates an array of pointers to Voxel structures for each of the voxels in the grid. These pointers are set to NULL initially and will be allocated only for any voxel with one or more overlapping primitives.[2]

⟨*Compute voxel widths and allocate voxels*⟩ ≡ 197
```
for (int axis = 0; axis < 3; ++axis) {
    width[axis] = delta[axis] / nVoxels[axis];
    invWidth[axis] = (width[axis] == 0.f) ? 0.f : 1.f / width[axis];
}
int nv = nVoxels[0] * nVoxels[1] * nVoxels[2];
voxels = AllocAligned<Voxel *>(nv);
memset(voxels, 0, nv * sizeof(Voxel *));
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
Vector width, invWidth;
Voxel **voxels;
```

Once the voxels themselves have been allocated, primitives can be added to the voxels that they overlap. The GridAccel constructor adds each primitive's corresponding Primitive pointer to the voxels that its bounding box overlaps.

⟨*Add primitives to grid voxels*⟩ ≡ 197
```
for (uint32_t i = 0; i < primitives.size(); ++i) {
    ⟨Find voxel extent of primitive 200⟩
    ⟨Add primitive to overlapping voxels 200⟩
}
```

First, the world space bounds of the primitive are converted to the integer voxel coordinates that contain its two opposite corners. This is done by the utility function GridAccel::posToVoxel(), which turns a world space (x, y, z) position into the coordinates of the voxel that contains that point.

AllocAligned() 1013
BBox::MaximumExtent() 73
GridAccel 196
GridAccel::invWidth 199
GridAccel::nVoxels 198
GridAccel::posToVoxel() 200
GridAccel::width 199
Primitive 185
Vector 57
Voxel 202

2 Some grid implementations try to save even more memory by using a hash table from (x, y, z) voxel number to voxel structures. This saves the memory for the voxels array, which may be substantial if the grid has very small voxels, and the vast majority of them are empty. However, this approach increases the computational expense of finding the Voxel structure for each voxel that a ray passes through.

⟨*Find voxel extent of primitive*⟩ ≡ 199
```
BBox pb = primitives[i]->WorldBound();
int vmin[3], vmax[3];
for (int axis = 0; axis < 3; ++axis) {
    vmin[axis] = posToVoxel(pb.pMin, axis);
    vmax[axis] = posToVoxel(pb.pMax, axis);
}
```

⟨*GridAccel Private Methods*⟩ ≡ 196
```
int posToVoxel(const Point &P, int axis) const {
    int v = Float2Int((P[axis] - bounds.pMin[axis]) *
                      invWidth[axis]);
    return Clamp(v, 0, nVoxels[axis]-1);
}
```

The `GridAccel::voxelToPos()` method is the opposite of `GridAccel::posToVoxel()`; it returns the position of a particular voxel's lower corner.

⟨*GridAccel Private Methods*⟩ +≡ 196
```
float voxelToPos(int p, int axis) const {
    return bounds.pMin[axis] + p * width[axis];
}
```

The primitive is now added to all of the voxels that its bounds overlap. Using its bounds for this test is a conservative test for voxel overlap—at worst it will overestimate the voxels that the primitive overlaps. Figure 4.5 shows an example of two cases where this method leads to primitives being stored in more voxels than necessary. Exercise 4.5 at the end of this chapter describes a more accurate method for associating primitives with voxels.

⟨*Add primitive to overlapping voxels*⟩ ≡ 199
```
for (int z = vmin[2]; z <= vmax[2]; ++z)
    for (int y = vmin[1]; y <= vmax[1]; ++y)
        for (int x = vmin[0]; x <= vmax[0]; ++x) {
            int o = offset(x, y, z);
            if (!voxels[o]) {
                ⟨Allocate new voxel and store primitive in it 201⟩
            }
            else {
                ⟨Add primitive to already-allocated voxel 201⟩
            }
        }
```

The `GridAccel::offset()` utility functions give the offset into the `voxels` array for a particular (x, y, z) voxel. It is the standard indexing scheme in C++ for encoding a multidimensional array in a 1D array. We have localized this computation into a separate function, however, in order to make it easier to experiment with different array layouts, such as blocked schemes for improved cache performance.

BBox 70
BBox::pMax 71
BBox::pMin 71
Clamp() 1000
Float2Int() 1002
GridAccel::invWidth 199
GridAccel::nVoxels 198
GridAccel::offset() 201
GridAccel::posToVoxel() 200
GridAccel::voxels 199
GridAccel::voxelToPos() 200
GridAccel::width 199
Point 63
Primitive::WorldBound() 185

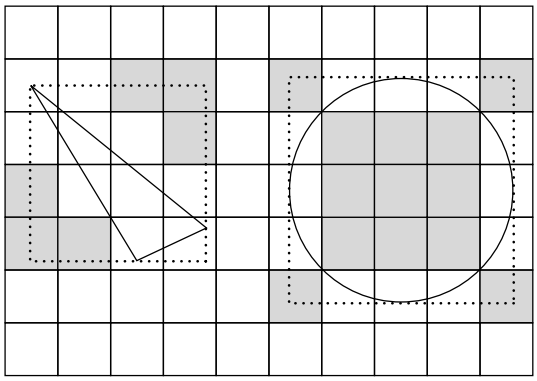

Figure 4.5: Two examples of cases where using the bounding box of a primitive to determine which grid voxels it should be stored in will cause it to be stored in a number of voxels unnecessarily. On the left, a long skinny triangle has a lot of empty space inside its axis-aligned bounding box, and it is unnecessarily added to the shaded voxels. On the right, the surface of the sphere doesn't intersect many of the voxels inside its bound, and they are also inaccurately included in the sphere's extent. While this error degrades performance, it doesn't lead to incorrect ray intersection results.

⟨*GridAccel Private Methods*⟩ +≡ 196
```
inline int offset(int x, int y, int z) const {
    return z*nVoxels[0]*nVoxels[1] + y*nVoxels[0] + x;
}
```

To further reduce memory used for dynamically allocated voxels and to improve their memory locality, the grid constructor uses a `MemoryArena` to hand out memory for voxels. The `MemoryArena`, implemented in Section A.5.4 in Appendix A, provides custom allocation routines based on allocating large blocks of memory and using them to service memory allocation requests. It doesn't support freeing memory from individual allocations; it will only free all of them at once. This improves allocation performance and essentially eliminates memory overhead for bookkeeping, thus reducing the system's overall memory use as well.

⟨*Allocate new voxel and store primitive in it*⟩ ≡ 200
```
voxels[o] = voxelArena.Alloc<Voxel>();
*voxels[o] = Voxel(primitives[i]);
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
MemoryArena voxelArena;
```

GridAccel::nVoxels 198
GridAccel::voxels 199
MemoryArena 1015
Voxel 202
Voxel::AddPrimitive() 202

If this isn't the first primitive to overlap this voxel, the `Voxel` has already been allocated and the primitive is handed off to the `Voxel::AddPrimitive()` method.

⟨*Add primitive to already-allocated voxel*⟩ ≡ 200
```
voxels[o]->AddPrimitive(primitives[i]);
```

The Voxel structure records which primitives overlap its extent using a vector. The Voxel::allCanIntersect member is used to record if all of the primitives in the voxel are intersectable or if some need refinement. It is initially conservatively set to false.

⟨*Voxel Declarations*⟩ ≡
```
struct Voxel {
    ⟨Voxel Public Methods 202⟩
private:
    vector<Reference<Primitive> > primitives;
    bool allCanIntersect;
};
```

When a Voxel is first created, a single Primitive is provided to the constructor.

⟨*Voxel Public Methods*⟩ ≡ 202
```
Voxel(Reference<Primitive> op) {
    allCanIntersect = false;
    primitives.push_back(op);
}
```

⟨*Voxel Public Methods*⟩ +≡ 202
```
void AddPrimitive(Reference<Primitive> prim) {
    primitives.push_back(prim);
}
```

We won't show the straightforward implementations of the GridAccel::WorldBound() or GridAccel::CanIntersect() methods or its destructor.

4.3.2 TRAVERSAL

The GridAccel::Intersect() method handles the task of determining which voxels a ray passes through and calling the appropriate ray–primitive intersection routines.

⟨*GridAccel Method Definitions*⟩ +≡
```
bool GridAccel::Intersect(const Ray &ray, Intersection *isect) const {
    ⟨Check ray against overall grid bounds 203⟩
    ⟨Set up 3D DDA for ray 204⟩
    ⟨Walk ray through voxel grid 205⟩
}
```

The first task is to determine where the ray enters the grid, which gives the starting point for traversal through the voxels. If the ray's origin is inside the grid's bounding box, then clearly it begins there. Otherwise, the GridAccel::Intersect() method finds the intersection of the ray with the grid's bounding box. If the ray hits the bounding box, the first intersection along the ray is the starting point. If the ray misses the grid's bounding box, there can be no intersection with any of the geometry in the grid so GridAccel::Intersect() returns immediately.

GridAccel 196
GridAccel::
CanIntersect() 202
GridAccel::Intersect() 202
GridAccel::WorldBound() 202
Intersection 186
Primitive 185
Ray 66
Reference 1011
Voxel 202
Voxel::allCanIntersect 202
Voxel::primitives 202

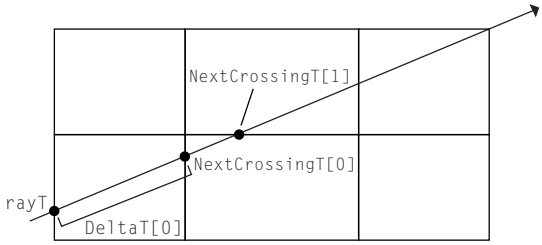

Figure 4.6: Stepping a Ray through a Voxel Grid. `rayT` is the parametric distance along the ray to the first intersection with the grid. The parametric distance along the ray to the point where it crosses into the next voxel in the x direction is stored in `NextCrossingT[0]`, and similarly for the y and z directions (not shown). When the ray crosses into the next x voxel, for example, it is immediately possible to update the value of `NextCrossingT[0]` by adding a fixed value, the voxel width in x divided by the ray's x direction, `DeltaT[0]`.

⟨*Check ray against overall grid bounds*⟩ ≡ 202

```
float rayT;
if (bounds.Inside(ray(ray.mint)))
    rayT = ray.mint;
else if (!bounds.IntersectP(ray, &rayT))
    return false;
Point gridIntersect = ray(rayT);
```

The intersection method next computes the initial (x, y, z) integer voxel coordinates for the ray as well as a number of auxiliary values that will make it efficient to incrementally compute the set of voxels that the ray passes through. The ray–voxel traversal computation is similar in spirit to Bresenham's classic line drawing algorithm, where the series of pixels that a line passes through are found incrementally using just addition and comparisons to step from one pixel to the next. The main difference between the ray marching algorithm and Bresenham's is that we would like to find *all* of the voxels that the ray passes through, while Bresenham's algorithm typically only turns on one pixel per row or column that a line passes through. This type of algorithm is known as a *digital differential analyzer* (DDA).

The values that the ray–voxel stepping algorithm needs to keep track of are the following:

1. The coordinates of the voxel currently being considered, `Pos`.
2. The parametric t position along the ray where it makes its next crossing into another voxel in each of the x, y, and z directions, `NextCrossingT` (Figure 4.6). For example, for a ray with a positive x direction component, the parametric value along the ray where it crosses into the next voxel in x, `NextCrossingT[0]` is the parametric starting point `rayT` plus the x distance to the next voxel divided by the ray's x direction component. (This is similar to the ray–plane intersection formula.)
3. The change in the current voxel coordinates after a step in each direction (1 or −1), stored in `Step`.

BBox::Inside() 72
BBox::IntersectP() 194
Point 63
Ray::mint 67

4. The distance along the ray between voxels in each direction, `DeltaT`. These values are found by dividing the width of a voxel in a particular direction by the ray's corresponding direction component, giving the parametric distance along the ray to travel to get from one side of a voxel to the other in the particular direction.
5. The coordinates of the voxel after the last one the ray passes through when it exits the grid, `Out`.

The first two items will be updated as we step through the grid, while the last three are constant for each ray.

⟨*Set up 3D DDA for ray*⟩ ≡ 202
```
float NextCrossingT[3], DeltaT[3];
int Step[3], Out[3], Pos[3];
for (int axis = 0; axis < 3; ++axis) {
    ⟨Compute current voxel for axis 204⟩
    if (ray.d[axis] >= 0) {
        ⟨Handle ray with positive direction for voxel stepping 204⟩
    }
    else {
        ⟨Handle ray with negative direction for voxel stepping 205⟩
    }
}
```

Computing the voxel address that the ray starts out in is easy since this method has already determined the position where the ray enters the grid. Thus, it can simply use the utility routine `GridAccel::posToVoxel()` defined earlier.

⟨*Compute current voxel for axis*⟩ ≡ 204
```
Pos[axis] = posToVoxel(gridIntersect, axis);
```

If the ray's direction component is zero for a particular axis, then the `NextCrossingT` value for that axis will be initialized to the IEEE floating-point ∞ value by the following computation. The voxel stepping logic later in this section will always decide to step in one of the other directions and will correctly never step in this direction. This is convenient because it can handle rays that are perpendicular to any axis without any special code to test for division by zero.

⟨*Handle ray with positive direction for voxel stepping*⟩ ≡ 204
```
NextCrossingT[axis] = rayT +
    (voxelToPos(Pos[axis]+1, axis) - gridIntersect[axis]) / ray.d[axis];
DeltaT[axis] = width[axis] / ray.d[axis];
Step[axis] = 1;
Out[axis] = nVoxels[axis];
```

GridAccel::nVoxels 198
GridAccel::posToVoxel() 200
GridAccel::voxelToPos() 200
GridAccel::width 199
Ray::d 67

Similar computations compute these values for rays with negative direction components:

⟨*Handle ray with negative direction for voxel stepping*⟩ ≡ 204
```
NextCrossingT[axis] = rayT +
    (voxelToPos(Pos[axis], axis) - gridIntersect[axis]) / ray.d[axis];
DeltaT[axis] = -width[axis] / ray.d[axis];
Step[axis] = -1;
Out[axis] = -1;
```

Once all the preprocessing is done for the ray, stepping through the grid can start. Starting with the first voxel that the ray passes through, the intersection routine checks for intersections with the primitives inside that voxel.

As the ray traverses the grid, it is necessary to handle some issues related to multi-threaded execution. In the `GridAccel` constructor, a reader-writer mutex, `rwMutex`, is created. A reader-writer mutex allows an arbitrary number of threads to request read-only access to shared data. However, if one of the threads wants to modify the data, it must upgrade its hold on the mutex to have write privileges; the `RWMutex` will only allow a single thread to have write privileges and only when no other threads hold read privileges.

⟨*Create reader-writer mutex for grid*⟩ ≡ 197
```
rwMutex = RWMutex::Create();
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
mutable RWMutex *rwMutex;
```

Before traversal starts, a reader lock is acquired from the mutex. If another thread holds a writer lock on the mutex, this thread will stall until the other has released the writer lock, indicating that it has finished updating the grid and it's safe for this thread to continue traversal.

If a hit is found during traversal, the Boolean flag `hitSomething` will be set to `true`. It is necessary to be careful, however, because the found intersection point may be outside the current voxel since primitives may overlap multiple voxels. Therefore, the method doesn't immediately return when done processing a voxel where an intersection was found. Instead, it takes advantage of the fact that the primitive's intersection routine will update the `Ray::maxt` member variable—thus, when stepping through voxels, it will return only when it enters a voxel at a point that is beyond the closest found intersection.

⟨*Walk ray through voxel grid*⟩ ≡ 202
```
RWMutexLock lock(*rwMutex, READ);
bool hitSomething = false;
for (;;) {
    ⟨Check for intersection in current voxel and advance to next 205⟩
}
return hitSomething;
```

⟨*Check for intersection in current voxel and advance to next*⟩ ≡ 205
```
Voxel *voxel = voxels[offset(Pos[0], Pos[1], Pos[2])];
if (voxel != NULL)
    hitSomething |= voxel->Intersect(ray, isect, lock);
⟨Advance to next voxel 207⟩
```

GridAccel 196
GridAccel::offset() 201
GridAccel::rwMutex 205
GridAccel::voxels 199
GridAccel::voxelToPos() 200
GridAccel::width 199
Ray::d 67
Ray::maxt 67
RWMutex 1039
RWMutex::Create() 1039
RWMutexLock 1039
Voxel 202
Voxel::Intersect() 206

For each nonempty voxel, the grid traversal method calls the `Voxel`'s `Intersect()` routine, which handles the details of calling the `Primitive::Intersect()` methods.

⟨*GridAccel Method Definitions*⟩ +≡
```
bool Voxel::Intersect(const Ray &ray, Intersection *isect,
                      RWMutexLock &lock) {
    ⟨Refine primitives in voxel if needed 206⟩
    ⟨Loop over primitives in voxel and find intersections 207⟩
}
```

The Boolean `Voxel::allCanIntersect` member indicates whether all of the primitives in the voxel are known to be intersectable. If its value is `false`, the `Intersect()` routine must loop over all of the primitives, calling their refinement routines as needed until only intersectable geometry remains. If refinement is necessary, we must deal with multi-threaded synchronization: we are going to modify shared data in the grid accelerator, so the read-only lock on the `RWMutex` is upgraded to a writer lock. If other threads are currently holding reader locks, this thread stalls until they have released their reader locks.

Once this thread has a writer lock, it is sure that no other threads are accessing the grid data structures. It is then free to loop over the primitives in the voxel and refine them, modifying the data stored in the voxel. When it is done, it can update `Voxel::allCanIntersect` to be `true`. It then releases its writer lock and continues traversal, holding only a reader lock and thus allowing other threads to access the grid's data.

⟨*Refine primitives in voxel if needed*⟩ ≡ 206
```
if (!allCanIntersect) {
    lock.UpgradeToWrite();
    for (uint32_t i = 0; i < primitives.size(); ++i) {
        Reference<Primitive> &prim = primitives[i];
        ⟨Refine primitive prim if it's not intersectable 207⟩
    }
    allCanIntersect = true;
    lock.DowngradeToRead();
}
```

Primitives that need refinement are refined until only intersectable primitives remain, and a new `GridAccel` is created to hold the returned primitives if more than one was returned. One reason to always make a `GridAccel` for multiple refined primitives is that doing so simplifies primitive refinement. A single `Primitive` always turns into a single object that represents all of the new `Primitives`, so it's never necessary to increase the number of primitives in the voxel. If this primitive overlaps multiple voxels, then because all of them hold a reference to a single `Primitive` for it, it suffices to just update the primitive reference directly, and there's no need to loop over all of the voxels.[3]

GridAccel 196
Intersection 186
Primitive 185
Primitive::Intersect() 186
Ray 66
Reference 1011
RWMutex 1039
RWMutexLock 1039
RWMutexLock::
DowngradeToRead() 1039
RWMutexLock::
UpgradeToWrite() 1039
Voxel 202
Voxel::allCanIntersect 202

3 The bounding box of the original unrefined primitive must encompass the refined geometry as well, so there's no danger that the refined geometry will overlap more voxels than before. On the other hand, it also may overlap many fewer voxels, which would lead to unnecessary intersection tests, since the grid implementation doesn't try to remove references to the primitive from voxels that it no longer overlaps.

⟨*Refine primitive* `prim` *if it's not intersectable*⟩ ≡ 206
```
if (!prim->CanIntersect()) {
    vector<Reference<Primitive> > p;
    prim->FullyRefine(p);
    if (p.size() == 1)
        primitives[i] = p[0];
    else
        primitives[i] = new GridAccel(p, false);
}
```

Once it is certain that there are only intersectable primitives in the voxel, the loop over `Primitives` for performing intersection tests is straightforward.

⟨*Loop over primitives in voxel and find intersections*⟩ ≡ 206
```
bool hitSomething = false;
for (uint32_t i = 0; i < primitives.size(); ++i) {
    Reference<Primitive> &prim = primitives[i];
    if (prim->Intersect(ray, isect))
        hitSomething = true;
}
return hitSomething;
```

After doing the intersection tests for the primitives in the current voxel, it is necessary to step to the next voxel in the ray's path. The grid must decide whether to step in the x, y, or z direction. Fortunately, the `NextCrossingT` variable gives the parametric distance to the next crossing for each direction, and it can choose the smallest one. Traversal can be terminated if this step goes outside of the voxel grid, or if the selected `NextCrossingT` value is beyond the t distance of an already-found intersection. Otherwise, the grid steps to the chosen voxel and increments the chosen direction's `NextCrossingT` by its `DeltaT` value, so that future traversal steps will know how far it is necessary to go before stepping in this direction again.

⟨*Advance to next voxel*⟩ ≡ 205
```
⟨Find stepAxis for stepping to next voxel 208⟩
if (ray.maxt < NextCrossingT[stepAxis])
    break;
Pos[stepAxis] += Step[stepAxis];
if (Pos[stepAxis] == Out[stepAxis])
    break;
NextCrossingT[stepAxis] += DeltaT[stepAxis];
```

Choosing the axis along which to step basically requires finding the smallest of three numbers, a straightforward task. However, in this case an optimization is possible because we don't care about the *value* of the smallest number, just its corresponding index in the `NextCrossingT` array. It is possible to compute this index without any branching, which can lead to performance improvements on modern CPUs, which generally pay a performance penalty for branches.

The following tricky bit of code determines which of the three `NextCrossingT` values is the smallest and sets `stepAxis` accordingly. It encodes this logic by setting each of the

GridAccel 196
Primitive 185
Primitive::CanIntersect() 186
Primitive::FullyRefine() 186
Primitive::Intersect() 186
Reference 1011

three low-order bits in an integer to the results of three comparisons between pairs of `NextCrossingT` values. It then uses a table (`cmpToAxis`) to map the resulting integer to the direction with the smallest value.

⟨*Find* `stepAxis` *for stepping to next voxel*⟩ ≡ 207
```
int bits = ((NextCrossingT[0] < NextCrossingT[1]) << 2) +
           ((NextCrossingT[0] < NextCrossingT[2]) << 1) +
           ((NextCrossingT[1] < NextCrossingT[2]));
const int cmpToAxis[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
int stepAxis = cmpToAxis[bits];
```

The grid also provides a special `GridAccel::IntersectP()` method that is optimized for checking for intersection along shadow rays, where we are only interested in the presence of an intersection and not the details of the intersection itself. It is almost identical to the `GridAccel::Intersect()` routine, except that it calls the `Primitive::IntersectP()` method of the primitives rather than `Primitive::Intersect()`, and it immediately stops traversal when any intersection is found. Because of the small number of differences, we won't include the implementation here.

⟨*GridAccel Public Methods*⟩ ≡ 196
```
bool IntersectP(const Ray &ray) const;
```

4.4 BOUNDING VOLUME HIERARCHIES

Bounding volume hierarchies (BVHs) are an approach for ray intersection acceleration based on primitive subdivision, where the primitives are partitioned into a hierarchy of disjoint sets. (In contrast, spatial subdivision generally partitions space into a hierarchy of disjoint sets.) Figure 4.7 shows a bounding volume hierarchy for a simple scene. Primitives are stored in the leaves and each node stores a bounding box of the primitives in the nodes beneath it. Thus, as a ray traverses through the tree, any time it doesn't intersect a node's bounds, the subtree beneath that node can be skipped.

One property of primitive subdivision is that each primitive appears in the hierarchy only once. In contrast, a primitive may overlap many grid voxels, and thus may be tested for intersection multiple times as the ray passes through them.[4] Another implication of this property is that the amount of memory needed to represent the hierarchy is bounded. For a binary BVH that stores a single primitive in each leaf, the total number of nodes is $2n - 1$, where n is the number of primitives. There will be n leaf nodes and $n - 1$ interior nodes. If leaves store multiple primitives, fewer nodes are needed.

BVHs are generally almost as efficient to build as grids are, while delivering much better performance thanks to being able to better adapt to irregular distributions of primitives in the scene. The kd-trees in the following section generally deliver slightly faster ray intersection calculations than BVHs but take substantially longer to build. On the other

GridAccel::Intersect() 202
GridAccel::IntersectP() 208
Primitive::Intersect() 186
Primitive::IntersectP() 186
Ray 66

4 The *mailboxing* technique can be used to avoid multiple intersections for accelerators that use spatial subdivision, though its implementation can be tricky in the presence of multi-threading. More information on mailboxing is available in the "Further Reading" section.

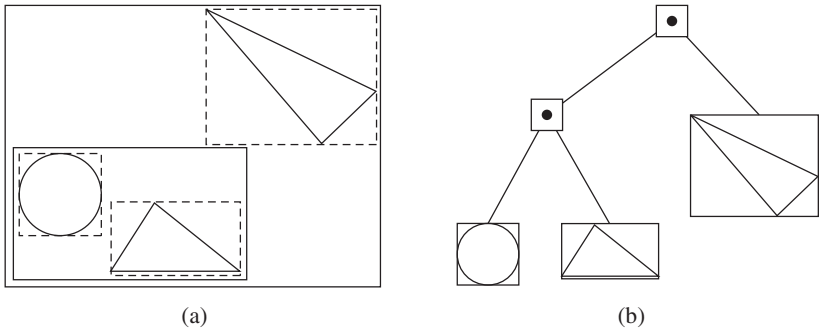

Figure 4.7: Bounding Volume Hierarchy for a Simple Scene. (a) A small collection of primitives, with bounding boxes shown by dashed lines. The primitives are aggregated based on proximity; here, the sphere and the equilateral triangle are bounded by another bounding box before being bounded by a bounding box that encompasses the entire scene (both shown in solid lines). (b) The corresponding bounding volume hierarchy. The root node holds the bounds of the entire scene. Here, it has two children, one storing a bounding box that encompasses the sphere and equilateral triangle (that in turn has those primitives as its children) and the other storing the bounding box that holds the skinny triangle.

hand, BVHs are generally more numerically robust and less prone to subtle round-off bugs than kd-trees are.

The BVH accelerator, `BVHAccel`, is defined in `accelerators/bvh.h` and `accelerators/bvh.cpp`. In addition to the primitives to be stored and the maximum number of primitives that can be in any leaf node, its constructor takes a string that describes which of three algorithms to use when partitioning primitives to build the tree. The default, "sah," indicates that an algorithm based on the "surface area heuristic" (discussed in Section 4.4.2) should be used. The other two approaches take slightly less computation when building the tree but create trees that are typically less efficient when used for ray intersections.

⟨*BVHAccel Method Definitions*⟩ ≡

```
BVHAccel::BVHAccel(const vector<Reference<Primitive> > &p,
                   uint32_t mp, const string &sm) {
    maxPrimsInNode = min(255u, mp);
    for (uint32_t i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(primitives);
    if (sm == "sah")         splitMethod = SPLIT_SAH;
    else if (sm == "middle") splitMethod = SPLIT_MIDDLE;
    else if (sm == "equal")  splitMethod = SPLIT_EQUAL_COUNTS;
    else {
        Warning("BVH split method \"%s\" unknown.  Using \"sah\".",
                sm.c_str());
        splitMethod = SPLIT_SAH;
    }
```

BVHAccel 209
BVHAccel::splitMethod 210
Primitive 185
Primitive::FullyRefine() 186
Reference 1011

```
        if (primitives.size() == 0) {
            nodes = NULL;
            return;
        }
        ⟨Build BVH from primitives 210⟩
    }
```

⟨*BVHAccel Private Data*⟩ ≡

```
    uint32_t maxPrimsInNode;
    enum SplitMethod { SPLIT_MIDDLE, SPLIT_EQUAL_COUNTS, SPLIT_SAH };
    SplitMethod splitMethod;
    vector<Reference<Primitive> > primitives;
```

4.4.1 BVH CONSTRUCTION

There are three stages to BVH construction. First, bounding information about each primitive is computed and stored in an array that will be used during tree construction. Next, the tree is built via a procedure that splits the primitives into subsets and recursively builds BVHs for the subsets. The result is a binary tree where each interior node holds pointers to its children and each leaf node holds references to one or more primitives. Finally, this tree is converted to a more compact (and thus more efficient) pointerless representation for use during rendering. (The implementation is more straightforward with this approach, versus computing the pointerless representation directly during construction, which is also possible.)

⟨*Build BVH from* `primitives`⟩ ≡ 209

```
    ⟨Initialize buildData array for primitives 210⟩
    ⟨Recursively build BVH tree for primitives 211⟩
    ⟨Compute representation of depth-first traversal of BVH tree 223⟩
```

For each primitive to be stored in the BVH, we store the centroid of its bounding box, its complete bounding box, and its index in the `primitives` array in an instance of the `BVHPrimitiveInfo` structure. As the tree is built, the `buildData` array will be recursively sorted and partitioned to place the primitives into groups that are spatially close to each other.

⟨*Initialize* `buildData` *array for primitives*⟩ ≡ 210

```
    vector<BVHPrimitiveInfo> buildData;
    buildData.reserve(primitives.size());
    for (uint32_t i = 0; i < primitives.size(); ++i) {
        BBox bbox = primitives[i]->WorldBound();
        buildData.push_back(BVHPrimitiveInfo(i, bbox));
    }
```

BBox 70
BVHAccel::primitives 210
BVHPrimitiveInfo 211
Primitive 185
Primitive::WorldBound() 185
Reference 1011

⟨*BVHAccel Local Declarations*⟩ ≡
```
struct BVHPrimitiveInfo {
    BVHPrimitiveInfo(int pn, const BBox &b)
        : primitiveNumber(pn), bounds(b) {
        centroid = .5f * b.pMin + .5f * b.pMax;
    }
    int primitiveNumber;
    Point centroid;
    BBox bounds;
};
```

The initial call to `recursiveBuild()` is given all of the primitives to be stored in the tree. It returns a pointer to the root of the tree, which is represented with the `BVHBuildNode` structure. The code here uses a `MemoryArena` to allocate nodes one at a time.

One important side-effect of the tree construction process is that a new array of primitives is returned via the `orderedPrims` parameter; this array stores the primitives ordered so that the primitives in leaf nodes occupy contiguous ranges in the array. It is swapped with the original `primitives` array after tree construction.

⟨*Recursively build BVH tree for primitives*⟩ ≡ 210
```
MemoryArena buildArena;
uint32_t totalNodes = 0;
vector<Reference<Primitive> > orderedPrims;
orderedPrims.reserve(primitives.size());
BVHBuildNode *root = recursiveBuild(buildArena, buildData, 0,
                                    primitives.size(), &totalNodes,
                                    orderedPrims);
primitives.swap(orderedPrims);
```

Each `BVHBuildNode` represents a node of the BVH. All nodes store a `BBox`, which stores the bounds of all of the children beneath the node. Each interior node stores pointers to its two children in `children`. Interior nodes also record the coordinate axis along which primitives were sorted for distribution to their two children; this information is used to improve the performance of the traversal algorithm. Leaf nodes need to record which primitive or primitives are stored in them; the elements of the `BVHAccel::primitives` array from the offset `firstPrimOffset` up to but not including `firstPrimOffset + nPrimitives` are the primitives in the leaf. (Hence the need for reordering the primitives array, so that this representation can be used, rather than, for example, storing a variable-sized array of primitive indices at each leaf node.)

⟨*BVHAccel Local Declarations*⟩ +≡
```
struct BVHBuildNode {
    ⟨BVHBuildNode Public Methods 212⟩
    BBox bounds;
    BVHBuildNode *children[2];
    uint32_t splitAxis, firstPrimOffset, nPrimitives;
};
```

BBox 70
BVHAccel::primitives 210
BVHAccel::
recursiveBuild() 213
BVHBuildNode 211
BVHPrimitiveInfo 211
MemoryArena 1015
Point 63
Primitive 185
Reference 1011

The BVHBuildNode constructor only initializes the children pointers; we'll distinguish between leaf and interior nodes by whether their children pointers are NULL or not, respectively.

⟨*BVHBuildNode Public Methods*⟩ ≡ 211
```
BVHBuildNode() { children[0] = children[1] = NULL; }
```

⟨*BVHBuildNode Public Methods*⟩ +≡ 211
```
void InitLeaf(uint32_t first, uint32_t n, const BBox &b) {
    firstPrimOffset = first;
    nPrimitives = n;
    bounds = b;
}
```

The InitInterior() method requires that the two children nodes already have been created, so that their pointers can be passed in. This requirement makes it easy to compute the bounds of the interior node, since the children bounds are immediately available.

⟨*BVHBuildNode Public Methods*⟩ +≡ 211
```
void InitInterior(uint32_t axis, BVHBuildNode *c0, BVHBuildNode *c1) {
    children[0] = c0;
    children[1] = c1;
    bounds = Union(c0->bounds, c1->bounds);
    splitAxis = axis;
    nPrimitives = 0;
}
```

In addition to the MemoryArena used for node allocation and the array of BVHPrimitiveInfo structures, recursiveBuild() takes as parameters the range [start, end). It is responsible for returning a BVH for the subset of primitives represented by the range from buildData[start] up to and including buildData[end-1]. If this represents only a single primitive, the recursion has bottomed out and a leaf node is created. Otherwise, this method partitions the elements of the array in that range using one of a few partitioning algorithms and reorders the array elements in the range accordingly, so that the ranges from [start, mid) and [mid, end) represent the partitioned subsets. If the partitioning is successful, these two primitive sets are in turn passed to recursive calls that will themselves return pointers to nodes for the two children of the current node.

totalNodes tracks the total number of BVH nodes that have been created; this number is used so that exactly the right number of the more compact LinearBVHNodes can be allocated later. Finally, the orderedPrims array is used to store primitive references as primitives are stored in leaf nodes of the tree. This array is initially empty; when a leaf node is created, it adds the primitives that overlap it to the end of the array, making it possible for leaf nodes to just store an offset into this array and a primitive count to represent the set of primitives that overlap it. Recall that when tree construction is finished, BVHAccel::primitives is replaced with the ordered primitives array created here.

BBox 70
BVHAccel::primitives 210
BVHBuildNode 211
BVHBuildNode::bounds 211
BVHBuildNode::children 211
BVHBuildNode::
firstPrimOffset 211
BVHBuildNode::
nPrimitives 211
BVHBuildNode::splitAxis 211
BVHPrimitiveInfo 211
LinearBVHNode 222
MemoryArena 1015
Union() 72

⟨*BVHAccel Method Definitions*⟩ +≡
```
BVHBuildNode *BVHAccel::recursiveBuild(MemoryArena &buildArena,
        vector<BVHPrimitiveInfo> &buildData, uint32_t start,
        uint32_t end, uint32_t *totalNodes,
        vector<Reference<Primitive> > &orderedPrims) {
    (*totalNodes)++;
    BVHBuildNode *node = buildArena.Alloc<BVHBuildNode>();
    ⟨Compute bounds of all primitives in BVH node 213⟩
    uint32_t nPrimitives = end - start;
    if (nPrimitives == 1) {
        ⟨Create leaf BVHBuildNode 213⟩
    }
    else {
        ⟨Compute bound of primitive centroids, choose split dimension dim 214⟩
        ⟨Partition primitives into two sets and build children 215⟩
    }
    return node;
}
```

⟨*Compute bounds of all primitives in BVH node*⟩ ≡ 213
```
BBox bbox;
for (uint32_t i = start; i < end; ++i)
    bbox = Union(bbox, buildData[i].bounds);
```

At leaf nodes, the primitives overlapping the leaf are appended to the `orderedPrims` array and a leaf node object is initialized.

⟨*Create leaf* `BVHBuildNode`⟩ ≡ 213, 215, 221
```
uint32_t firstPrimOffset = orderedPrims.size();
for (uint32_t i = start; i < end; ++i) {
    uint32_t primNum = buildData[i].primitiveNumber;
    orderedPrims.push_back(primitives[primNum]);
}
node->InitLeaf(firstPrimOffset, nPrimitives, bbox);
```

For interior nodes, the collection of primitives must be partitioned between the two children subtrees. Given n primitives, there are in general $2^n - 2$ possible ways to partition them into two nonempty groups. In practice when building BVHs, one generally considers partitions along a coordinate axis, meaning that there are about $6n$ candidate partitions. (Along each axis, each primitive may be put into the first partition or the second partition.)

Here, we choose one of the three coordinate axes to use in partitioning the primitives. We select the axis with the greatest variation of bounding box centroids for the current set of primitives. (An alternative would be to try all three axes and select the one that gave the

BBox 70
BVHBuildNode 211
BVHBuildNode::InitLeaf() 212
BVHPrimitiveInfo 211
BVHPrimitiveInfo::bounds 211
BVHPrimitiveInfo::
primitiveNumber 211
MemoryArena 1015
Primitive 185
Reference 1011

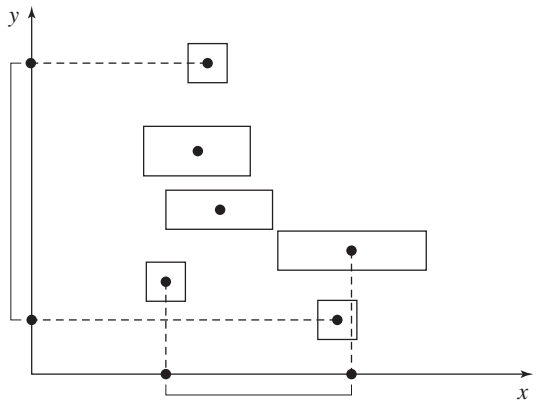

Figure 4.8: Choosing the Axis Along Which to Partition Primitives. The `BVHAccel` chooses an axis along which to partition the primitives based on which axis has the largest range of the centroids of the primitives' bounding boxes. Here, in two dimensions, their extent is largest along the y axis (filled points on the axes), so the primitives will be partitioned in y.

best result, but in practice this approach works well.) This approach gives good partitions in many reasonable scenes; Figure 4.8 illustrates the strategy.

The general goal in partitioning here is to select a partition of primitives that doesn't have too much overlap of the bounding boxes of the two resulting primitive sets—if there is substantial overlap then it will more frequently be necessary to traverse both children subtrees when traversing the tree, requiring more computation than if it had been possible to more effectively prune away collections of primitives. This idea of finding effective primitive partitions will be made more rigorous shortly, in the discussion of the surface area heuristic.

⟨*Compute bound of primitive centroids, choose split dimension* `dim`⟩ ≡ 213

```
BBox centroidBounds;
for (uint32_t i = start; i < end; ++i)
    centroidBounds = Union(centroidBounds, buildData[i].centroid);
int dim = centroidBounds.MaximumExtent();
```

If all of the centroid points are at the same position (i.e., the centroid bounds have zero volume), then recursion stops and a leaf node is created with the primitives; none of the splitting methods here is effective in that (unusual) case. In the usual case, the primitives are partitioned using the chosen method and passed to two recursive calls to `recursiveBuild()`.

BBox 70
BBox::MaximumExtent() 73
BVHPrimitiveInfo::
centroid 211
Union() 72

⟨*Partition primitives into two sets and build children*⟩ ≡ 213

```
uint32_t mid = (start + end) / 2;
if (centroidBounds.pMax[dim] == centroidBounds.pMin[dim]) {
    ⟨Create leaf BVHBuildNode 213⟩
    return node;
}
⟨Partition primitives based on splitMethod⟩
node->InitInterior(dim,
                   recursiveBuild(buildArena, buildData, start, mid,
                                  totalNodes, orderedPrims),
                   recursiveBuild(buildArena, buildData, mid, end,
                                  totalNodes, orderedPrims));
```

We won't include the code fragment ⟨*Partition primitives based on* `splitMethod`⟩ here; it just uses the value of `BVHAccel::splitMethod` to determine which primitive partitioning scheme to use. These three schemes will be described in the following few pages.

A simple `splitMethod` is `SPLIT_MIDDLE`, which first computes the midpoint of the primitives' centroids along the splitting axis. This method is implemented in the fragment ⟨*Partition primitives through node's midpoint*⟩. The primitives are classified into the two sets, depending on whether their centroids are above or below the midpoint. This partitioning is easily done with the `std::partition()` C++ standard library function, which takes a range of elements in an array and a comparison function and orders the elements in the array so that all of the elements that return `true` for the given predicate function appear in the range before those that return `false` for it.[5] It returns a pointer to the first element that had a `false` value for the predicate, which is converted into an offset into the `buildData` array so that we can pass it to the recursive call. Figure 4.9 illustrates this method, including cases where it does and does not work well.

⟨*Partition primitives through node's midpoint*⟩ ≡

```
float pmid = .5f * (centroidBounds.pMin[dim] + centroidBounds.pMax[dim]);
BVHPrimitiveInfo *midPtr = std::partition(&buildData[start],
                                          &buildData[end-1]+1,
                                          CompareToMid(dim, pmid));
mid = midPtr - &buildData[0];
```

The `CompareToMid` predicate returns `true` if the given primitive's bound's centroid is below the given midpoint.

BBox::pMax 71
BBox::pMin 71
BVHAccel::recursiveBuild() 213
BVHAccel::splitMethod 210
BVHBuildNode::InitInterior() 212
BVHPrimitiveInfo 211

5 Note the unusual expression of the indexing of the `buildData` array, `&buildData[end-1]+1`. The code is written in this way for somewhat obscure reasons. In the C programming language, it is legal to compute the pointer one element past the end of an array so that iteration over array elements can continue until the current pointer is equal to the end point. To that end, we would like to just write the expression `&buildData[end]` here. However, buildData was allocated as a C++ `vector`; some `vector` implementations issue a run-time error of the offset passed to their `[]` operator is past the end of the array. Because we're not trying to reference the value of the element one past the end of the array but just compute its address, this operation is in fact safe. Therefore, we compute the same address in the end with the expression used here, while also satisfying any `vector` error checking.

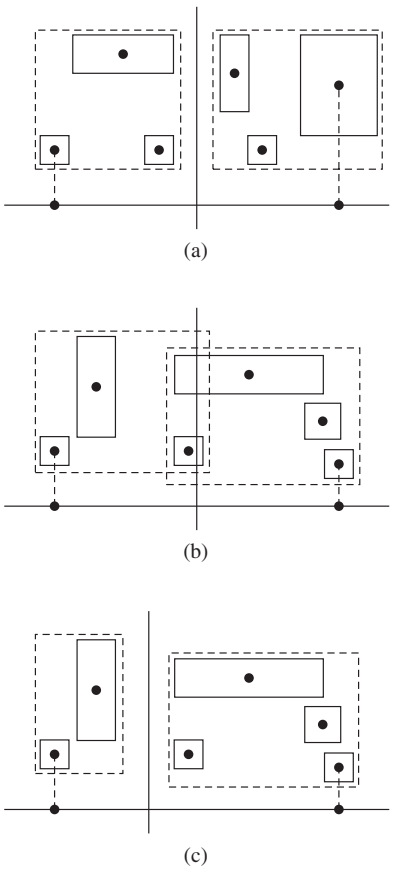

Figure 4.9: Splitting Primitives Based on the Midpoint of Centroids on an Axis. (a) For some distributions of primitives, such as the one shown here, splitting based on the midpoint of the centroids along the chosen axis works well. (The bounding boxes of the two resulting primitive groups are shown with dashed lines.) (b) For distributions like this one, the midpoint is a suboptimal choice; the two resulting bounding boxes overlap substantially. (c) If the same group of primitives from (b) is instead split along the line shown here, the resulting bounding boxes are smaller and don't overlap at all, leading to better performance when rendering.

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct CompareToMid {
    CompareToMid(int d, float m) { dim = d; mid = m; }
    int dim;
    float mid;
    bool operator()(const BVHPrimitiveInfo &a) const {
        return a.centroid[dim] < mid;
    }
};
```

BVHPrimitiveInfo 211
BVHPrimitiveInfo::
centroid 211

Another straightforward partitioning scheme is used when `splitMethod` is `SPLIT_EQUAL_COUNTS`; it is implemented in ⟨*Partition primitives into equally-sized subsets*⟩. It partitions the primitives into two equal-sized subsets such that the first half of the n of them are the $n/2$ with smallest centroid coordinate values along the chosen axis and the second half are the ones with the largest centroid coordinate values. While this approach can sometimes work well, the case in Figure 4.9(b) is one where this method also fares poorly.

This scheme is also easily implemented with a standard library call, `std::nth_element()`. It takes a start, middle, and ending pointer as well as a comparison function. It orders the array so that the element at the middle pointer is the one that would be there if the array was fully sorted, and such that all of the elements before the middle one compare to less than the middle element and all of the elements after it compare to greater than it. This ordering can be done in $O(n)$ time, with n the number of elements, which is more efficient than the $O(n \log n)$ of completely sorting the array.

⟨*Partition primitives into equally-sized subsets*⟩ ≡ 219

```
mid = (start + end) / 2;
std::nth_element(&buildData[start], &buildData[mid],
                 &buildData[end-1]+1, ComparePoints(dim));
```

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct ComparePoints {
    ComparePoints(int d) { dim = d; }
    int dim;
    bool operator()(const BVHPrimitiveInfo &a,
                    const BVHPrimitiveInfo &b) const {
        return a.centroid[dim] < b.centroid[dim];
    }
};
```

4.4.2 THE SURFACE AREA HEURISTIC

The two primitive partitioning approaches above can work well for some distributions of primitives, but they often choose partitions that perform poorly in practice, leading to more nodes of the tree being visited by rays and hence unnecessarily inefficient ray–primitive intersection computations at rendering time. Most of the best current algorithms for building acceleration structures for ray-tracing are based on the "surface area heuristic" (SAH), which provides a well-grounded cost model for answering questions like "which of a number of partitions of primitives will lead to a better BVH for ray–primitive intersection tests?," or "which of a number of possible positions to split space in a spatial subdivision scheme will lead to a better acceleration structure?"

The SAH model estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray–primitive intersection tests for a particular partitioning of primitives. Algorithms for building acceleration structures can then follow the goal of minimizing total cost. Typically, a greedy algorithm is used that minimizes the cost for each single node of the hierarchy being built individually.

BVHPrimitiveInfo 211

BVHPrimitiveInfo::
centroid 211

The ideas behind the SAH cost model are straightforward: at any point in building an adaptive acceleration structure (primitive subdivision or spatial subdivision), we could just create a leaf node for the current region and geometry. In that case, any ray that passes through this region will be tested against all of the overlapping primitives and will incur a cost of

$$\sum_{i=1}^{N} t_{\text{isect}}(i),$$

where N is the number of primitives and $t_{\text{isect}}(i)$ is the time to compute a ray–object intersection with the ith primitive.

The other option is to split the region. In that case, rays will incur the cost

$$c(A, B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i), \tag{4.1}$$

where t_{trav} is the time it takes to traverse the interior node and determine which of the children the ray passes through, p_A and p_B are the probabilities that the ray passes through each of the child nodes (assuming binary subdivision), a_i and b_i are the indices of primitives in the two children nodes, and N_A and N_B are the number of primitives that overlap the regions of the two child nodes, respectively. The choice of how primitives are partitioned affects both the values of the two probabilities as well as the set of primitives on each side of the split.

In `pbrt`, we will make the simplifying assumption that $t_{\text{isect}}(i)$ is the same for all of the primitives; this assumption is probably not too far from reality, and any error that it introduces doesn't seem to affect the performance of accelerators very much. Another possibility would be to add a method to `Primitive` that returns an estimate of the number of CPU cycles its intersection test requires.

The probabilities p_A and p_B can be computed using ideas from geometric probability. It can be shown that for a convex volume A contained in another convex volume B, the conditional probability that a random ray passing through B will also pass through A is the ratio of their surface areas, s_A and s_B:

$$p(A|B) = \frac{s_A}{s_B}.$$

Because we are interested in the cost for rays passing through the node, we can use this result directly. Thus, if we are considering refining a region of space space A such that there are two new subregions with bounds B and C (Figure 4.10), the probability that a ray passing through A will also pass through either of the subregions is easily computed.

When `splitMethod` has the value `SPLIT_SAH`, the SAH is used for building the BVH, choosing a partition of the primitives along the chosen axis that gives a minimal SAH cost estimate by considering a number of candidate partitions. This is the default, and it creates the most efficient trees for rendering. However, once we have refined down to a

Primitive 185

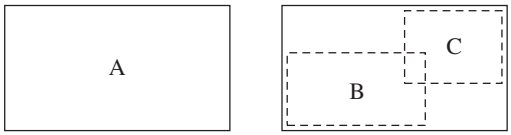

Figure 4.10: If a node of the bounding hierarchy with surface area s_A is split into two children with surface areas s_B and s_C, the probabilities that a ray passing through A also passes through B and C are given by s_B/s_A and s_C/s_A, respectively. Note that $s_B + s_C > s_A$, unless one of them is empty.

small handful of primitives, we switch over to partitioning into equally sized subsets. The incremental computational cost for applying the SAH at this point isn't worthwhile.

⟨*Partition primitives using approximate SAH*⟩ ≡
```
if (nPrimitives <= 4) {
    ⟨Partition primitives into equally-sized subsets 217⟩
}
else {
    ⟨Allocate BucketInfo for SAH partition buckets 219⟩
    ⟨Initialize BucketInfo for SAH partition buckets 220⟩
    ⟨Compute costs for splitting after each bucket 221⟩
    ⟨Find bucket to split at that minimizes SAH metric 221⟩
    ⟨Either create leaf or split primitives at selected SAH bucket 221⟩
}
```

Rather than exhaustively considering all $2n$ possible partitions along the axis, computing the SAH for each to select the best, the implementation here instead divides the range along the axis into a small number of buckets of equal extent. It then only considers partitions at bucket boundaries. This approach is more efficient than considering all partitions while usually still producing partitions that are nearly as effective. This idea is illustrated in Figure 4.11.

⟨*Allocate* BucketInfo *for SAH partition buckets*⟩ ≡ 219
```
const int nBuckets = 12;
struct BucketInfo {
    BucketInfo() { count = 0; }
    int count;
    BBox bounds;
};
BucketInfo buckets[nBuckets];
```

For each primitive in the range, we determine the bucket that its centroid lies in and update the bucket's bounds to include the primitive's bounds.

BBox 70

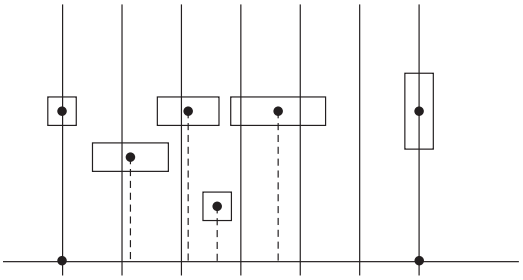

Figure 4.11: Choosing a Splitting Plane with the Surface Area Heuristic for BVHs. The projected extent of primitive bounds centroids is projected onto the chosen split axis. Each primitive is placed in a bucket along the axis based on the centroid of its bounds. The implementation then estimates the cost for splitting the primitives along the planes along each of the bucket boundaries (solid lines); whichever one gives the minimum cost per the surface area heuristic is selected.

⟨*Initialize* `BucketInfo` *for SAH partition buckets*⟩ ≡ 219

```
for (uint32_t i = start; i < end; ++i) {
    int b = nBuckets *
        ((buildData[i].centroid[dim] - centroidBounds.pMin[dim]) /
         (centroidBounds.pMax[dim] - centroidBounds.pMin[dim]));
    if (b == nBuckets) b = nBuckets-1;
    buckets[b].count++;
    buckets[b].bounds = Union(buckets[b].bounds, buildData[i].bounds);
}
```

For each bucket, we now have a count of the number of primitives and the bounds of all of their respective bounding boxes. We want to use the SAH to estimate the cost of splitting at each of the bucket boundaries. The fragment below loops over all of the buckets and initializes the `cost[i]` array to store the estimated SAH cost for splitting after the `i`th bucket. (It doesn't consider a split after the last bucket, which by definition wouldn't split the primitives.)

We arbitrarily set the estimated intersection cost to one, and then set the estimated traversal cost to 1/8. (One of the two of them can always be set to one since it is the relative, rather than absolute, magnitudes of the estimated traversal and intersection costs that determines their effect.) While the absolute amount of computation for node traversal—a ray–bounding box intersection—is only slightly less than the amount of computation needed to intersect a ray with a shape, ray–primitive intersection tests in `pbrt` go through two virtual function calls, which add significant overhead, so we estimate their cost here as eight times more than the ray–box intersection.

This computation has $O(n^2)$ complexity in the number of buckets, though a linear-time implementation based on a forward scan over the buckets and a backward scan over the buckets that incrementally compute and store bounds and counts is possible. For the small n here, the performance impact is generally acceptable, though for a more highly optimized renderer addressing this inefficiency may be worthwhile.

`BucketInfo::bounds` 219
`BucketInfo::count` 219
`BVHPrimitiveInfo::centroid` 211

⟨*Compute costs for splitting after each bucket*⟩ ≡ 219
```
float cost[nBuckets-1];
for (int i = 0; i < nBuckets-1; ++i) {
    BBox b0, b1;
    int count0 = 0, count1 = 0;
    for (int j = 0; j <= i; ++j) {
        b0 = Union(b0, buckets[j].bounds);
        count0 += buckets[j].count;
    }
    for (int j = i+1; j < nBuckets; ++j) {
        b1 = Union(b1, buckets[j].bounds);
        count1 += buckets[j].count;
    }
    cost[i] = .125f + (count0*b0.SurfaceArea() + count1*b1.SurfaceArea()) /
              bbox.SurfaceArea();
}
```

Given all of the costs, a linear scan through the `cost` array finds the partition with minimum cost.

⟨*Find bucket to split at that minimizes SAH metric*⟩ ≡ 219
```
float minCost = cost[0];
uint32_t minCostSplit = 0;
for (int i = 1; i < nBuckets-1; ++i) {
    if (cost[i] < minCost) {
        minCost = cost[i];
        minCostSplit = i;
    }
}
```

If the found bucket boundary for partitioning has a lower estimated cost than building a node with the existing primitives or if more than the maximum number of primitives allowed in a node is present, the `std::partition()` function is used to do the work of reordering nodes in the `buildData` array. Recall from its usage above that this function ensures that all elements of the array that return `true` from the given predicate appear before those that return `false`, and that it returns a pointer to the first element where the predicate returns `false`. Because we arbitrarily set the estimated intersection cost to one previously, the estimated cost for just creating a leaf node is equal to the number of primitives, `nPrimitives`.

BBox 70
BBox::SurfaceArea() 72
BucketInfo::bounds 219
BucketInfo::count 219
BVHAccel::maxPrimsInNode 210
BVHPrimitiveInfo 211
Union() 72

⟨*Either create leaf or split primitives at selected SAH bucket*⟩ ≡ 219
```
if (nPrimitives > maxPrimsInNode ||
    minCost < nPrimitives) {
    BVHPrimitiveInfo *pmid = std::partition(&buildData[start],
        &buildData[end-1]+1,
        CompareToBucket(minCostSplit, nBuckets, dim, centroidBounds));
    mid = pmid - &buildData[0];
}
```

```
else {
    ⟨Create leaf BVHBuildNode 213⟩
}
```

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct CompareToBucket {
    CompareToBucket(int split, int num, int d, const BBox &b)
        : centroidBounds(b)
    { splitBucket = split; nBuckets = num; dim = d; }
    bool operator()(const BVHPrimitiveInfo &p) const;

    int splitBucket, nBuckets, dim;
    const BBox &centroidBounds;
};
```

Classifying a primitive involves recomputing which bucket it maps to and classifying the bucket with respect to the bucket split boundary.

⟨*BVHAccel Local Declarations*⟩ +≡

```
bool CompareToBucket::operator()(const BVHPrimitiveInfo &p) const {
    int b = nBuckets * ((p.centroid[dim] - centroidBounds.pMin[dim]) /
            (centroidBounds.pMax[dim] - centroidBounds.pMin[dim]));
    if (b == nBuckets) b = nBuckets-1;
    return b <= splitBucket;
}
```

4.4.3 COMPACT BVH FOR TRAVERSAL

Once the BVH tree is built, the last step is to convert it into a compact representation—doing so improves cache, memory, and thus overall system performance. The final BVH is stored in a linear array in memory. The nodes of the original tree are laid out in depth-first order, which means that the first child of each interior node is immediately after the node in memory. The offset to the second child of each interior node is then stored explicitly. See Figure 4.12 for an illustration of the relationship between tree topology and node order in memory.

The `LinearBVHNode` structure stores the information needed to traverse the BVH. In addition to the bounding box for each node, for leaf nodes it stores the offset and primitive count for the primitives in the node. For interior nodes, it stores the offset to the second child as well as which of the coordinate axes the primitives were partitioned along when the hierarchy was built; this information is used in the traversal routine below to try to visit nodes in front-to-back order along the ray.

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct LinearBVHNode {
    BBox bounds;
    union {
        uint32_t primitivesOffset;    // leaf
        uint32_t secondChildOffset;   // interior
    };
```

BBox 70
BVHPrimitiveInfo 211
LinearBVHNode 222

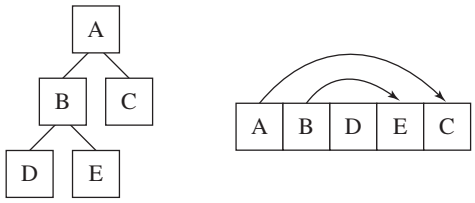

Figure 4.12: Linear Layout of a BVH in Memory. The nodes of the BVH (left) are stored in memory in depth-first order (right). Therefore, for any interior node of the tree (A and B in this example), the first child is found immediately after the parent node in memory. The second child is found via an offset pointer, represented here with lines with arrows. Leaf nodes of the tree (D, E, and C) have no children.

```
    uint8_t nPrimitives;  // 0 -> interior node
    uint8_t axis;         // interior node: xyz
    uint8_t pad[2];       // ensure 32 byte total size
};
```

This structure is padded to ensure that it's 32 bytes large. Doing so ensures that, if the nodes are allocated such that the first node is cache-line aligned, then none of the subsequent nodes will straddle cache lines (as long as the cache line size is at least 32 bytes, which is the case on modern CPU architectures).

⟨*BVHAccel Private Data*⟩ +≡
```
LinearBVHNode *nodes;
```

The built tree is transformed to the `LinearBVHNode` representation by the `flattenBVHTree()` method, which performs a depth-first traversal and stores the nodes in memory in linear order.

⟨*Compute representation of depth-first traversal of BVH tree*⟩ ≡ 210
```
nodes = AllocAligned<LinearBVHNode>(totalNodes);
for (uint32_t i = 0; i < totalNodes; ++i)
    new (&nodes[i]) LinearBVHNode;
uint32_t offset = 0;
flattenBVHTree(root, &offset);
```

BVHAccel::
flattenBVHTree() 224
BVHAccel::nodes 223
LinearBVHNode 222

Flattening the tree to the linear representation is straightforward; the `*offset` parameter tracks the current offset into the `BVHAccel::nodes` array. Note that the current node is added to the array before the recursive calls to process its children (if the node is an interior node).

⟨*BVHAccel Method Definitions*⟩ +≡
```
uint32_t BVHAccel::flattenBVHTree(BVHBuildNode *node, uint32_t *offset) {
    LinearBVHNode *linearNode = &nodes[*offset];
    linearNode->bounds = node->bounds;
    uint32_t myOffset = (*offset)++;
    if (node->nPrimitives > 0) {
        linearNode->primitivesOffset = node->firstPrimOffset;
        linearNode->nPrimitives = node->nPrimitives;
    }
    else {
        ⟨Creater interior flattened BVH node 224⟩
    }
    return myOffset;
}
```

At interior nodes, recursive calls are made to flatten the two subtrees. The first one ends up immediately after the current node in the array, as desired, and the offset of the second one, returned by its recursive `flattenBVHTree()` call, is stored in this node's `secondChildOffset` member.

⟨*Creater interior flattened BVH node*⟩ ≡ 224
```
linearNode->axis = node->splitAxis;
linearNode->nPrimitives = 0;
flattenBVHTree(node->children[0], offset);
linearNode->secondChildOffset = flattenBVHTree(node->children[1],
                                               offset);
```

4.4.4 TRAVERSAL

The BVH traversal code is quite simple—there are no recursive function calls and only a tiny amount of data to maintain about the current state of the traversal. The `Intersect()` method starts by precomputing a few values related to the ray that will be used repeatedly.

⟨*BVHAccel Method Definitions*⟩ +≡
```
bool BVHAccel::Intersect(const Ray &ray, Intersection *isect) const {
    if (!nodes) return false;
    bool hit = false;
    Point origin = ray(ray.mint);
    Vector invDir(1.f / ray.d.x, 1.f / ray.d.y, 1.f / ray.d.z);
    uint32_t dirIsNeg[3] = { invDir.x < 0, invDir.y < 0, invDir.z < 0 };
    ⟨Follow ray through BVH nodes to find primitive intersections 225⟩
    return hit;
}
```

Each time the `while` loop in `Intersect()` starts an iteration, `nodeNum` holds the offset into the `nodes` array of the node to be visited. It starts with a value of zero, representing the root of the tree. The nodes that still need to be visited are stored in the `todo[]` array, which acts as a stack; `todoOffset` holds the offset to the next free element in the stack.

BVHAccel 209
BVHAccel::flattenBVHTree() 224
BVHAccel::nodes 223
BVHBuildNode 211
BVHBuildNode::bounds 211
BVHBuildNode::children 211
BVHBuildNode::firstPrimOffset 211
BVHBuildNode::nPrimitives 211
BVHBuildNode::splitAxis 211
Intersection 186
LinearBVHNode 222
LinearBVHNode::axis 222
LinearBVHNode::bounds 222
LinearBVHNode::nPrimitives 222
LinearBVHNode::primitivesOffset 222
LinearBVHNode::secondChildOffset 222
Point 63
Ray 66
Ray::mint 67
Vector 57

⟨*Follow ray through BVH nodes to find primitive intersections*⟩ ≡ 224
```
uint32_t todoOffset = 0, nodeNum = 0;
uint32_t todo[64];
while (true) {
    const LinearBVHNode *node = &nodes[nodeNum];
    ⟨Check ray against BVH node 225⟩
}
```

At each node, we check to see if the ray intersects the node's bounding box (or starts inside of it). We visit the node if so, testing for intersection with its primitives if it's a leaf node or processing its children if it's an interior node. If no intersection is found, then the offset of the next node to be visited is retrieved from `todo[]` (or, traversal is complete if the stack is empty).

⟨*Check ray against BVH node*⟩ ≡ 225
```
if (::IntersectP(node->bounds, ray, invDir, dirIsNeg)) {
    if (node->nPrimitives > 0) {
        ⟨Intersect ray with primitives in leaf BVH node 226⟩
    }
    else {
        ⟨Put far BVH node on todo stack, advance to near node 227⟩
    }
}
else {
    if (todoOffset == 0) break;
    nodeNum = todo[--todoOffset];
}
```

`BVHAccel` uses a specialized `IntersectP()` function for checking for intersection of rays with bounding boxes. It takes a direction vector that stores the reciprocal of the actual direction, thus changing three divides to multiplies in the slab intersection tests, as well as precomputed values that indicate whether each direction component is negative. The implementation is based on the approach presented by Williams et al. (2005), which performs three main optimizations to the basic `BBox::IntersectP()` routine:

- The `for` loop is unrolled, with the three tests handled directly.
- The reciprocals of the direction components are precomputed and passed in, making it possible to reuse the results of the divisions across all of the bounding box intersection tests done for the ray.
- The sign of the ray's direction components is precomputed, making it possible to eliminate the comparisons of the computed `tNear` and `tFar` values in the original routine and just directly compute the respective near value and the far values. Because the comparisons that order these values from low to high in the original code are dependent on computed values, they can be inefficient for processors to execute, since the computation of their values must be completely finished before the comparison can be made.

`BBox::IntersectP()` 194
`LinearBVHNode` 222

Note also that this routine returns `true` if the ray segment is entirely inside the bounding box, even if the intersections are not within the ray's [`mint`, `maxt`] range; this property is

also desirable for BVH traversal. Because so many ray–bounding box intersection tests are performed while traversing the BVH tree, we found that this optimized method provided approximately a 15% performance improvement in overall rendering time compared to using `BBox::IntersectP()`.

⟨*BVHAccel Local Declarations*⟩ +≡
```
static inline bool IntersectP(const BBox &bounds, const Ray &ray,
        const Vector &invDir, const uint32_t dirIsNeg[3]) {
    ⟨Check for ray intersection against x and y slabs 226⟩
    ⟨Check for ray intersection against z slab⟩
    return (tmin < ray.maxt) && (tmax > ray.mint);
}
```

If the ray direction vector is negative, the "near" parametric intersection will be found with the slab with the larger of the two bounding values, and the far intersection will be found with the slab with the smaller of them. The implementation here uses this observation to compute the near and far parametric values in each direction directly.

⟨*Check for ray intersection against x and y slabs*⟩ ≡ 226
```
float tmin =  (bounds[  dirIsNeg[0]].x - ray.o.x) * invDir.x;
float tmax =  (bounds[1-dirIsNeg[0]].x - ray.o.x) * invDir.x;
float tymin = (bounds[  dirIsNeg[1]].y - ray.o.y) * invDir.y;
float tymax = (bounds[1-dirIsNeg[1]].y - ray.o.y) * invDir.y;
if ((tmin > tymax) || (tymin > tmax))
    return false;
if (tymin > tmin) tmin = tymin;
if (tymax < tmax) tmax = tymax;
```

The fragment ⟨*Check for ray intersection against z slab*⟩ is analogous and isn't included here.

If the current node is a leaf, then the ray must be tested for intersection with the primitives inside it. The next node to visit is then found from the `todo` stack; even if an intersection is found in the current node, the remaining nodes must be visited, in case one of them yields a closer intersection. However, if an intersection is found, the ray's `maxt` value will be updated to the intersection distance; this makes it possible to efficiently discard remaining nodes that are farther away than the intersection.

⟨*Intersect ray with primitives in leaf BVH node*⟩ ≡ 225
```
for (uint32_t i = 0; i < node->nPrimitives; ++i)
    if (primitives[node->primitivesOffset+i]->Intersect(ray, isect))
        hit = true;
if (todoOffset == 0) break;
nodeNum = todo[--todoOffset];
```

For an interior node that the ray hits, it is necessary to visit both of their children. As described above, it's desirable to visit the first child that the ray passes through before visiting the second one, in case there is a primitive that the ray intersects in the first one, so that the ray's `maxt` value can be updated, thus reducing the ray's extent and thus the number of node bounding boxes it intersects.

BBox 70
BBox::IntersectP() 194
BVHAccel::primitives 210
LinearBVHNode::
nPrimitives 222
Primitive::Intersect() 186
Ray 66
Vector 57

An efficient way to perform a front-to-back traversal without incurring the expense of intersecting the ray with both child nodes and comparing the distances is to use the sign of the ray's direction vector for the coordinate axis along which primitives were partitioned for the current node: if the sign is negative, we should visit the second child before the first child, since the primitives that went into the second child's subtree were on the upper side of the partition point. (And conversely for a positive-signed direction.) Doing this is straightforward: The offset for the node to be visited first is copied to `nodeNum` and the offset for the other node is added to the `todo` stack. (Recall that the first child is immediately after the current node due to the depth-first layout of nodes in memory.)

⟨*Put far BVH node on* `todo` *stack, advance to near node*⟩ ≡ 225

```
if (dirIsNeg[node->axis]) {
   todo[todoOffset++] = nodeNum + 1;
   nodeNum = node->secondChildOffset;
}
else {
   todo[todoOffset++] = node->secondChildOffset;
   nodeNum = nodeNum + 1;
}
```

The `BVHAccel::IntersectP()` method is essentially the same as the regular intersection method, with the usual two differences that `Primitives`' `IntersectP()` methods are called rather than `Intersect()`, and traversal stops immediately if any intersection is found.

4.5 KD-TREE ACCELERATOR

Binary space partitioning (BSP) trees adaptively subdivide space into irregularly sized regions. The most important consequence of this difference with regular grids is that they can be a much more effective data structure for storing irregularly distributed collections of geometry. A BSP tree starts with a bounding box that encompasses the entire scene. If the number of primitives in the box is greater than some threshold, the box is split in half by a plane. Primitives are then associated with whichever half they overlap and primitives that lie in both halves are associated with both of them. (This is in contrast to BVHs, where each primitive is assigned to only one of the two subgroups after a split.)

The splitting process continues recursively either until each leaf region in the resulting tree contains a sufficiently small number of primitives or until a maximum depth is reached. Because the splitting planes can be placed at arbitrary positions inside the overall bound and because different parts of 3D space can be refined to different degrees, BSP trees can easily handle uneven distributions of geometry.

Two variations of BSP trees are *kd-trees* and *octrees*. A kd-tree simply restricts the splitting plane to be perpendicular to one of the coordinate axes; this makes both traversal and construction of the tree more efficient, at the cost of some flexibility in how space is subdivided. The octree uses three axis-perpendicular planes to simultaneously split the box into eight regions at each step (typically by splitting down the center of the extent in each direction). In this section, we will implement a kd-tree for ray intersection

LinearBVHNode::
secondChildOffset 222

Primitive 185

acceleration in the KdTreeAccel class. Source code for this class can be found in the files accelerators/kdtreeaccel.h and accelerators/kdtreeaccel.cpp.

⟨*KdTreeAccel Declarations*⟩ ≡
```
class KdTreeAccel : public Aggregate {
public:
    ⟨KdTreeAccel Public Methods 245⟩
private:
    ⟨KdTreeAccel Private Methods⟩
    ⟨KdTreeAccel Private Data 229⟩
};
```

In addition to the primitives to be stored, the KdTreeAccel constructor takes a few parameters that are used to guide the decisions that will be made as the tree is built; these parameters are stored in member variables (isectCost, traversalCost, maxPrims, maxDepth, and emptyBonus) for later use. For simplicity of implementation, the KdTreeAccel requires that all of the primitives it stores be intersectable. We leave as an exercise the task of improving the implementation to do lazy refinement like the GridAccel does. Therefore, the constructor starts out by refining the given primitives until all are intersectable before building the tree. See Figure 4.13 for an overview of how the tree is built.

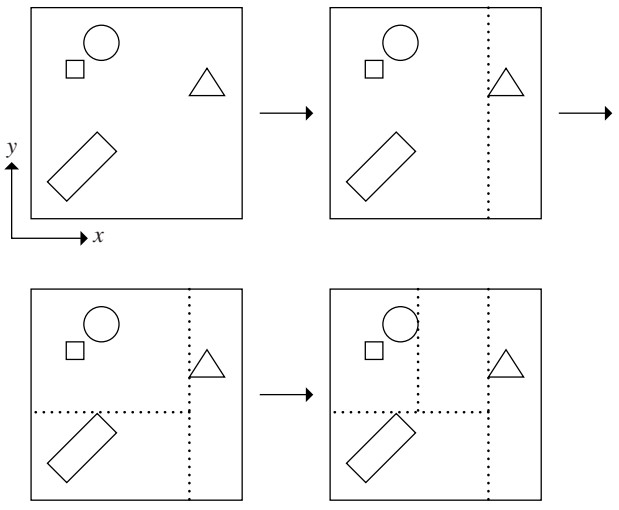

Figure 4.13: The kd-tree is built by recursively splitting the bounding box of the scene geometry along one of the coordinate axes. Here, the first split is along the x axis; it is placed so that the triangle is precisely alone in the right region and the rest of the primitives end up on the left. The left region is then refined a few more times with axis-aligned splitting planes. The details of the refinement criteria—which axis is used to split space at each step, at which position along the axis the plane is placed, and at what point refinement terminates—can all substantially affect the performance of the tree in practice.

Aggregate 192
GridAccel 196
KdTreeAccel 228

⟨*KdTreeAccel Method Definitions*⟩ ≡
```
KdTreeAccel::KdTreeAccel(const vector<Reference<Primitive> > &p,
                         int icost, int tcost, float ebonus, int maxp,
                         int md)
    : isectCost(icost), traversalCost(tcost), maxPrims(maxp), maxDepth(md),
      emptyBonus(ebonus) {
    for (uint32_t i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(primitives);
    ⟨Build kd-tree for accelerator 232⟩
}
```

⟨*KdTreeAccel Private Data*⟩ ≡ 228
```
int isectCost, traversalCost, maxPrims, maxDepth;
float emptyBonus;
vector<Reference<Primitive> > primitives;
```

4.5.1 TREE REPRESENTATION

The kd-tree is a binary tree, where each interior node always has both children and where leaves of the tree store the primitives that overlap them. Each interior node must provide access to three pieces of information:

- Split axis: which of the x, y, or z axes was split at this node.
- Split position: the position of the splitting plane along the axis.
- Children: information about how to reach the two child nodes beneath it.

Each leaf node needs to record only which primitives overlap it.

It is worth going through a bit of trouble to ensure that all interior nodes and many leaf nodes use just 8 bytes of memory (assuming 4-byte `float`s and pointers) because doing so ensures that four nodes will fit into a 32-byte cache line. Because there are often many nodes in the tree and because many nodes are generally accessed for each ray, minimizing the size of the node representation substantially improves cache performance. Our initial implementation used a 16-byte node representation; when we reduced the size to 8 bytes we obtained nearly a 20% speed increase. Both leaves and interior nodes are represented by the following `KdAccelNode` structure. The comments after each `union` member indicate whether a particular field is used for interior nodes, leaf nodes, or both.[6]

⟨*KdTreeAccel Local Declarations*⟩ ≡
```
struct KdAccelNode {
    ⟨KdAccelNode Methods 231⟩
    union {
        float split;            // Interior
        uint32_t onePrimitive;  // Leaf
        uint32_t *primitives;   // Leaf
    };
```

KdAccelNode 229
KdTreeAccel 228
Primitive 185
Primitive::FullyRefine() 186
Reference 1011

6 The attentive reader will note that on a system with 64-bit pointers the `KdAccelNode` structure will actually be 12 bytes, not 8. We leave the correction of this shortcoming to an exercise at the end of the chapter.

```
private:
    union {
        uint32_t flags;         // Both
        uint32_t nPrims;        // Leaf
        uint32_t aboveChild;    // Interior
    };
};
```

The two low-order bits of the `KdAccelNode::flags` variable are used to differentiate between interior nodes with x, y, and z splits (where these bits hold the values 0, 1, and 2, respectively) and leaf nodes (where these bits hold the value 3). It is relatively easy to store leaf nodes in 8 bytes: since the low 2 bits of `KdAccelNode::flags` are used to indicate that this is a leaf, the upper 30 bits of `KdAccelNode::nPrims` are available to record how many primitives overlap it. Then, if just a single primitive overlaps a `KdAccelNode` leaf, an unsigned integer index into the `KdTreeAccel::primitives` array identifies the `Primitive`. If more than one primitive overlaps, memory is dynamically allocated for an array of their indices pointed to by `KdAccelNode::primitives`.

Leaf nodes are easy to initialize, though we have to be careful with the details since both `flags` and `nPrims` share the same storage; we need to be careful to not clobber data for one of them while initializing the other. Furthermore, the number of primitives must be shifted two bits to the left before being stored so that the low two bits of `KdAccelNode::flags` can both be set to 1 to indicate that this is a leaf node.

⟨*KdTreeAccel Method Definitions*⟩ +≡

```
void KdAccelNode::initLeaf(uint32_t *primNums, int np,
                           MemoryArena &arena) {
    flags = 3;
    nPrims |= (np << 2);
    ⟨Store primitive ids for leaf node 230⟩
}
```

For leaf nodes with zero or one overlapping primitives, no dynamic memory allocation is necessary thanks to the `KdAccelNode::onePrimitive` field. For the case where multiple primitives overlap, the caller passes in a `MemoryArena` for allocating memory for the arrays of `Primitive` ids.

⟨*Store primitive ids for leaf node*⟩ ≡ 230

```
if (np == 0)
    onePrimitive = 0;
else if (np == 1)
    onePrimitive = primNums[0];
else {
    primitives = arena.Alloc<uint32_t>(np);
    for (int i = 0; i < np; ++i)
        primitives[i] = primNums[i];
}
```

Getting interior nodes down to 8 bytes is also reasonably straightforward. One 32-bit `float` stores the position along the chosen split axis where the node splits space, and, as

KdAccelNode 229
KdAccelNode::flags 229
KdAccelNode::nPrims 229
KdAccelNode::
onePrimitive 229
KdAccelNode::primitives 229
KdTreeAccel::primitives 229
MemoryArena 1015
MemoryArena::Alloc() 1016
Primitive 185

explained earlier, the lowest 2 bits of `KdAccelNode::flags` are used to record which axis the node was split along. All that is left is to store enough information to find the two children of the node as we're traversing the tree.

Rather than storing two pointers or offsets, we lay the nodes out in a way that lets us only store one child pointer: all of the nodes are allocated in a single contiguous block of memory, and the child of an interior node that is responsible for space below the splitting plane is always stored in the array position immediately after its parent (this also improves cache performance, by keeping at least one child close to its parent in memory). The other child, representing space above the splitting plane, will end up at somewhere else in the array; a single integer offset, `KdAccelNode::aboveChild`, stores its position in the nodes array. This representation is similar to the one used for BVH nodes in Section 4.4.3.

Given all those conventions, the code to initialize an interior node is straightforward. As in the `initLeaf()` method, it's important to assign the value to `flags` before setting `aboveChild`, and to compute the logical OR of the shifted above child value so as to not clobber the bits stored in `flags`.

⟨*KdAccelNode Methods*⟩ ≡ 229
```
void initInterior(uint32_t axis, uint32_t ac, float s) {
    split = s;
    flags = axis;
    aboveChild |= (ac << 2);
}
```

Finally, we'll provide a few methods to extract various values from the node, so that callers don't have to be aware of the subtle details of its representation in memory.

⟨*KdAccelNode Methods*⟩ +≡ 229
```
float SplitPos() const { return split; }
uint32_t nPrimitives() const { return nPrims >> 2; }
uint32_t SplitAxis() const { return flags & 3; }
bool IsLeaf() const { return (flags & 3) == 3; }
uint32_t AboveChild() const { return aboveChild >> 2; }
```

4.5.2 TREE CONSTRUCTION

The kd-tree is built with a recursive top-down algorithm. At each step, we have an axis-aligned region of space and a set of primitives that overlap that region. Either the region is split into two subregions and turned into an interior node, or a leaf node is created with the overlapping primitives, terminating the recursion.

As mentioned in the discussion of `KdAccelNodes`, all tree nodes are stored in a contiguous array. `KdTreeAccel::nextFreeNode` records the next node in this array that is available, and `KdTreeAccel::nAllocedNodes` records the total number that have been allocated. By setting both of them to zero and not allocating any nodes at start-up, the implementation here ensures that an allocation will be done immediately when the first node of the tree is initialized.

KdAccelNode 229
KdAccelNode::aboveChild 229
KdAccelNode::flags 229
KdAccelNode::nPrims 229
KdAccelNode::split 229
KdTreeAccel::
nAllocedNodes 232
KdTreeAccel::
nextFreeNode 232

It is also necessary to determine a maximum tree depth if one wasn't given to the constructor. Although the tree construction process will normally terminate naturally at a reasonable depth, it is important to cap the maximum depth so that the amount of memory used for the tree cannot grow without bound in pathological cases. We have found that the value $8 + 1.3 \log(N)$ gives a reasonable maximum depth for a variety of scenes.

⟨*Build kd-tree for accelerator*⟩ ≡ 229
```
nextFreeNode = nAllocedNodes = 0;
if (maxDepth <= 0)
    maxDepth = Round2Int(8 + 1.3f * Log2Int(float(primitives.size())));
⟨Compute bounds for kd-tree construction 232⟩
⟨Allocate working memory for kd-tree construction 236⟩
⟨Initialize primNums for kd-tree construction 232⟩
⟨Start recursive construction of kd-tree 233⟩
⟨Free working memory for kd-tree construction⟩
```

⟨*KdTreeAccel Private Data*⟩ +≡ 228
```
KdAccelNode *nodes;
int nAllocedNodes, nextFreeNode;
```

Because the construction routine will be repeatedly using the bounding boxes of the primitives along the way, they are stored in a `vector` before tree construction starts so that the potentially slow `Primitive::WorldBound()` methods don't need to be called repeatedly.

⟨*Compute bounds for kd-tree construction*⟩ ≡ 232
```
vector<BBox> primBounds;
primBounds.reserve(primitives.size());
for (uint32_t i = 0; i < primitives.size(); ++i) {
    BBox b = primitives[i]->WorldBound();
    bounds = Union(bounds, b);
    primBounds.push_back(b);
}
```

⟨*KdTreeAccel Private Data*⟩ +≡ 228
```
BBox bounds;
```

One of the parameters to the tree construction routine is an array of primitive indices indicating which primitives overlap the current node. Because all primitives overlap the root node (when the recursion begins) we start with an array initialized with values from zero through `primitives.size()-1`.

⟨*Initialize* `primNums` *for kd-tree construction*⟩ ≡ 232
```
uint32_t *primNums = new uint32_t[primitives.size()];
for (uint32_t i = 0; i < primitives.size(); ++i)
    primNums[i] = i;
```

`KdTreeAccel::buildTree()` is called for each tree node. It is responsible for deciding if the node should be an interior node or leaf and updating the data structures appropriately. The last three parameters, `edges`, `prims0`, and `prims1`, are pointers to data from the

BBox 70
KdAccelNode 229
KdTreeAccel::buildTree() 233
KdTreeAccel::primitives 229
Log2Int() 1001
Primitive::WorldBound() 185
prims0 240
prims1 240
Round2Int() 1002
Union() 72

⟨*Allocate working memory for kd-tree construction*⟩ fragment, which will be defined and documented in a few pages.

⟨*Start recursive construction of kd-tree*⟩ ≡ 232
```
buildTree(0, bounds, primBounds, primNums, primitives.size(),
          maxDepth, edges, prims0, prims1);
```

The main parameters to `KdTreeAccel::buildTree()` are the offset into the array of `KdAccelNodes` to use for the node that it creates, `nodeNum`; the bounding box that gives the region of space that the node covers, `nodeBounds`; and the indices of primitives that overlap it, `primNums`. The remainder of the parameters will be described later, closer to where they are used.

⟨*KdTreeAccel Method Definitions*⟩ +≡
```
void KdTreeAccel::buildTree(int nodeNum, const BBox &nodeBounds,
        const vector<BBox> &allPrimBounds, uint32_t *primNums,
        int nPrimitives, int depth, BoundEdge *edges[3],
        uint32_t *prims0, uint32_t *prims1, int badRefines) {
    ⟨Get next free node from nodes array 233⟩
    ⟨Initialize leaf node if termination criteria met 233⟩
    ⟨Initialize interior node and continue recursion 234⟩
}
```

If all of the allocated nodes have been used up, node memory is reallocated with twice as many entries and the old values are copied. The first time `KdTreeAccel::buildTree()` is called, `KdTreeAccel::nAllocedNodes` is zero and an initial block of tree nodes is allocated.

⟨*Get next free node from* nodes *array*⟩ ≡ 233
```
if (nextFreeNode == nAllocedNodes) {
    int nAlloc = max(2 * nAllocedNodes, 512);
    KdAccelNode *n = AllocAligned<KdAccelNode>(nAlloc);
    if (nAllocedNodes > 0) {
        memcpy(n, nodes, nAllocedNodes * sizeof(KdAccelNode));
        FreeAligned(nodes);
    }
    nodes = n;
    nAllocedNodes = nAlloc;
}
++nextFreeNode;
```

A leaf node is created (stopping the recursion) either if there are a sufficiently small number of primitives in the region, or if the maximum depth has been reached. The `depth` parameter starts out as the tree's maximum depth and is decremented at each level.

⟨*Initialize leaf node if termination criteria met*⟩ ≡ 233
```
if (nPrimitives <= maxPrims || depth == 0) {
    nodes[nodeNum].initLeaf(primNums, nPrimitives, arena);
    return;
}
```

AllocAligned() 1013
BBox 70
BoundEdge 235
KdAccelNode 229
KdAccelNode::initLeaf() 230
KdTreeAccel 228
KdTreeAccel::arena 234
KdTreeAccel::buildTree() 233
KdTreeAccel::maxPrims 229
KdTreeAccel::nAllocedNodes 232
KdTreeAccel::nextFreeNode 232
KdTreeAccel::nodes 232
prims0 240
prims1 240

As described earlier, `KdAccelNode::initLeaf()` uses a memory arena to allocate space for variable-sized arrays of primitives. Because the arena used here is a member variable, all of the memory it allocates will automatically be freed when the `KdTreeAccel` object is destroyed.

⟨*KdTreeAccel Private Data*⟩ +≡ 228
```
MemoryArena arena;
```

If this is an internal node, it is necessary to choose a splitting plane, classify the primitives with respect to that plane, and recurse.

⟨*Initialize interior node and continue recursion*⟩ ≡ 233
⟨*Choose split axis position for interior node* 236⟩
⟨*Create leaf if no good splits were found* 239⟩
⟨*Classify primitives with respect to split* 239⟩
⟨*Recursively initialize children nodes* 240⟩

Our implementation chooses a split using the surface area heuristic (SAH) introduced in Section 4.4.2. The SAH is applicable to kd-trees as well as BVHs; here, the estimated cost is computed for a series of candidate splitting planes in the node, and the split that gives the lowest cost is chosen.

In the implementation here, the intersection cost t_{isect} and the traversal cost t_{trav} can be set by the user; their default values are 80 and 1, respectively. Ultimately, it is the ratio of these two values that determines the behavior of the tree-building algorithm.[7] The greater ratio between these values compared to the values used for BVH construction reflects the fact that visiting a kd-tree node is relatively much less expensive than a BVH node.

One modification to the SAH used for BVH trees is that for kd-trees it is worth giving a slight preference to choosing splits where one of the children has no primitives overlapping it, since rays passing through these regions can immediately advance to the next kd-tree node without any ray–primitive intersection tests. Thus, the revised costs for unsplit and split regions are, respectively,

$$t_{\text{isect}} N, \quad \text{and}$$
$$t_{\text{trav}} + (1 - b_e)(p_B N_B t_{\text{isect}} + p_A N_A t_{\text{isect}}),$$

where b_e is a "bonus" value that is zero unless one of the two regions is completely empty, in which case it takes on a value between zero and one.

Given a way to compute the probabilities for the cost model, the only problem to address is how to generate candidate splitting positions and how to efficiently compute the cost for each candidate. It can be shown that the minimum cost with this model will be attained at a split that is coincident with one of the faces of one of the primitive's bounding

7 Many other implementations of this approach seem to use values for these costs that are much closer together, sometimes even approaching equal values (for example, see Hurley et al. 2002). The values used here gave the best performance for a number of test scenes in `pbrt`. We suspect that this discrepancy is due to the fact that ray–primitive intersection tests in `pbrt` require two virtual function calls and a ray world-to-object-space transformation, in addition to the cost of performing the actual intersection test. Highly optimized ray tracers that only support triangle primitives don't pay any of that additional cost. See Section 18.1.2 for further discussion of this design trade-off.

`KdAccelNode::initLeaf()` 230
`KdTreeAccel` 228
`MemoryArena` 1015

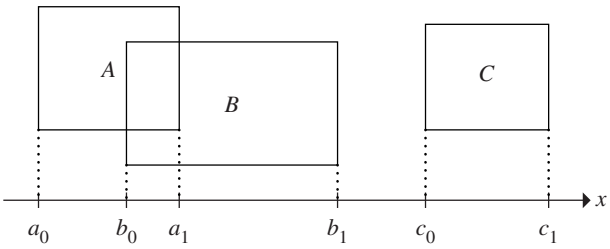

Figure 4.14: Given an axis along which we'd like to consider possible splits, the primitives' bounding boxes are projected onto the axis, which leads to an efficient algorithm to track how many primitives would be on each side of a particular splitting plane. Here, for example, a split at a_1 would leave A completely below the splitting plane, B straddling it, and C completely above it. Each point on the axis, a_0, a_1, b_0, b_1, c_0, and c_1, is represented by an instance of the `BoundEdge` structure.

boxes—there's no need to consider splits at intermediate positions. (To convince yourself of this, consider the behavior of the cost function between the edges of the faces.) Here, we will consider all bounding box faces inside the region for one or more of the three coordinate axes.

The cost for checking all of these candidates thus can be kept relatively low with a carefully structured algorithm. To compute these costs, we will sweep across the projections of the bounding boxes onto each axis and keep track of which gives the lowest cost (Figure 4.14). Each bounding box has two edges on each axis, each of which is represented by an instance of the `BoundEdge` structure. This structure records the position of the edge along the axis, whether it represents the start or end of a bounding box (going from low to high along the axis), and which primitive it is associated with.

⟨*KdTreeAccel Local Declarations*⟩ +≡
```
struct BoundEdge {
    ⟨BoundEdge Public Methods 235⟩
    float t;
    int primNum;
    enum { START, END } type;
};
```

⟨*BoundEdge Public Methods*⟩ ≡ 235
```
BoundEdge(float tt, int pn, bool starting) {
    t = tt;
    primNum = pn;
    type = starting ? START : END;
}
```

BoundEdge 235
BoundEdge::END 235
BoundEdge::primNum 235
BoundEdge::START 235
BoundEdge::t 235
BoundEdge::type 235

At most, `2 * primitives.size()` `BoundEdges` are needed for computing costs for any tree node, so the memory for the edges for all three axes is allocated once and then reused for each node that is created. The fragment ⟨*Free working memory for kd-tree construction*⟩, not included here, frees this space after the tree has been built.

⟨*Allocate working memory for kd-tree construction*⟩ ≡ 232
```
BoundEdge *edges[3];
for (int i = 0; i < 3; ++i)
    edges[i] = new BoundEdge[2*primitives.size()];
```

After determining the estimated cost for creating a leaf, `KdTreeAccel::buildTree()` chooses an axis to try to split along and computes the cost function for each candidate split. `bestAxis` and `bestOffset` record the axis and bounding box edge index that have given the lowest cost so far, `bestCost`. `invTotalSA` is initialized to the reciprocal of the node's surface area; its value will be used when computing the probabilities of rays passing through each of the candidate children nodes.

⟨*Choose split axis position for interior node*⟩ ≡ 234
```
int bestAxis = -1, bestOffset = -1;
float bestCost = INFINITY;
float oldCost = isectCost * float(nPrimitives);
float totalSA = nodeBounds.SurfaceArea();
float invTotalSA = 1.f / totalSA;
Vector d = nodeBounds.pMax - nodeBounds.pMin;
⟨Choose which axis to split along 236⟩
int retries = 0;
retrySplit:
⟨Initialize edges for axis 236⟩
⟨Compute cost of all splits for axis to find best 237⟩
```

This method first tries to find a split along the axis with the largest spatial extent; if successful, this choice helps to give regions of space that tend toward being square in shape. This is an intuitively sensible approach. Later, if it was unsuccessful in finding a good split along this axis, it will go back and try the others in turn.

⟨*Choose which axis to split along*⟩ ≡ 236
```
uint32_t axis = nodeBounds.MaximumExtent();
```

First the `edges` array for the axis is initialized using the bounding boxes of the overlapping primitives. The array is then sorted from low to high along the axis so that it can sweep over the box edges from first to last.

⟨*Initialize edges for* `axis`⟩ ≡ 236
```
for (int i = 0; i < nPrimitives; ++i) {
    int pn = primNums[i];
    const BBox &bbox = allPrimBounds[pn];
    edges[axis][2*i] =   BoundEdge(bbox.pMin[axis], pn, true);
    edges[axis][2*i+1] = BoundEdge(bbox.pMax[axis], pn, false);
}
sort(&edges[axis][0], &edges[axis][2*nPrimitives]);
```

The C++ standard library routine `sort()` requires that the structure being sorted define an ordering; this is done using the `BoundEdge::t` values. However, one subtlety is that if the `BoundEdge::t` values match, it is necessary to try to break the tie by comparing the

BBox 70
BBox::MaximumExtent() 73
BBox::SurfaceArea() 72
BoundEdge 235
BoundEdge::t 235
INFINITY 1002
KdTreeAccel::buildTree() 233
KdTreeAccel::isectCost 229
Vector 57

node's types; this is necessary since `sort()` depends on the fact that the only time `a < b` and `b < a` are both `false` is when `a == b`.

⟨*BoundEdge Public Methods*⟩ +≡ 235
```
bool operator<(const BoundEdge &e) const {
    if (t == e.t)
        return (int)type < (int)e.type;
    else return t < e.t;
}
```

Given the sorted array of edges, we'd like to quickly compute the cost function for a split at each one of them. The probabilities for a ray passing through each child node are easily computed using their surface areas, and the number of primitives on each side of the split is tracked by the variables `nBelow` and `nAbove`. We would like to keep their values updated such that if we chose to split at `edget` for a particular pass through the loop, `nBelow` will give the number of primitives that would end up below the splitting plane and `nAbove` would give the number above it.[8]

At the first edge, all primitives must be above that edge by definition, so `nAbove` is initialized to `nPrimitives` and `nBelow` is set to zero. When the loop is considering a split at the end of a bounding box's extent, `nAbove` needs to be decremented, since that box, which must have previously been above the splitting plane, will no longer be above it if splitting is done at the point. Similarly, after calculating the split cost, if the split candidate was at the start of a bounding box's extent, then the box will be on the below side for all subsequent splits. The tests at the start and end of the loop body update the primitive counts for these two cases.

⟨*Compute cost of all splits for* `axis` *to find best*⟩ ≡ 236
```
int nBelow = 0, nAbove = nPrimitives;
for (int i = 0; i < 2*nPrimitives; ++i) {
    if (edges[axis][i].type == BoundEdge::END) --nAbove;
    float edget = edges[axis][i].t;
    if (edget > nodeBounds.pMin[axis] &&
        edget < nodeBounds.pMax[axis]) {
        ⟨Compute cost for split at ith edge 238⟩
    }
    if (edges[axis][i].type == BoundEdge::START) ++nBelow;
}
```

Given all of this information, the cost for a particular split can be computed. `belowSA` and `aboveSA` hold the surface areas of the two candidate child bounds; they are easily computed by adding up the areas of the six faces.

BBox::pMax 71
BBox::pMin 71
BoundEdge 235
BoundEdge::END 235
BoundEdge::START 235
BoundEdge::t 235
BoundEdge::type 235

8 When multiple bounding box faces project to the same point on the axis, this invariant may not be true at those points. However, as implemented here it will only overestimate the counts and, more importantly, will have the correct value for one of the multiple times through the loop at each of those points, so the algorithm functions correctly in the end anyway.

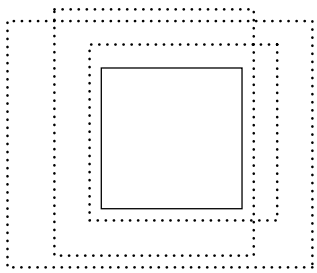

Figure 4.15: If multiple bounding boxes (dotted lines) overlap a kd-tree node (solid lines) as shown here, there is no possible split position that can result in fewer than all of the primitives being on both sides of it.

⟨*Compute cost for split at* `i` *th edge*⟩ ≡ 237

```
uint32_t otherAxis0 = (axis + 1) % 3, otherAxis1 = (axis + 2) % 3;
float belowSA = 2 * (d[otherAxis0] * d[otherAxis1] +
                     (edget - nodeBounds.pMin[axis]) *
                     (d[otherAxis0] + d[otherAxis1]));
float aboveSA = 2 * (d[otherAxis0] * d[otherAxis1] +
                     (nodeBounds.pMax[axis] - edget) *
                     (d[otherAxis0] + d[otherAxis1]));
float pBelow = belowSA * invTotalSA;
float pAbove = aboveSA * invTotalSA;
float eb = (nAbove == 0 || nBelow == 0) ? emptyBonus : 0.f;
float cost = traversalCost +
             isectCost * (1.f - eb) * (pBelow * nBelow + pAbove * nAbove);
⟨Update best split if this is lowest cost so far 238⟩
```

If the cost computed for this candidate split is the best one so far, the details of the split are recorded.

⟨*Update best split if this is lowest cost so far*⟩ ≡ 238

```
if (cost < bestCost)  {
    bestCost = cost;
    bestAxis = axis;
    bestOffset = i;
}
```

It may happen that there are no possible splits found in the previous tests (Figure 4.15 illustrates a case where this may happen). In this case, there isn't a single candidate position at which to split the node along the current axis. At this point, splitting is tried for the other two axes in turn. If neither of them can find a split (when `retries` is equal to two), then there is no useful way to refine the node, since both children will still have the same number of overlapping primitives. When this condition occurs, all that can be done is to give up and make a leaf node.

BBox::pMax 71
BBox::pMin 71

It is also possible that the best split will have a cost that is still higher than the cost for not splitting the node at all. If it is substantially worse and there aren't too many primitives, a leaf node is made immediately. Otherwise, `badRefines` keeps track of how many bad splits have been made so far above the current node of the tree. It's worth allowing a few slightly poor refinements since later splits may be able to find better ones given a smaller subset of primitives to consider.

⟨*Create leaf if no good splits were found*⟩ ≡ 234
```
if (bestAxis == -1 && retries < 2) {
    ++retries;
    axis = (axis+1) % 3;
    goto retrySplit;
}
if (bestCost > oldCost) ++badRefines;
if ((bestCost > 4.f * oldCost && nPrimitives < 16) ||
    bestAxis == -1 || badRefines == 3) {
    nodes[nodeNum].initLeaf(primNums, nPrimitives, arena);
    return;
}
```

Having chosen a split position, the bounding box edges can be used to classify the primitives as being above, below, or on both sides of the split in the same way as was done to keep track of `nBelow` and `nAbove` in the earlier code. Note that the `bestOffset` entry in the arrays is skipped in the loops below; this is necessary so that the primitive whose bounding box edge was used for the split isn't incorrectly categorized as being on both sides of the split.

⟨*Classify primitives with respect to split*⟩ ≡ 234
```
int n0 = 0, n1 = 0;
for (int i = 0; i < bestOffset; ++i)
    if (edges[bestAxis][i].type == BoundEdge::START)
        prims0[n0++] = edges[bestAxis][i].primNum;
for (int i = bestOffset+1; i < 2*nPrimitives; ++i)
    if (edges[bestAxis][i].type == BoundEdge::END)
        prims1[n1++] = edges[bestAxis][i].primNum;
```

BoundEdge 235
BoundEdge::END 235
BoundEdge::primNum 235
BoundEdge::START 235
BoundEdge::type 235
KdAccelNode::initLeaf() 230
KdTreeAccel::buildTree() 233
prims0 240
prims1 240

Recall that the node number of the "below" child of this node in the kd-tree nodes array is the current node number plus one. After the recursion has returned from that side of the tree, the `nextFreeNode` offset is used for the "above" child. The only other important detail here is that the `prims0` memory is passed directly for reuse by both children, while the `prims1` pointer is advanced forward first. This is necessary since the current invocation of `KdTreeAccel::buildTree()` depends on its `prims1` values being preserved over the first recursive call to `KdTreeAccel::buildTree()` in the following, since it must be passed as a parameter to the second call. However, there is no corresponding need to preserve the `edges` values or to preserve `prims0` beyond its immediate use in the first recursive call.

⟨*Recursively initialize children nodes*⟩ ≡ 234
```
float tsplit = edges[bestAxis][bestOffset].t;
BBox bounds0 = nodeBounds, bounds1 = nodeBounds;
bounds0.pMax[bestAxis] = bounds1.pMin[bestAxis] = tsplit;
buildTree(nodeNum+1, bounds0,
          allPrimBounds, prims0, n0, depth-1, edges,
          prims0, prims1 + nPrimitives, badRefines);
uint32_t aboveChild = nextFreeNode;
nodes[nodeNum].initInterior(bestAxis, aboveChild, tsplit);
buildTree(aboveChild, bounds1, allPrimBounds, prims1, n1,
          depth-1, edges, prims0, prims1 + nPrimitives, badRefines);
```

Thus, much more space is needed for the `prims1` array of integers for storing the worst-case possible number of overlapping primitive numbers than for the `prims0` array, which only needs to handle the primitives at a single level at a time.

⟨*Allocate working memory for kd-tree construction*⟩ +≡ 232
```
uint32_t *prims0 = new uint32_t[primitives.size()];
uint32_t *prims1 = new uint32_t[(maxDepth+1) * primitives.size()];
```

4.5.3 TRAVERSAL

Figure 4.16 shows the basic process of ray traversal through the tree. Intersecting the ray with the tree's overall bounds gives initial `tmin` and `tmax` values, marked with points in the figure. As with the other accelerators in this chapter, if the ray misses the scene bounds, this method can immediately return `false`. Otherwise, it starts to descend into the tree, starting at the root. At each interior node, it determines which of the two children the ray enters first and processes both children in order. Traversal ends either when the ray exits the tree or when the closest intersection is found.

⟨*KdTreeAccel Method Definitions*⟩ +≡
```
bool KdTreeAccel::Intersect(const Ray &ray,
                            Intersection *isect) const {
    ⟨Compute initial parametric range of ray inside kd-tree extent 240⟩
    ⟨Prepare to traverse kd-tree for ray 241⟩
    ⟨Traverse kd-tree nodes in order for ray 242⟩
}
```

The algorithm starts by finding the overall parametric range $[t_{\min}, t_{\max}]$ of the ray's overlap with the tree, exiting immediately if there is no overlap.

⟨*Compute initial parametric range of ray inside kd-tree extent*⟩ ≡ 240
```
float tmin, tmax;
if (!bounds.IntersectP(ray, &tmin, &tmax))
    return false;
```

The array of `KdToDo` structures is used to record the nodes yet to be processed for the ray; it is ordered so that the last active entry in the array is the next node that should be considered. The maximum number of entries needed in this array is the maximum

BBox 70
BBox::IntersectP() 194
BBox::pMax 71
BBox::pMin 71
BoundEdge::t 235
Intersection 186
KdAccelNode::
initInterior() 231
KdTreeAccel 228
KdTreeAccel::buildTree() 233
KdTreeAccel::
nextFreeNode 232
KdTreeAccel::primitives 229
prims0 240
prims1 240
Ray 66

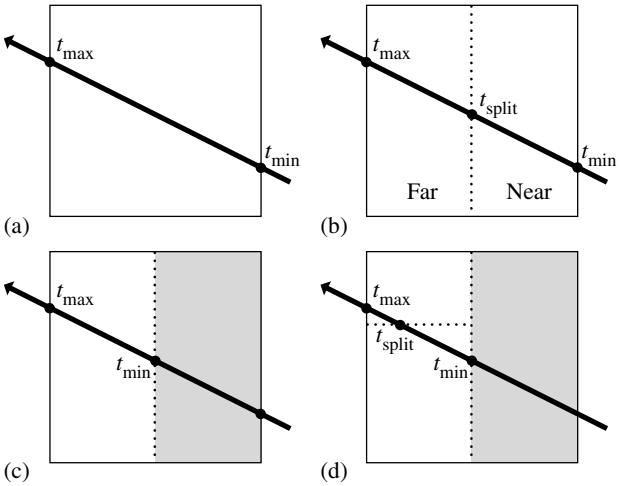

Figure 4.16: Traversal of a Ray through the Kd-Tree. (a) The ray is intersected with the bounds of the tree, giving an initial parametric $[t_{min}, t_{max}]$ range to consider. (b) Because this range is nonempty, it is necessary to consider the two children of the root node here. The ray first enters the child on the right, labeled "near," where it has a parametric range $[t_{min}, t_{split}]$. If the near node is a leaf with primitives in it, ray–primitive intersection tests are performed; otherwise, its children nodes are processed. (c) If no hit is found in the node, or if a hit is found beyond $[t_{min}, t_{split}]$, then the far node, on the left, is processed. (d) This sequence continues—processing tree nodes in a depth-first, front-to-back traversal—until the closest intersection is found or the ray exits the tree.

depth of the kd-tree; the array size used in the following should be more than enough in practice.

⟨*Prepare to traverse kd-tree for ray*⟩ ≡ 240
```
Vector invDir(1.f/ray.d.x, 1.f/ray.d.y, 1.f/ray.d.z);
#define MAX_TODO 64
KdToDo todo[MAX_TODO];
int todoPos = 0;
```

⟨*KdTreeAccel Declarations*⟩ +≡
```
struct KdToDo {
    const KdAccelNode *node;
    float tmin, tmax;
};
```

KdAccelNode 229
KdToDo 241
Ray::d 67
Vector 57

The traversal continues through the nodes, processing a single leaf or interior node each time through the loop. The values `tmin` and `tmax` will always hold the parametric range for the ray's overlap with the current node.

⟨*Traverse kd-tree nodes in order for ray*⟩ ≡ 240
```
bool hit = false;
const KdAccelNode *node = &nodes[0];
while (node != NULL) {
    ⟨Bail out if we found a hit closer than the current node 242⟩
    if (!node->IsLeaf()) {
        ⟨Process kd-tree interior node 242⟩
    }
    else {
        ⟨Check for intersections inside leaf node 244⟩
        ⟨Grab next node to process from todo list 245⟩
    }
}
return hit;
```

An intersection may have been previously found in a primitive that overlaps multiple nodes. If the intersection was outside the current node when first detected, it is necessary to keep traversing the tree until we come to a node where `tmin` is beyond the intersection. Only then is it certain that there is no closer intersection with some other primitive.

⟨*Bail out if we found a hit closer than the current node*⟩ ≡ 242
```
if (ray.maxt < tmin) break;
```

For interior tree nodes the first thing to do is to intersect the ray with the node's splitting plane; given the intersection point, we can determine if one or both of the children nodes need to be processed and in what order the ray passes through them.

⟨*Process kd-tree interior node*⟩ ≡ 242
```
⟨Compute parametric distance along ray to split plane 242⟩
⟨Get node children pointers for ray 243⟩
⟨Advance to next child node, possibly enqueue other child 244⟩
```

The parametric distance to the split plane is computed in the same manner as was done in computing the intersection of a ray and an axis-aligned plane for the ray–bounding box test. We use the precomputed `invDir` value to save a divide each time through the loop.

⟨*Compute parametric distance along ray to split plane*⟩ ≡ 242
```
int axis = node->SplitAxis();
float tplane = (node->SplitPos() - ray.o[axis]) * invDir[axis];
```

Now it is necessary to determine the order in which the ray encounters the children nodes, so that the tree is traversed in front-to-back order along the ray. Figure 4.17 shows the geometry of this computation. The position of the ray's origin with respect to the splitting plane is enough to distinguish between the two cases, ignoring for now the case where the ray doesn't actually pass through one of the two nodes. The rare case when the ray's origin lies on the splitting plane requires careful handling in this case, as its direction needs to be used instead to discriminate between the two cases.

KdAccelNode 229
KdAccelNode::IsLeaf() 231
KdAccelNode::SplitAxis() 231
KdAccelNode::SplitPos() 231
KdTreeAccel::nodes 232
Ray::maxt 67
Ray::o 67

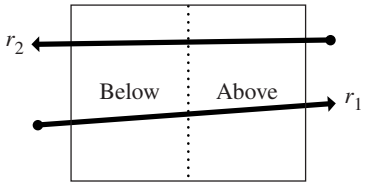

Figure 4.17: The position of the origin of the ray with respect to the splitting plane can be used to determine which of the node's children should be processed first. If the origin of a ray like r_1 is on the "below" side of the splitting plane, we should process the below child before the above child, and vice versa.

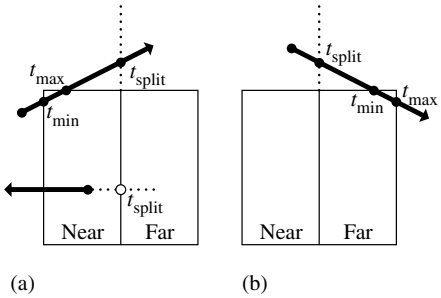

Figure 4.18: Two cases where both children of a node don't need to be processed because the ray doesn't overlap them. (a) The top ray intersects the splitting plane beyond the ray's t_{max} position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated by a negative t_{split} value. (b) The ray intersects the plane before the ray's t_{min} value, indicating that the near child doesn't need processing.

⟨*Get node children pointers for ray*⟩ ≡ 242

```
const KdAccelNode *firstChild, *secondChild;
int belowFirst = (ray.o[axis] <  node->SplitPos()) ||
                 (ray.o[axis] == node->SplitPos() && ray.d[axis] >= 0);
if (belowFirst) {
    firstChild = node + 1;
    secondChild = &nodes[node->AboveChild()];
}
else {
    firstChild = &nodes[node->AboveChild()];
    secondChild = node + 1;
}
```

KdAccelNode 229
KdAccelNode::SplitPos() 231
Ray::o 67

It may not be necessary to process both children of this node. Figure 4.18 shows some configurations where the ray only passes through one of the children. The ray will never

miss both children, since otherwise the current interior node should never have been traversed.

The first `if` test in the following code corresponds to Figure 4.18(a): only the near node needs to be processed if it can be shown that the ray doesn't overlap the far node because it faces away from it or doesn't overlap it because $t_{\text{split}} > t_{\text{max}}$. Figure 4.18(b) shows the similar case tested in the second `if` test: the near node may not need processing if the ray doesn't overlap it. Otherwise, the `else` clause handles the case of both children needing processing; the near node will be processed next, and the far node goes on the `todo` list.

⟨*Advance to next child node, possibly enqueue other child*⟩ ≡ 242
```
if (tplane > tmax || tplane <= 0)
    node = firstChild;
else if (tplane < tmin)
    node = secondChild;
else {
    ⟨Enqueue secondChild in todo list 244⟩
    node = firstChild;
    tmax = tplane;
}
```

⟨*Enqueue* `secondChild` *in todo list*⟩ ≡ 244
```
todo[todoPos].node = secondChild;
todo[todoPos].tmin = tplane;
todo[todoPos].tmax = tmax;
++todoPos;
```

If the current node is a leaf, intersection tests are performed against the primitives in the leaf.

⟨*Check for intersections inside leaf node*⟩ ≡ 242
```
uint32_t nPrimitives = node->nPrimitives();
if (nPrimitives == 1) {
    const Reference<Primitive> &prim = primitives[node->onePrimitive];
    ⟨Check one primitive inside leaf node 244⟩
}
else {
    uint32_t *prims = node->primitives;
    for (uint32_t i = 0; i < nPrimitives; ++i) {
        const Reference<Primitive> &prim = primitives[prims[i]];
        ⟨Check one primitive inside leaf node 244⟩
    }
}
```

Processing an individual primitive is just a matter of passing the intersection request on to the primitive.

⟨*Check one primitive inside leaf node*⟩ ≡ 244
```
if (prim->Intersect(ray, isect))
    hit = true;
```

KdAccelNode::nPrimitives() 231
KdAccelNode::onePrimitive 229
KdAccelNode::primitives 229
KdToDo::node 241
KdToDo::tmax 241
KdToDo::tmin 241
KdTreeAccel::primitives 229
Primitive 185
Primitive::Intersect() 186
Reference 1011

After doing the intersection tests at the leaf node, the next node to process is loaded from the `todo` array. If no more nodes remain, then the ray has passed through the tree without hitting anything.

⟨*Grab next node to process from todo list*⟩ ≡ 242
```
if (todoPos > 0) {
    --todoPos;
    node = todo[todoPos].node;
    tmin = todo[todoPos].tmin;
    tmax = todo[todoPos].tmax;
}
else
    break;
```

Like the `GridAccel` and `BVHAccel`, the `KdTreeAccel` has a specialized intersection method for shadow rays that is not shown here. It is similar to the `KdTreeAccel::Intersect()` method, just calling `Primitive::IntersectP()` method and returning `true` as soon as it finds any intersection without worrying about finding the closest one.

⟨*KdTreeAccel Public Methods*⟩ ≡ 228
```
bool IntersectP(const Ray &ray) const;
```

4.6 DEBUGGING AGGREGATES

Bugs in aggregates can be notoriously difficult to find and fix; once an aggregate implementation mostly works, the difficulty is that it does the correct thing for *almost* all of the rays that it's given and it's only a very small subset where a bug manifests itself. Even worse, the bug may be due to a small error made back when the acceleration structure was first built. Working backward from a ray with an incorrect intersection to the original source of the bug can be a very tedious process. We have learned some effective techniques and built some useful testing infrastructure for debugging aggregates that we will discuss here.

First, we need to define what a correct result for an intersection calculation is. We will say that given a ray and a collection of primitives, the correct intersection result is a primitive the ray hits with minimum t value along the ray, subject to the ray's parametric `mint-maxt` range, so long as the ray also intersects the bounding box returned by the primitive.

There are a few subtleties in this statement. One is that it is possible that multiple primitives will report an intersection at the same t value. In this case, we say that a correct answer is an intersection with any of those primitives. It's not worth the implementation complexity to define a more specific requirement—for example, that the primitive that appeared first in the input file must be reported; this constraint would make accelerator implementations needlessly complex and introduce overhead that's not generally worthwhile for rendering.

The second subtlety stems from small numeric inconsistencies between the primitive's bounding box and the primitive's intersection routine. There may be rays that are reported to not intersect the bounding box, yet the primitive's `Intersect()` method may

BVHAccel 209
GridAccel 196
KdToDo::node 241
KdToDo::tmax 241
KdToDo::tmin 241
KdTreeAccel 228
KdTreeAccel::Intersect() 240
Primitive::IntersectP() 186
Ray 66

report intersection. Even if the bounding box provided by the primitive is correct (i.e., it fully encompasses the primitive's spatial extent), it is possible that the inevitable small errors from floating-point roundoff in the `BBox::IntersectP()` routine will mean that a ray is reported to not intersect the bounding box even though the primitive's intersection method reports that the ray does in fact intersect it.[9] It's also not reasonable to require accelerators to report only primitive intersections where the ray also intersects the primitive's bounding box: for example, the grid accelerator tests the ray against all of the primitives in a grid voxel, regardless of whether it intersects their bounding boxes; for that algorithm, there isn't any reason to also test the ray against all of the primitives' bounding boxes given the knowledge that the primitive's bound overlapped the grid voxel and the ray passes through the grid voxel.

Given all this, for the purposes of defining a correct accelerator, we will therefore also say that primitive intersections where the ray is not found to intersect the bounding box are not expected to be found by the accelerator, but that it is also not incorrect for an accelerator to report such an intersection.

This definition of correctness for intersection computations means that valid image differences can occur when a scene is rendered with different accelerators. For example, one may choose one instance of an ambiguous intersection, where two or more primitives report an intersection with the same t value, and the other accelerator may report another instance. Even if this were not the case, testing and debugging accelerators by comparing image differences is somewhat unwieldy; given a small difference in two rendered images, working backward to find a bug in an accelerator that led to that difference can be a significant debugging chore.

4.6.1 FINDING BUGS IN AGGREGATES

`pbrt` provides an `AggregateTest` `Renderer` for testing aggregate implementations. Its implementation is straightforward: given a scene, it generates a large number of random rays in the scene and first traces each one using whichever accelerator was specified in the scene's description file. It then exhaustively tests the ray for intersection against every primitive in the scene. If the results are inconsistent (subject to the definition of a correct intersection calculation above), then the accelerator being tested must have a bug. Information about these rays is printed for use in later debugging runs. `AggregateTest` is found in the files `renderers/aggregatetest.h` and `renderers/aggregatetest.cpp`.

Finding and fixing bugs using directed tests like these is generally much easier than finding and fixing them after seeing a surprising error in an image. Isolating a single ray where the accelerator is not computing the right result helps narrow the debugging problem; to the extent that automated tests like those here can find instances of bugs, the easier the debugging process is. Another significant advantage of targeted testing code is that, when one makes changes to the system, one can run the tests and ensure that a subtle bug hasn't been introduced by the tests; the returns from the work to implement tests like these in the first place are generally worthwhile.

AggregateTest 247
BBox::IntersectP() 194
Renderer 24

9 This shortcoming presumably could be addressed through careful analysis of the roundoff error in the `BBox::IntersectP()` routine and by appropriate modifications to it to ensure that the computation is sufficiently conservative that this problem doesn't occur.

The AggregateTest constructor is not included here; it fully refines all of the primitives passed to it, computes their bounding boxes, and stores a count of the number of test iterations to run.

⟨*AggregateTest Private Data*⟩ ≡
```
int nIterations;
vector<Reference<Primitive> > primitives;
vector<BBox> bboxes;
```

Testing is done in the Render() method. It first computes a bounding box that is moderately larger than the full scene extent; random rays will be generated inside this bounding box. It then runs for the number of requested iterations, generating random rays and computing intersections with them.

⟨*AggregateTest Method Definitions*⟩ ≡
```
void AggregateTest::Render(const Scene *scene) {
    RNG rng;
    ⟨Compute bounding box of region used to generate random rays 247⟩
    Point lastHit;
    float lastEps = 0.f;
    for (int i = 0; i < nIterations; ++i) {
        ⟨Choose random rays, rayAccel and rayAll for testing 247⟩
        ⟨Compute intersections using accelerator and exhaustive testing 248⟩
        ⟨Report any inconsistencies between intersections 249⟩
    }
}
```

⟨*Compute bounding box of region used to generate random rays*⟩ ≡ 247
```
BBox bbox = scene->WorldBound();
bbox.Expand(bbox.pMax[bbox.MaximumExtent()] -
            bbox.pMin[bbox.MaximumExtent()]);
```

The ⟨*Choose random rays, rayAccel and rayAll for testing*⟩ generates random rays in the scene. Its goal is to be as efficient as possible at generating rays that are likely to be troublesome and expose corner cases in the implementations of accelerators.[10]

⟨*Choose random rays, rayAccel and rayAll for testing*⟩ ≡ 247
```
⟨Choose ray origin for testing accelerator 248⟩
⟨Choose ray direction for testing accelerator 248⟩
⟨Choose ray epsilon for testing accelerator 248⟩
Ray rayAccel(org, dir, eps);
Ray rayAll = rayAccel;
```

The ray origin is chosen in one of two ways: either as a random point inside the scene's bounding box or at the surface hit by the previous ray. Starting some of the rays on (or, strictly speaking, near) scene surfaces is important; not only are the majority of

AggregateTest::
nIterations 247
BBox 70
BBox::Expand() 72
Point 63
Primitive 185
Ray 66
Reference 1011
RNG 1003
Scene 22
Scene::WorldBound() 24

10 Another effective approach for generating these testing rays would be to log all of the rays generated during the regular process of rendering the scene and then rerun them through this testing code to make sure the intersection results are consistent. This modification is left for an exercise at the end of the chapter.

rays traced in the process of rendering rays leaving intersected surfaces, but errors from incorrect intersection computations from these rays aren't always easily visible. If rays from the camera have egregiously wrong intersection results, the bug will be obviously visible in the rendered image. Rays reflected from surfaces that have wrong intersection results may not have as obvious a visual manifestation.

⟨*Choose ray origin for testing accelerator*⟩ ≡ 247
```
Point org(Lerp(rng.RandomFloat(), bbox.pMin.x, bbox.pMax.x),
          Lerp(rng.RandomFloat(), bbox.pMin.y, bbox.pMax.y),
          Lerp(rng.RandomFloat(), bbox.pMin.z, bbox.pMax.z));
if ((rng.RandomUInt() % 4) == 0) org = lastHit;
```

The ray direction is usually chosen by randomly selecting a direction. However, occasionally setting two of the direction vectors to zero is worthwhile: rays parallel to a coordinate axis can be problematic, so exercising this case is useful.

⟨*Choose ray direction for testing accelerator*⟩ ≡ 247
```
Vector dir = UniformSampleSphere(rng.RandomFloat(), rng.RandomFloat());
if ((rng.RandomUInt() % 32) == 0)      dir.x = dir.y = 0.f;
else if ((rng.RandomUInt() % 32) == 0) dir.x = dir.z = 0.f;
else if ((rng.RandomUInt() % 32) == 0) dir.y = dir.z = 0.f;
```

Finally, the "epsilon" value for the ray (the minimum parameteric distance before which intersections are ignored) is chosen. The implementation randomly chooses between three typical values—zero, the epsilon value returned at the last surface intersection, and a small floating-point value.

⟨*Choose ray epsilon for testing accelerator*⟩ ≡ 247
```
float eps = 0.f;
if (rng.RandomFloat() < .25) eps = lastEps;
else if (rng.RandomFloat() < .25) eps = 1e-3f;
```

Given the ray, `AggregateTest` uses both the regular `Aggregate` and an exhaustive test of all primitives in the scene to check for intersections. Note that it checks for the case where the ray is not reported to intersect the bounding box but still hits the geometry here, setting `inconsistentBounds` in that case. Inconsistent intersections for these rays won't be reported.

⟨*Compute intersections using accelerator and exhaustive testing*⟩ ≡ 247
```
Intersection isectAccel, isectAll;
bool hitAccel = scene->Intersect(rayAccel, &isectAccel);
bool hitAll = false;
bool inconsistentBounds = false;
for (uint32_t j = 0; j < primitives.size(); ++j) {
    if (bboxes[j].IntersectP(rayAll))
        hitAll |= primitives[j]->Intersect(rayAll, &isectAll);
    else if (primitives[j]->Intersect(rayAll, &isectAll))
        inconsistentBounds = true;
}
```

Aggregate 192
BBox::IntersectP() 194
BBox::pMax 71
BBox::pMin 71
Intersection 186
Lerp() 1000
Point 63
Primitive::Intersect() 186
RNG::RandomFloat() 1003
RNG::RandomUInt() 1003
Scene::Intersect() 23
Vector 57

As long as no intersections were found where the ray hit the primitive but not its bounding box, then if the exhaustive test and the accelerator compute different parametric t values for the intersection point, a warning is printed here, including information about the origin and direction of the ray that exhibits the bug. Rather than use the conventional `%f` `printf()` formatting for the floating-point values, the code here uses `%a`, which prints the given value as a hexidecimal floating-point value. For example, the value 2.5 is printed `0x1.4p+1`. This representation of floating-point values is useful in that it is guaranteed to represent the floating-point value exactly, without any roundoff error. (In contrast, the number will generally be rounded when using `%f`.) It's important that the ray that exhibits the bug be stored precisely, since a slight perturbation to it may not exhibit the bug any more.

⟨*Report any inconsistencies between intersections*⟩ ≡ 247

```
if (!inconsistentBounds &&
    ((hitAccel != hitAll) || (rayAccel.maxt != rayAll.maxt)))
    Warning("Disagreement: t accel %.16g [%a] t exhaustive %.16g [%a]\n"
            "Ray: org [%a, %a, %a], dir [%a, %a, %a], mint = %a",
            rayAccel.maxt, rayAll.maxt, rayAccel.maxt, rayAll.maxt,
            rayAll.o.x, rayAll.o.y, rayAll.o.z,
            rayAll.d.x, rayAll.d.y, rayAll.d.z, rayAll.mint);
if (hitAll) {
    lastHit = rayAll(rayAll.maxt);
    lastEps = isectAll.rayEpsilon;
}
```

4.6.2 FIXING BUGS IN AGGREGATES

Finding a ray, scene, and accelerator where an incorrect intersection is found is only a start; tracking down the actual bug from this point is not easy. The first step in fixing this sort of bug is to determine which primitive is the one that should have been determined to be the closest intersecting primitive—from there, the question is "Why wasn't the ray found to intersect with it?" Adding code to the accelerator's constructor to immediately trace the ray that was found to hit the bug after the acceleration data structure is built gives an easy point at which to set a breakpoint in the debugger.

For accelerators based on spatial subdivision, one can take the position of the missed intersection and determine which spatial region the intersection point lies in. (The intersection point may also lie on the boundary between two nodes, which can be a problematic case for implementations.) The bug then must come from one of two causes:

- There is an error in the traversal code, and the ray never passes through the node where the intersection lies.
- Or, there is an error in the code that builds the accelerator, and the primitive is not present in the node where the intersection occurs.

Intersection::rayEpsilon 186
Ray::d 67
Ray::maxt 67
Ray::o 67

For example, when traversing a spatial data structure, if the ray doesn't pass through the node that holds the geometry, then at some parent of that node the traversal code will make an incorrect decision and decide to not recurse down to the subtree that has the node with the intersection. Finding the node where the intersection occurs and from that

the path from the root of the tree to that node makes it possible to isolate where this error happens. Alternatively, if the ray passes through that node but the primitive isn't present in it, then the construction code needs to be examined to figure out why the primitive wasn't included in the node.

For accelerators based on primitive subdivision, the task is similar. The primitive that should have been hit will be present in one or more nodes of the data struture; the question again is "Why didn't the ray visit those nodes, or if it did, then why wasn't an intersection found?"

Drilling down to the source of these bugs can often be done more easily by instrumenting the code with `printf()` calls that show what the system is doing (as it traverses the data structure or builds it, respectively). We have found that after narrowing down to a scene and ray that exhibits a bug, that a mixture of both printing detailed information about the code's execution and stepping through execution in a debugger is effective. One advantage of printing out a trace of the execution of the code is that it can be easier to work forward and backward through the trace file to find the source of error.

4.6.3 AGGREGATE PERFORMANCE BUGS

Beyond correctness bugs, errors that cause performance problems can be nefarious and difficult to find—with these bugs, although the system still computes the correct result, it just does so very inefficiently. An example of this type of bug was present in the first version of the `pbrt` system: due to a subtle bug in code that computed specular refraction directions, rays with floating-point "not a number" values for their direction components would very rarely be generated when rendering the ecosystem scene in Figure 4.1. When given a ray with NaN direction components, the kd-tree accelerator traversal code will visit *every* node in the entire tree. For the ecosystem scene, this meant that these rays would be tested for intersection with all 19 million triangles in the scene. This is obviously extremely wasteful, but it only happened for a handful of rays in the scene, so the overall rendering time wasn't sufficiently bad to be obviously wrong. When we found and fixed this bug, performance rendering that scene increased by a factor of three.

The best approach we have found to finding these sorts of bugs is to liberally gather statistics about the code's execution and to use visualization tools to understand its behavior. For example, simple code to track the maximum number of ray–primitive intersection tests for all the rays traced during rendering would have made it obvious that, for at least one ray, over 19 million intersection tests were being performed. From this insight, working backward to figure out which ray was the culprit and then why this was the case would have been relatively straightforward.

FURTHER READING

After the introduction of the ray-tracing algorithm, an enormous amount of research was done to try to find effective ways to speed it up, primarily by developing improved ray-tracing acceleration structures. Arvo and Kirk's chapter in *An Introduction to Ray Tracing* (Glassner 1989a) summarizes the state of the art as of 1989 and still provides an

excellent taxonomy for categorizing different approaches to ray intersection acceleration. *Ray Tracing News* (*www.acm.org/tog/resources/RTNews/*) is a very good resource for general ray-tracing information and has particularly useful discussions about intersection acceleration approaches, implementation issues, and tricks of the trade.

Kirk and Arvo (1988) introduced the unifying principle of *meta-hierarchies*. They showed that by implementing acceleration data structures to conform to the same interface as is used for primitives in the scene, it's easy to mix and match multiple intersection acceleration schemes. `pbrt` follows this model since the `Aggregate` inherits from the `Primitive` base class.

In the interests of making it easier to compare the performance of different ray intersection algorithms, there have been some efforts to create standard databases of scenes to test various ray intersection algorithms, notably Haines's "standard procedural database" (SPD) (Haines 1987) and Lext et al.'s BART scenes, which include animation (Lext, Assarsson, and Möller 2001). A few of the SPD scenes are available in the `pbrt` file format in the `pbrt` distribution.

Grids

Fujimoto, Tanaka, and Iwata (1986) were the first to introduce uniform voxel grids for ray tracing, similar to the approach implemented in this chapter. Snyder and Barr (1987) described a number of key improvements to this approach and showed their use for rendering extremely complex scenes. Hierarchical grids were first described by Jevans and Wyvill (1989). More complex techniques for hierarchical grids were developed by Cazals, Drettakis, and Puech (1995) and Klimaszewski and Sederberg (1997). The grid traversal method used in this chapter is essentially the one described by Cleary and Wyvill (1988).

Choosing an optimal grid resolution has received attention from a number of researchers. A recent paper in this area is by Ize et al. (2007), who provided a solid foundation for selecting an optimal grid resolution and for deciding when to refine into subgrids, when hierarchical grids are being used. They derived theoretical results using a number of simplifying assumptions and then showed the applicability of the results to rendering real-world scenes. They also included a good selection of pointers to previous work in this area.

Lagae and Dutré (2008a) described an innovative representation for uniform grids that has the desirable properties that not only does each primitive have a single index into a voxel, but each voxel has only a single primitive index. They show that this representation has very low memory usage and is still quite efficient.

Hunt and Mark (2008) showed that building grids in perspective space, where the center of projection is the camera or a light source, can make tracing rays from the camera or light substantially more efficient. Although this approach requires multiple acceleration structures, the performance benefits from multiple specialized structures for different classes of rays can be substantial. Their approach is also notable in that it is in some ways a middle-ground between rasterization and ray tracing.

Aggregate 192
Primitive 185

Bounding Volume Hierarchies

Clark (1976) first suggested using bounding volumes to cull collections of objects for standard visible-surface determination algorithms. Building on this work, Rubin and

Whitted (1980) developed the first hierarchical data structures for scene representation for fast ray tracing, although their method depended on the user to define the hierarchy. Kay and Kajiya (1986), implemented one of the first practical object subdivision approaches based on bounding objects with collections of slabs.

Goldsmith and Salmon (1987) described an algorithm for automatically computing bounding volume hierarchies and applied techniques for estimating the probability of a ray intersecting a bounding volume based on the volume's surface area. Most current methods for building BVHs are based on top-down construction of the tree, first creating the root node and then partitioning the primitives into children and continuing recursively. An alternative approach was demonstrated by Walter et al. (2008), who showed that bottom-up construction, where the leaves are created first and then agglomerated into parent nodes, is viable and can build somewhat better trees than top-down approaches. Kensler (2008) presented algorithms that make local adjustments to the BVH tree after it has been built to improve its quality.

The `BVHAccel` implementation in this chapter is based on the construction algorithm described by Wald (2007) and Gunther et al. (2007). The bounding box test is the one introduced by Williams et al. (2005). An even more efficient bounding box test that does additional precomputation in exchange for higher performance when the same ray is tested for intersection against many bounding boxes was developed by Eisemann et al. (2007); we leave implementing their method for an exercise.

The BVH traversal algorithm used in `pbrt` was concurrently developed by a number of researchers; see the notes by Boulos and Haines (2006) for more details and background. Another option for tree traversal is that of Kay and Kajiya (1986); they maintained a heap of nodes ordered by ray distance.

One shortcoming of BVHs is that even a small number of relatively large primitives that have overlapping bounding boxes can substantially reduce the efficiency of the BVH: many of the nodes of the tree will be overlapping, solely due to the overlapping bounding boxes of geometry down at the leaves. Ernst and Greiner (2007) proposed "split clipping" as a solution to this problem; the restriction that each primitive only appears once in the tree is lifted, and the bounding boxes of large input primitives are subdivided into a set of tighter sub-bounds which are then used for tree construction. This happens only during the tree construction and doesn't affect the tree representation or the rendering algorithm. Dammertz and Keller (2008) observed that the problematic primitives are the ones with a large amount of empty space in their bounding box relative to their surface area, so they subdivided the most egregious triangles and reported substantial performance improvements. Stich et al. (2009) developed an approach that splits primitives during BVH construction, rather than as a preprocess, making it possible to only split primitives when a SAH cost reduction was found. See also Popov et al.'s recent paper on a theoretically optimum BVH partitioning algorithm and its relationship to previous approaches (Popov et al. 2009).

The memory requirements for BVHs can be substantial. In our implementation, each node is 32 bytes. With up to 2 BVH tree nodes needed per primitive in the scene, the total overhead may be as high as 64 bytes per primitive. Cline et al. (2006) suggested a more compact representation for BVH nodes, at some expense of efficiency. First, they quan-

BVHAccel 209

tized the bounding box stored in each node using 8 or 16 bytes to offset with respect to the bounding box of the entire tree. Second, they used *implicit indexing*, where the node i's children are at positions $2i$ and $2i + 1$ in the node array (assuming a $2\times$ branching factor). They showed substantial memory savings, with moderate performance impact. See also Mahovsky's Ph.D. thesis (2005) for other approaches to reducing BVH memory usage.

Yoon and Manocha (2006) described algorithms for cache-efficient layout of BVHs and kd-trees and demonstrated performance improvements from doing so. See also Ericson's book (2004) for extensive discussion of this topic.

kd-trees

Glassner (1984) introduced the use of octrees for ray intersection acceleration; this approach was more robust for scenes with nonuniform distributions of geometry than grids. Use of the kd-tree was first described by Kaplan (1985). Kaplan's tree construction algorithm always split nodes down their middle; MacDonald and Booth (1990) introduced the surface area heuristic approach, estimating ray–node traversal probabilities using relative surface areas. Naylor (1993) has also written on general issues of constructing good kd-trees. Havran and Bittner (2002) revisited many of these issues and introduced useful improvements. Adding a bonus factor to the surface area heuristic for tree nodes that are completely empty, as is done in our implementation, was suggested by Hurley et al. (2002).

Jansen (1986) first developed the efficient ray traversal algorithm for kd-trees. Arvo also investigated this problem and discussed it in a note in *Ray Tracing News* (Arvo 1988). Sung and Shirley (1992) described a ray traversal algorithm's implementation for a BSP-tree accelerator; our `KdTreeAccel` traversal code is loosely based on theirs.

The asymptotic complexity of the kd-tree construction algorithm in `pbrt` is $O(n \log^2 n)$. Wald and Havran (2006) showed that it's possible to build kd-trees in $(n \log n)$ time with some additional implementation complexity; they reported a 2 to $3\times$ speedup in construction time for typical scenes.

The best kd-trees for ray tracing are built using "perfect splits," where the primitive being inserted into the tree is clipped to the bounds of the current node at each step. This eliminates the issue that, for example, an object's bounding box may intersect a node's bounding box and thus be stored in it, even though the object itself doesn't intersect the node's bounding box. This approach was introduced by Havran and Bittner (2002) and discussed further by Hurley et al. (2002) and Wald and Havran (2006). See also Soupikov et al. (2008).

kd-tree construction tends to be much slower than BVH construction (especially if "perfect splits" are used), so parallel construction algorithms are of particular interest. Recent work in this area includes that of Shevtsov et al. (2007b), who presented an efficient parallel kd-tree construction algorithm with good scalability to multiple processors.

The Surface Area Heuristic

KdTreeAccel 228

A number of researchers have investigated improvements to the surface area heuristic since its introduction to ray tracing by MacDonald and Booth (1990). Fabianowski et al. (2009) derived a version that replaces the assumption that rays are uniformly distributed

throughout space with the assumption that ray origins are uniformly distributed inside the scene's bounding box. Hunt and Mark (2008b) built specialized acceleration structures for the camera and each light, using a perspective projection to warp space as seen from the camera or light. This warping led to a new SAH that better accounts for the fact that the rays aren't in fact uniformly distributed but that a large number of them originate from a single point or a set of nearby points (for depth of field and area light sources, respectively). Hunt (2008) showed how the SAH should be modified when the "mailboxing" optimization is being used.

Evaluating the SAH can be costly, particularly when many different splits or primitive partitions are being considered. One solution to this problem is to only compute it at a subset of the candidate points—for example, along the lines of the bucketing approach used in the `BVHAccel` in `pbrt`. Hurley et al. (2002) suggested this approach for building kd-trees, and Popov et al. (2006) applied it to kd-trees. Shevtsov et al. (2007) introduced the improvement of binning the full extents of triangles, not just their centroids.

Hunt et al. (2006) noted that if you only have to evaluate the SAH at one point, for example, you don't need to sort the primitives, but only need to do a linear scan over them to compute primitve counts and bounding boxes on each point. They showed that approximating the SAH with a piecewise quadratic based on evaluating it at a number of individual positions and using that to choose a good split leads to effective trees. A similar approximation was used by Popov et al. (2006).

Other Topics in Acceleration Structures

Weghorst, Hooper, and Greenberg (1984) discussed the trade-offs of using various shapes for bounding volumes and suggested projecting objects to the screen and using a z-buffer rendering to accelerate finding intersections for camera rays.

A number of researchers have investigated the applicability of general BSP trees, where the splitting planes aren't necessarily axis aligned, as they are with kd-trees. Kammaje and Mora (2007) built BSP trees using a preselected set of candidate splitting planes, and Budge et al. (2008) developed a number of improvements to their approach, though only approached kd-tree performance in practice due to a slower construction stage and slower traversal than kd-trees. Ize et al. (2008) showed a BSP implementation that renders scenes faster than modern kd-trees, but at the cost of extremely long construction times.

There are many techniques for traversing a collection of rays through the acceleration structure together, rather than just one at a time. This approach ("packet tracing") is an important component of high-performance ray tracing; it's discussed in more depth in Section 18.2. Another major area of recent research has been acceleration structures that can be incrementally updated over frames of an animation, rather than requiring reconstruction from scratch. See, for example, Wald et al. (2007) for recent work in this area.

An innovative approach was suggested by Arvo and Kirk (1987), who introduced a five-dimensional data structure that subdivided based on both 3D spatial and 2D ray directions. Another interesting approach for scenes described with triangle meshes was developed by Lagae and Dutré (2008b): they computed a constrained tetrahedralization, where all triangle faces of the model are represented in the tetrahedralization. Rays are then stepped through tetrahedra until they intersect a triangle from the scene descrip-

BVHAccel 209

tion. This approach is still a few times slower than the state-of-the-art in kd-trees and BVHs but is an interesting new way to think about the problem.

There is an interesting middle-ground between kd-trees and BVHs, where the tree node holds a splitting plane for each child, rather than just a single splitting plane. For example, this refinement makes it possible to do object subdivision in a kd-tree-like acceleration structure, putting each primitive in just one subtree and allowing the subtrees to overlap, while still preserving many of the benefits of efficient kd-tree traversal. Ooi et al. (1987) first introduced this refinement to kd-trees for storing spatial data, naming it the "spatial kd-tree" (skd-tree). Skd-trees have recently been applied to ray tracing by a number of researchers, including Zachmann (2002), Woop et al. (2006), Wächter and Keller (2006), Havran et al. (2006), and Zuniga and Uhlmann (2006).

When spatial subdivision is used, primitives may overlap multiple nodes of the structure and a ray may be tested for intersection with the same primitive multiple times as it passes through the structure. Arnaldi, Priol, and Bouatouch (1987) and Amanatides and Woo (1987) developed the "mailboxing" technique to address this issue: each ray is given a unique integer identifier and each primitive records the id of the last ray that was tested against it. If the ids match, then the intersection test is unnecessary and can be skipped.

EXERCISES

4.1 What kind of scenes are worst-case scenarios for the three acceleration structures in `pbrt`? (Consider specific geometric configurations that the approaches will respectively be unable to handle well.) Construct scenes with these characteristics, and measure the performance of `pbrt` as you add more primitives. How does the worst case for one behave when rendered with the others?

4.2 Read the paper by Ize et al. (2007) and apply their methods for selecting grid resolution to the `GridAccel` in this chapter. Measure the trade-offs related to time spent building the grid, memory used to represent the grid, and time needed to find ray–object intersections for different grid resolutions.

4.3 Generalize the grid implementation in this chapter to be hierarchical: refine voxels that have an excessive number of primitives overlapping them to instead hold a finer subgrid to store its geometry. (See, for example, Jevans and Wyvill (1989) for one approach to this problem and Ize et al. (2007) for effective methods for deciding when refinement is worthwhile.)

4.4 Implement the compact grid representation introduced by Lagae and Dutré (2008a). How does the performance of your implementation compare to the `GridAccel` in `pbrt`? (Measure construction time, memory use, and time to find ray intersections.)

4.5 Implement smarter overlap tests for building accelerators. Using objects' bounding boxes to determine which grid cells and which sides of a kd-tree split they overlap can hurt performance by causing unnecessary intersection tests. (Recall Figure 4.5.) Add a `bool Shape::Overlaps(const BBox &) const` method

GridAccel 196

to the shape interface that takes a world space bounding box and determines if the shape truly overlaps the given bound.

A default implementation could get the world bound from the shape and use that for the test, and specialized versions could be written for frequently used shapes. Implement this method for `Spheres` and `Triangles` and modify the accelerators to call it. You may find it helpful to read Akenine-Möller's paper on fast triangle-box overlap testing (Akenine-Möller 2001). Measure the change in `pbrt`'s overall performance due to this change, separately accounting for increased time spent building the acceleration structure and reduction in ray–object intersection time due to fewer intersections. For a variety of scenes, determine how many fewer intersection tests are performed thanks to this improvement.

4.6 Implement "split clipping" in `pbrt`'s BVH implementation. Read the papers by both Ernst and Greiner (2007) and Dammertz and Keller (2008) and implement one of their approaches to subdivide primitives with large bounding boxes relative to their surface area into multiple subprimitives for tree construction. (Doing so will probably require modification to the `Shape` interface; you will probably want to design a new interface that allows some shapes to indicate that they are unable to subdivide themselves, so that you only need to implement this method for triangles, for example.) Measure the improvement for rendering actual scenes; a compelling way to gather this data is to do the experiment that Dammertz and Keller did, where a scene is rotated around an axis over progressive frames of an animation. Typically, many triangles that are originally axis aligned will have very loose bounding boxes as they rotate more, leading to a substantial performance degradation if split clipping isn't used.

4.7 Fix either the `BVHAccel` or the `KdTreeAccel` so that it doesn't always immediately refine all primitives before building the tree but instead builds subtrees on demand. Care must be taken when updating the data structures in the presence of multi-threading so that other threads don't see the tree in an inconsistent state as it is being updated. One option is to use a reader-writer mutex, as the `GridAccel` does, though the cost of acquiring the mutex for each ray is significant. More efficient is to use a lock-free approach for updating the data structure, as described in Section A.9.2.

4.8 On systems with 64-bit pointers, the `KdAccelNode` structure will actually be 12 bytes large, thanks to an 8-byte pointer for the `KdAccelNode::primitives` array. Modify the implementation to fix this problem. One approach would be to allocate all of the memory for all of the `primitives` arrays contiguously—for example, with a `vector` stored in the `KdTreeAccel`. Then, nodes of the tree that had multiple primitives would store an offset into this vector where their primitive numbers started, rather than a pointer. How much does this change affect performance in practice for reasonably complex scenes that use a kd-tree accelerator?

4.9 Investigate alternative SAH cost functions for building BVHs or kd-trees. How much can a poor cost function hurt its performance? How much improvement

`BVHAccel` 209
`GridAccel` 196
`KdAccelNode` 229
`KdAccelNode::primitives` 229
`KdTreeAccel` 228
`Shape` 108

can be had compared to the current one? (See the discussion in the "Further Reading" section for ideas about how the SAH may be improved.)

3 4.10 Construction time for the `BVHAccel` and particularly the `KdTreeAccel` can be a meaningful portion of overall rendering time, yet the implementations in this chapter do not parallelize building the acceleration structures. Investigate techniques for parallel construction of accelerators such as described by Wald (2007) and Shevtsov et al. (2007) and implement one of them in `pbrt`. How much of a speedup do you achieve in accelerator construction? How does the speedup scale with additional processors? Measure how much of a speedup your changes translate to for overall rendering. For what types of scenes does your implementation have the greatest impact?

3 4.11 The idea of using spatial data structures for ray intersection acceleration can be generalized to include spatial data structures that themselves hold other spatial data structures, rather than just primitives. Not only could we have a grid that has subgrids inside the grid cells that have many primitives in them (thus partially solving the adaptive refinement problem), but we could also have the scene organized into a hierarchical bounding volume where the leaf nodes are grids that hold smaller collections of spatially nearby primitives. Such hybrid techniques can bring the best of a variety of spatial data structure-based ray intersection acceleration methods. In `pbrt`, because both geometric primitives and intersection accelerators inherit from the `Primitive` base class and thus provide the same interface, it's easy to mix and match in this way.

Modify `pbrt` to build hybrid acceleration structures—for example, using a BVH to coarsely sort the scene geometry and then uniform grids at the leaves of the tree to manage dense, spatially local collections of geometry. Measure the running time and memory use for rendering schemes with this method compared to the current accelerators.

2 4.12 Eisemann et al. (2007) described an even more efficient ray–box intersection test than is used in the `BVHAccel`. It does more computation at the start for each ray, but makes up for this work with fewer computations to do tests for individual bounding boxes. Implement their method in `pbrt` and measure the change in rendering time for a variety of scenes. Are there simple scenes where the additional upfront work doesn't pay off? How does the improvement for highly complex scenes compare to the improvement for simpler scenes?

3 4.13 It is often possible to introduce some approximation into the computation of shadows from very complex geometry (consider, for example, the branches and leaves of a tree casting a shadow). Lacewell et al. (2008) suggested augmenting the acceleration structure with a prefiltered directionally varying representation of occlusion for regions of space. As shadow rays pass through these regions, an approximate visibility probability can be returned rather than a binary result, and the cost of tree traversal and object intersection tests is reduced. Implement this approach in `pbrt` and measure its performance.

3 4.14 The automated testing code in `AggregateTest` can be an effective way to find test cases that show a scene and a particular ray that exhibit a bug. For very complex

`AggregateTest` 247
`BVHAccel` 209
`KdTreeAccel` 228
`Primitive` 185

scenes, however, debugging these failures can be tedious; if a simpler scene could be found that exhibited the same bug, programmer time can be saved. Investigate automated techniques for finding minimal reproduction cases for bugs. For example, Zeller and Hildebrandt (2002) describe an algorithm that automatically tries to maximally simplify failure-inducing input; their method could be applied to try to remove primitives from the scene description and find a simpler scene that still had an error.

4.15 In addition to using existing scenes to test accelerators, randomly generated scenes can also be effective at finding bugs. In general, the search space for accelerator bugs is enormous, though the number of available processing cycles is also large. (Running random tests on accelerators in the background for weeks or months can be worthwhile in that bugs found in this manner are easier to track down than running into them when rendering a scene and seeing an unexpected image artifact.) Implement code that randomly generates scenes for testing. Effective approaches include building scenes from existing models, randomly transforming them to place them, perturbing existing models (for example, changing some vertex positions of a triangle mesh), or generating completely random scenes (for example, a random number of triangles, each with random vertex positions). Can you find bugs in `pbrt`'s accelerators with this technique?

4.16 Modify the system to record all rays that were traced while rendering a scene to a file and then use rays gathered in this way to test the aggregates. Modify the `AggregateTest` to read files of these rays and use them for its tests. The easiest way to collect rays is probably to modify the `Scene::Intersect()` and `IntersectP()` methods to save them. Be careful to use either the `%a` formatting string described in Section 4.6.1 for accurately writing floating-point values to a file, or write the raw bits of the `float`, for example, using `fwrite()`. Also be aware of the implications of multi-threading: if multiple threads write to the same file, their output can be interleaved. Either use a mutex to protect access to the file, or be sure to store all of the values for a ray with a single `fprintf()` or `fwrite()` call, rather than using multiple calls to write all of the components. (Implementations of the C standard library do guarantee atomicity from single calls to those functions, using a mutex internally.) Try to fix any bugs you find in aggregates.

AggregateTest 247
Scene::Intersect() 23