Physically Based Rendering读书笔记 part 3

计算机图形学

by silver_gp
原文地址：http://www.pbr-book.org/3ed-2018/contents.html

---

4. Primitives and Intersection Acceleration

上一个章节中关注点是几何体，例如交点和包围盒，但它所提供的信息并不足够，例如需要材质信息，因此这个章节引入Primitive抽象类。
直接被渲染的Shape，用GeometricPrimitive表示。
而TransformedPrimitive类引入了动画变换，以及对象实例化。

另外此章节引入了Aggregate类，是个Primitive对象的容器，在这个类中有射线检测的加速结构。
这个章节介绍了两种加速结构的实现，一个叫BVHAccel，另一种是KDTreeAccel。
现在开始。

4.1 Primitive Interface And Geometric Primitives

```
<<Primitive Declarations>>=
class Primitive {
public:
    <<Primitive Interface>>
       virtual ~Primitive();
       ///世界坐标下的包围盒
       virtual Bounds3f WorldBound() const = 0;
       ///相交测试函数
       virtual bool Intersect(const Ray &r, SurfaceInterac
tion *) const = 0;
       ///相交测试函数简化版
       virtual bool IntersectP(const Ray &r) const = 0;
       ///光源属性，自身不发光则直接返回nullptr
       virtual const AreaLight *GetAreaLight() const = 0;
       ///材质属性
       virtual const Material *GetMaterial() const = 0;
       ///初始化了光线散播属性
       virtual void ComputeScatteringFunctions(SurfaceInte
raction *isect,
           MemoryArena &arena, TransportMode mode,
           bool allowMultipleLobes) const = 0;

};
```

ComputeSctteringFunction初始化light scattering的属性，其中BSDF定义了交点的光线传播属性，而BSSRDF描述了次表面光线传输相关的信息，专门用于描述皮肤，牛奶等材质。除了用MemoryArena类给BSDF/BSSRDF分配内存外，它的TransportMode指明了到达交点的射线是来自于摄像机，还是来自于光源。参数allowMultipleLobes控制了某些类型的BRDF的显示细节。

```
<<SurfaceInteraction Public Data>>+=
BSDF *bsdf = nullptr;
BSSRDF *bssrdf = nullptr;
```

BSDF和BSSRDF对象是通过SurfaceInteraction对象传递给ComputeScatteringFUnctions()函数的。

4.1.1 Geometric Primitives

GeometricPrimitive类描述了一个场景中的形状Shape，在场景中，为每个形状分配了一个GeometricPrimitive对象。

```
<<GeometricPrimitive Declarations>>=
class GeometricPrimitive : public Primitive {
public:
    <<GeometricPrimitive Public Methods>>
       virtual Bounds3f WorldBound() const;
       virtual bool Intersect(const Ray &r, SurfaceInterac
tion *isect) const;
       virtual bool IntersectP(const Ray &r) const;
       GeometricPrimitive(const std::shared_ptr<Shape> &sh
ape,
                  const std::shared_ptr<Material> &material,
                  const std::shared_ptr<AreaLight> &areaLigh
t,
                  const MediumInterface &mediumInterface)
           : shape(shape), material(material), areaLight(a
reaLight),
             mediumInterface(mediumInterface) {
       }
       const AreaLight *GetAreaLight() const;
       const Material *GetMaterial() const;
       void ComputeScatteringFunctions(SurfaceInteraction
 *isect, MemoryArena &arena,
           TransportMode mode, bool allowMultipleLobes) co
nst;

private:
    <<GeometricPrimitive Private Data>>
        ///形状
```

```
        std::shared_ptr<Shape> shape;
        ///材质
        std::shared_ptr<Material> material;
        ///自发光的话，光源信息
        std::shared_ptr<AreaLight> areaLight;
        ///参与介质
        MediumInterface mediumInterface;

};
```

这里面大多数定义的方法，就是直接调用shape的对应方法。

```
<<GeometricPrimitive Method Definitions>>=
bool GeometricPrimitive::Intersect(const Ray &r,
        SurfaceInteraction *isect) const {
    Float tHit;
    if (!shape->Intersect(r, &tHit, isect))
        return false;
    //每执行一次，都把tMax缩短一点，最后得到最近的那个
    r.tMax = tHit;
    isect->primitive = this;
    <<Initialize SurfaceInteraction::mediumInterface after
 Shape intersection>>
       if (mediumInterface.IsMediumTransition())
           isect->mediumInterface = mediumInterface;
       else
           isect->mediumInterface = MediumInterface(r.medi
um);

    return true;
}
<<GeometricPrimitive Method Definitions>>+=
void GeometricPrimitive::ComputeScatteringFunctions(
        SurfaceInteraction *isect, MemoryArena &arena, Tra
nsportMode mode,
        bool allowMultipleLobes) const {
    //这里直接调用material的对应方法
    if (material)
        material->ComputeScatteringFunctions(isect, arena,
 mode,
            allowMultipleLobes);
```

```
}
```

4.1.2 TransformedPrimitive: Object Instancing and Animated Primitives

TransformedPrimitive持有Primitive对象和AnimatedTransform对象，这个额外的transform是的它有两个有用的特性，对象实例化和图元的动画变换。
对象实例化能让不同变换姿态的对象共享一份mesh，这对于节省内存非常有帮助。
AnimatedTransform允许rigid-body动画。
之前讲到Shape拥有一个本地空间到世界空间变换的transform，但如果一个TransformedPrimitive持有一个Shape，那么这个transform就不是到世界空间的了，只有应用了TransformedPrimitive的变换，才真正转换到世界空间中。从设计上讲，Shape类不需要知道所有它的实例。

```
<<TransformedPrimitive Declarations>>=
class TransformedPrimitive : public Primitive {
public:
    <<TransformedPrimitive Public Methods>>
       ///primitive to world变换定义了当前实例local space 到
 world space的变换，如果primitive本身有变换，则要把变换组合到一起
       TransformedPrimitive(std::shared_ptr<Primitive> &pr
imitive,
                const AnimatedTransform &PrimitiveToWorld)
           : primitive(primitive), PrimitiveToWorld(Primit
iveToWorld) { }
       bool Intersect(const Ray &r, SurfaceInteraction *i
n) const;
       bool IntersectP(const Ray &r) const;
       const AreaLight *GetAreaLight() const { return null
ptr; }
       const Material *GetMaterial() const { return nullpt
r; }
       void ComputeScatteringFunctions(SurfaceInteraction
 *isect, MemoryArena &arena,
           TransportMode mode, bool allowMultipleLobes) co
nst {
           Severe("TransformedPrimitive::ComputeScattering
Functions() shouldn't be called");
       }
       Bounds3f WorldBound() const {
```

```
            return PrimitiveToWorld.MotionBounds(primitive-
>WorldBound());
        }

private:
    <<TransformedPrimitive Private Data>>
        ///primitive数据
        std::shared_ptr<Primitive> primitive;
        const AnimatedTransform PrimitiveToWorld;

};
```

TransformedPrimitive类持有Primitive对象，如果需要多个Primitive共同组合成一个几何体的话，就需要把它们放到Aggregate类当中。TransformedPrimitive负责桥接Primitive和对应的动画变换。
TransformedPrimitive::Intersect()方法将射线变换到primitive坐标系下，如果有相交，就把primitive坐标系下的tMax复制给外部要返回的ray的tMax

```
<<TransformedPrimitive Method Definitions>>=
bool TransformedPrimitive::Intersect(const Ray &r,
        SurfaceInteraction *isect) const {
    <<Compute ray after transformation by PrimitiveToWorld
>>
        Transform InterpolatedPrimToWorld;
        //首先用时间采样PrimitiveToWorld的变换过程中的某一点的变
换
        PrimitiveToWorld.Interpolate(r.time, &InterpolatedP
rimToWorld);
        //然后执行逆变换，将世界空间的射线变换到TransformedPrimit
ive空间下
        Ray ray = Inverse(InterpolatedPrimToWorld)(r);

    if (!primitive->Intersect(ray, isect))
        return false;
    r.tMax = ray.tMax;
    //最后再将相交的结果变换回世界空间
    <<Transform instance's intersection data to world spac
e>>
        if (!InterpolatedPrimToWorld.IsIdentity())
            *isect = InterpolatedPrimToWorld(*isect);
```

```
    return true;
}
```

最后要介绍的，就是WorldBound()函数，其实也就是直接调用primitive对应的函数，然后再进行一次之前介绍过的运动相关的变换。

```
<<TransformedPrimitive Public Methods>>+=
Bounds3f WorldBound() const {
    return PrimitiveToWorld.MotionBounds(primitive->WorldB
ound());
}
```

另外，TransformedPrimitive GetAreaLight(), GetMaterial(),和ComputeScatteringFunctions()永远都不会被调用到，因为当射线真的击中的时候，应该调用primitive中对应的方法，而不是这个类的。

4.2 Aggregates

在射线跟踪过程中，加速结构非常重要，否则的话，射线需要对每个物体一一作检测。加速结构涉及到两个主要议题：空间分割，和物体分割。空间分割是通过将3D空间分成不同的区域，然后找出哪些primitive在这些空间区域中。射线检测执行的时候，只去检查射线划过的空间区域内的primitive。而物体分割是把一个物体进行细分，分成不同的子mesh。如果一个射线根本没有与物体外层包围盒相交，则根本不用去考虑射线和物体内部碰撞。
我们接下来会同时使用这两种方法，KdTreeAccel用于空间分割，而BVHAccel用于物体分割。
Aggregate类聚合了Primitive类的对象，同时Aggregate从Primitive派生。Integrator类可以对单独的Primitive很好的工作，同时也可以用Aggregate类来试验新的加速算法。

```
<<Aggregate Declarations>>=
class Aggregate : public Primitive {
public:
    <<Aggregate Public Methods>>
        //以下3个函数不能被直接调用，以为Aggregate本身就是个集合
的概念，调用它并没有什么意义
        const AreaLight *GetAreaLight() const;
        const Material *GetMaterial() const;
        void ComputeScatteringFunctions(SurfaceInteraction
 *isect,
```

```
            MemoryArena &arena, TransportMode mode, bool al
lowMultipleLobes) const;

};
```

4.3 Bounding Volume Hierarchies

Bounding Volume Hierarchies(BVHs)是一个用于加速射线相交检测的方法，它基于primitive分割。它将primitive分解为树状的不相交的部分(与此对应的是，空间分割算法通常都将空间分割成树状的不相交的部分)。

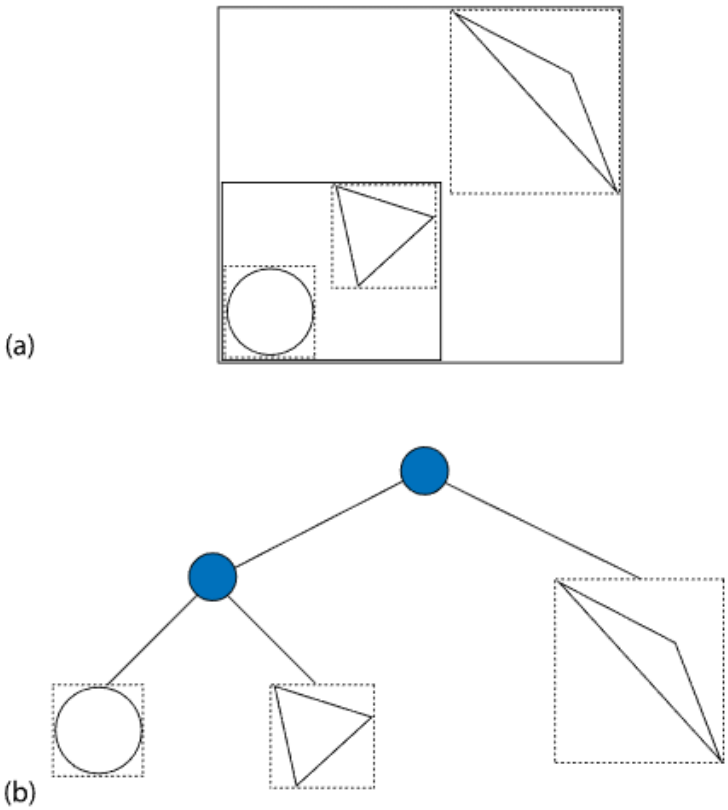


Figure 4.2: Bounding Volume Hierarchy for a Simple Scene. (a) A small collection of primitives, with bounding boxes shown by dashed lines. The primitives are aggregated based on proximity; here, the sphere and the equilateral triangle are bounded by another bounding box before being bounded by a bounding 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.

图元(primitive)分割的特点是，每一个图元仅仅在层次结构中出现一次，但是一个图元也许会在多个不重叠的空间分割区域内，因此在射线检测的时候，很有可能被测试多次。另外用于描述这个分割的内存空间是确定的，对于一个二叉树BVH来讲，每一个单独的图元都用一个叶子节点表示，那么总节点数量就是$2n-1$，其中n是图元数量。如果允许一个叶子节点保存多个图元的话，用的节点数量会更少。

BVHAccel类封装了加速算法，构造函数中传入的枚举型参数代表构造BVH树用的算法。

其中SAH指的是”Surface Area Heuristic”，HLBVH算法则更为有效，这些算法后面会介绍。

```
<<BVHAccel Public Types>>=
enum class SplitMethod { SAH, HLBVH, Middle, EqualCounts
 };
<<BVHAccel Method Definitions>>=
BVHAccel::BVHAccel(const std::vector<std::shared_ptr<Primi
tive>> &p,
         int maxPrimsInNode, SplitMethod splitMethod)
     : maxPrimsInNode(std::min(255, maxPrimsInNode)), prim
itives(p),
       splitMethod(splitMethod) {
    if (primitives.size() == 0)
        return;
    <<Build BVH from primitives>>
       <<Initialize primitiveInfo array for primitives>>
          std::vector<BVHPrimitiveInfo> primitiveInfo(prim
itives.size());
          for (size_t i = 0; i < primitives.size(); ++i)
              primitiveInfo[i] = { i, primitives[i]->World
Bound() };

       <<Build BVH tree for primitives using primitiveInfo
>>
          MemoryArena arena(1024 * 1024);
          int totalNodes = 0;
          std::vector<std::shared_ptr<Primitive>> orderedP
rims;
          BVHBuildNode *root;
          if (splitMethod == SplitMethod::HLBVH)
              root = HLBVHBuild(arena, primitiveInfo, &tot
alNodes, orderedPrims);
          else
              root = recursiveBuild(arena, primitiveInfo,
0, primitives.size(),
                                    &totalNodes, orderedPr
ims);
          primitives.swap(orderedPrims);
```

```
        <<Compute representation of depth-first traversal o
f BVH tree>>
           nodes = AllocAligned<LinearBVHNode>(totalNodes);
           int offset = 0;
           flattenBVHTree(root, &offset);


}
<<BVHAccel Private Data>>=
const int maxPrimsInNode;
const SplitMethod splitMethod;
std::vector<std::shared_ptr<Primitive>> primitives;
```

4.3.1 BVH Construction

构造BVH有3个阶段。首先每个图元的包围盒信息会首先被构造出来，然后存储到数组中；接下来，通过splitMethod指定的算法将树构造出来，得到一棵二叉树，其内部节点持有它所有子的指针，叶子节点持有一个或者多个图元；最后，将这棵二叉树转化为更紧缩的格式，不再使用指针。

```
<<Build BVH from primitives>>=
<<Initialize primitiveInfo array for primitives>>
   //首先，将图元列表中每个图元的包围盒存储在数组中
   std::vector<BVHPrimitiveInfo> primitiveInfo(primitives.
size());
   for (size_t i = 0; i < primitives.size(); ++i)
       primitiveInfo[i] = { i, primitives[i]->WorldBound()
 };

<<Build BVH tree for primitives using primitiveInfo>>
   MemoryArena arena(1024 * 1024);
   int totalNodes = 0;
   std::vector<std::shared_ptr<Primitive>> orderedPrims;
   BVHBuildNode *root;
   if (splitMethod == SplitMethod::HLBVH)
       //通过HLBVH算法生成的树
       root = HLBVHBuild(arena, primitiveInfo, &totalNode
s, orderedPrims);
   else
```

```
        //通过其他算法生成的树，同时还允许传入primitives的begin和
end范围，使其只对一部分primitive生成bvh树
        //在内部将primitives分成两个部分以后再分别对这两个部分递归
调用，totalNodes返回全部bvhnode
        root = recursiveBuild(arena, primitiveInfo, 0, prim
itives.size(),
                              &totalNodes, orderedPrims);
    //生成树的算法会返回一个经过排序的primitive列表
    primitives.swap(orderedPrims);


<<Compute representation of depth-first traversal of BVH t
ree>>
    nodes = AllocAligned<LinearBVHNode>(totalNodes);
    int offset = 0;
    flattenBVHTree(root, &offset);

For each primitive to be stored in the BVH, we store the c
entroid of its bounding box, its complete bounding box, an
d its index in the primitives array in an instance of the
 BVHPrimitiveInfo structure.

<<Initialize primitiveInfo array for primitives>>=
std::vector<BVHPrimitiveInfo> primitiveInfo(primitives.siz
e());
for (size_t i = 0; i < primitives.size(); ++i)
    primitiveInfo[i] = { i, primitives[i]->WorldBound() };


<<BVHAccel Local Declarations>>=
struct BVHPrimitiveInfo {
    BVHPrimitiveInfo(size_t primitiveNumber, const Bounds3
f &bounds)
        : primitiveNumber(primitiveNumber), bounds(bound
s),
          centroid(.5f * bounds.pMin + .5f * bounds.pMax)
 { }
    ///图元索引
    size_t primitiveNumber;
    ///包围盒
    Bounds3f bounds;
```

```
    ///质心，实际上就是包围盒的中心而已
    Point3f centroid;
};
```

BVHBuildNode就是BVH的节点

```
<<BVHAccel Local Declarations>>+=
struct BVHBuildNode {
    <<BVHBuildNode Public Methods>>
        ///作为叶子节点初始化
       void InitLeaf(int first, int n, const Bounds3f &b)
 {
           firstPrimOffset = first;
           nPrimitives = n;
           bounds = b;
           //两个子都是空的，就是叶子节点
           children[0] = children[1] = nullptr;
       }
       ///作为内部节点初始化
       void InitInterior(int axis, BVHBuildNode *c0, BVHBu
ildNode *c1) {
           children[0] = c0;
           children[1] = c1;
           bounds = Union(c0->bounds, c1->bounds);
           splitAxis = axis;
           nPrimitives = 0;
       }

    Bounds3f bounds;
    //如果两个子都是nullptr，那么当前节点就是叶子节点
    BVHBuildNode *children[2];
    //splitAxis:将当前空间分割为两个子空间的分割方向，因为一个叶子
节点可能包含多个primitive，因此这里的offset
    //就是相对于BVHAccel::primitives数组的偏移，以及primitive
的数量
    int splitAxis, firstPrimOffset, nPrimitives;
};
```

recursiveBuild()函数接受MemoryAreana用来分配内存，以及BVHPrimitiveInfo结构，用于存储图元信息，以及[start,end)区间，来取primitiveInfo[start]到primitive[end-1]范围内

的图元数据。如果只包含了一个图元数据，则直接生成叶子节点，否则的话，就会将图元数组进行分割，分成[start,mid)以及[mid,end)两个部分，如果分割算法成功了，则会递归地将两个部分继续分割。
totalNodes返回所有创建出来的BVHNode的数量，因为后面还要把这些BVHNodes放到LinearBVHNodes数组中。最后orderedPrims数组用于存储图元引用，当一个叶子节点创建之后，recursiveBuild()函数将这个叶子节点所包含的图元放到数组末尾，使得叶子节点仅仅储存在数组中的偏移以及图元的数量就够了。当构建完毕后，BVHAccel::primitives就会被经过排序的图元数组覆盖。

```
<<BVHAccel Method Definitions>>+=
BVHBuildNode *BVHAccel::recursiveBuild(MemoryArena &arena,
        std::vector<BVHPrimitiveInfo> &primitiveInfo, int
start,
        int end, int *totalNodes,
        std::vector<std::shared_ptr<Primitive>> &orderedPr
ims) {
    BVHBuildNode *node = arena.Alloc<BVHBuildNode>();
    (*totalNodes)++;
    <<Compute bounds of all primitives in BVH node>>
    int nPrimitives = end - start;
    if (nPrimitives == 1) {
        <<Create leaf BVHBuildNode>>
    } else {
        <<Compute bound of primitive centroids, choose spl
it dimension dim>>
        <<Partition primitives into two sets and build chi
ldren>>
    }
    return node;
}
<<Compute bounds of all primitives in BVH node>>=
Bounds3f bounds;
for (int i = start; i < end; ++i)
    bounds = Union(bounds, primitiveInfo[i].bounds);
```

构建叶子节点，把叶子节点，叶子节点中所有的图元都会被放入orderedPrims数组中

```
<<Create leaf BVHBuildNode>>=
int firstPrimOffset = orderedPrims.size();
for (int i = start; i < end; ++i) {
```

```
        int primNum = primitiveInfo[i].primitiveNumber;
        orderedPrims.push_back(primitives[primNum]);
    }
    node->InitLeaf(firstPrimOffset, nPrimitives, bounds);
    return node;
```

而对于内部节点，需要把所有图元分割成两个子树，给定n个图元，则通常有$2^n - 2$种划分方法，在构建BVH节点时，一种方法是沿着坐标轴去划分，这意味着会有$6n$种可能的分割方式，对于每个轴，都可以把一个图元划分到一组或者另一组，一共3根轴，因此对于每个图元一共是6种可能。
我们只选择三根坐标轴的一根用于分割图元。选择标准为：使用当前图元（或者图元集合）的包围盒的中心点，取坐标最大的那个轴。

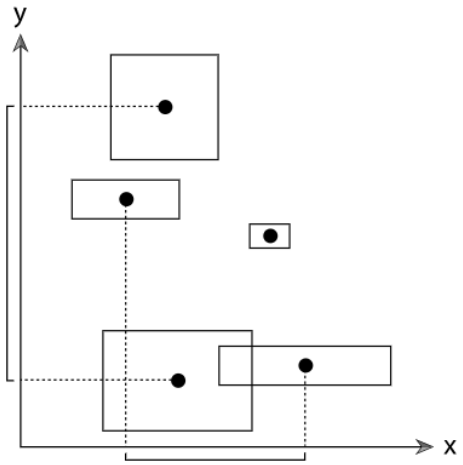


Figure 4.3: Choosing the Axis along Which to Partition Primitives. The `BVHAccel` chooses an axis along which to partition the primitives based on which axis has the largest range of the centroids of the primitives' bounding boxes. Here, in two dimensions, their extent is largest along the y axis (filled points on the axes), so the primitives will be partitioned in y.

选择分割图元方式的原则就是最终结果要让两组图元的包围盒尽量少的重叠，如果有大量的重叠，在遍历树的时候就要遍历两个子树。

```
    <<Compute bound of primitive centroids, choose split dimen
    sion dim>>=
    Bounds3f centroidBounds;
    for (int i = start; i < end; ++i)
        centroidBounds = Union(centroidBounds, primitiveInfo
    [i].centroid);
    int dim = centroidBounds.MaximumExtent();
```

如果所有中心点都在同一个位置（例如中心包围盒的体积为0），那么递归调用停止，构建叶子节点。这是种特殊情况，任何分割方法都没有用。除了这种情况，图元会继续用选择的方法进行分割。

```
<<Partition primitives into two sets and build children>>=

int mid = (start + end) / 2;
if (centroidBounds.pMax[dim] == centroidBounds.pMin[dim])
 {
    <<Create leaf BVHBuildNode>>
} else {
    <<Partition primitives based on splitMethod>>
    node->InitInterior(dim,
                       recursiveBuild(arena, primitiveInf
o, start, mid,
                                      totalNodes, orderedP
rims),
                       recursiveBuild(arena, primitiveInf
o, mid, end,
                                      totalNodes, orderedP
rims));
}
```

其中代码片段Partition primitives based on splitMethod，使用了BVHAccel::splitMethod去决定用那个算法去分割图元。
一个简单的splitMethod是Middle，首先分割轴的计算图元包围盒中心点的中点。图元因此被分成了两组，根据包围盒中心点是否大于或小于中心点。在实现，可以使用std::partition()函数，这个函数实际上是个排序函数，比较函数返回true的元素，在返回false元素之前，并且返回值是第一个false元素的指针。那么代码就一目了然了：

```
<<Partition primitives through node's midpoint>>=
Float pmid = (centroidBounds.pMin[dim] + centroidBounds.pM
ax[dim]) / 2;
BVHPrimitiveInfo *midPtr =
    std::partition(&primitiveInfo[start], &primitiveInfo[e
nd-1]+1,
        [dim, pmid](const BVHPrimitiveInfo &pi) {
            return pi.centroid[dim] < pmid;
        });
mid = midPtr - &primitiveInfo[0];
```

```
    if (mid != start && mid != end)
        break;
```

记住，这里的代码是为了通过排序，将primitiveInfo分割成两段，以配合后面构建左右子树的操作。这个mid并不是指的是真正的中间，而是左右子树的分割点。primitiveInfo以及start,end是需要构造子树的所有图元

splitMethod还可以是SplitMethod::EqualCounts，代码片段Partition primitives into equally sized subsets就会开始执行。顾名思义，这个算法会降图元列表分割成两个数量相同的的子集，有n个元素的图元，沿着选定的分割轴，n/2个元素的包围盒中心坐标会小一些，另外n/2个会大一些。大多数情况下这个算法都可以正常工作，但是下图这个情况就不太正常。

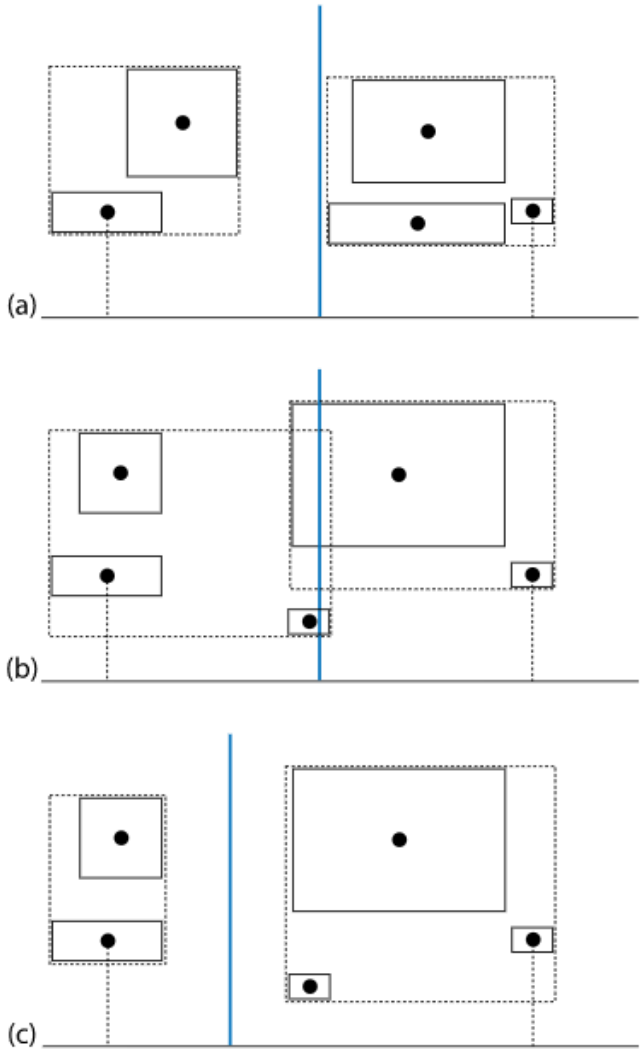


Figure 4.4: Splitting Primitives Based on the Midpoint of Centroids on an Axis. (a) For some distributions of primitives, such as the one shown here, splitting based on the midpoint of the centroids along the chosen axis (thick blue line) works well. (The bounding boxes of the two resulting primitive 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.

这个算法通过std::nth_element()函数就可以做到。这个函数用于元素进行二分排序，把

某个特定位置的元素排正确，排序结果是，middle前面的都比它小，后面的都比它大，或者反过来。但两个部分内部不一定有序。

```
<<Partition primitives into equally sized subsets>>=
mid = (start + end) / 2;
std::nth_element(&primitiveInfo[start], &primitiveInfo[mi
d],
                 &primitiveInfo[end-1]+1,
    [dim](const BVHPrimitiveInfo &a, const BVHPrimitiveInf
o &b) {
        return a.centroid[dim] < b.centroid[dim];
    });
```

4.3.2 The Surface Area Heuristic

上面两种算法实际运用的时候效果并没有那么好，因此更为常用的方法是被称为”表面面积启发”(Surface Area Heuristic, SAH)的算法，这个算法考虑到了遍历树的开销，因此才用了最小化开销的原则来构建树。通常情况下，通过贪心算法来构建每一个节点。
SAH的算法思想很直接：在构建适应性加速结构的某一时刻，我们能为当前的区域或者几何体构建一个节点，在这种情况下，任何一个穿过这个区域的射线需要进行的射线几何体相交测试需要对所有几何体进行检测，这带来的开销为：

$$\sum_{i-1}^{N} t_{isect}(i)$$

其中，N为图元数量，$t_{isect}(i)$为是第i个图元进行射线相交测试的开销。
而另一个选择就是进行空间分割。在此情况下，开销变为

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

其中，t_{trav}是遍历内部节点，以及决定到底遍历左右子树的时间，p_A和p_B是进入左右子树的概率，a_i和b_i是左右子树的图元索引，N_A和N_B是覆盖左右两个子树的图元数量。
如何分割图元直接影响到了上述的两个概率以及两边的图元都有谁。
在pbrt中，我们可以简单假设$t_{isect}(i)$是常量时间。这也意味着可能需要返回计算一个射线几何体检测需要用到多少个cpu时钟周期。
p_A和p_B的计算可以用计算几何概率的思想。假设有一个凸体A和一个凸体B，凸体A被凸体B包围，假设有一条均匀分布的射线，求射线在通过凸体B的条件下还通过了凸体A的条件概率为：

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

其中，s_A是凸体A的面积，s_B是凸体B的面积。
因此，对于空间A内的子空间B和C，穿过A的射线同时也穿过B或C的概率实际上是非常容易求解的。

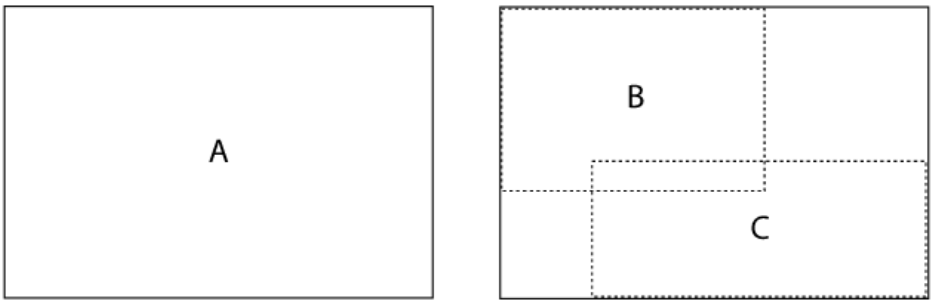


Figure 4.5: If a node of the bounding hierarchy with surface area s_A is split into two children with surface areas s_B and s_C, the probabilities that a ray passing through A also passes through B and C are given by s_B/s_A and s_C/s_A, respectively.

当splitMethod被设置为了SplitMethod::SAH，则开始使用SAH算法来构建BVH。通过考虑各种分割方式，最终要找到有着最小SAH开销估计的沿着选择的坐标轴的分割方式。然而当已经有一群小图元之后，实现方式就需要改成等数量的分割方式，此时去增量地计算开支就不那么划算了。

```
<<Partition primitives using approximate SAH>>=
if (nPrimitives <= 4) {
    <<Partition primitives into equally sized subsets>>
} else {
    <<Allocate BucketInfo for SAH partition buckets>>
    <<Initialize BucketInfo for SAH partition buckets>>
    <<Compute costs for splitting after each bucket>>
    <<Find bucket to split at that minimizes SAH metric>>
    <<Either create leaf or split primitives at selected S
AH bucket>>
}
```

SAH方法是沿着坐标轴，分割成一系列等长度的桶，然后考虑基于桶的范围的划分。

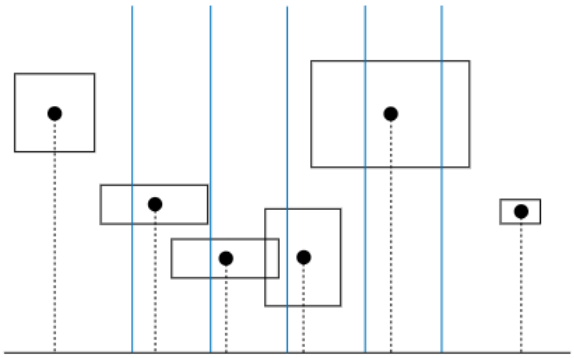


Figure 4.6: Choosing a Splitting Plane with the Surface Area Heuristic for BVHs. The projected extent of primitive bounds centroids is projected onto the chosen split axis. Each primitive is placed in a bucket along the axis based on the centroid of its bounds. The implementation then estimates the cost for splitting the primitives along the planes along each of the bucket boundaries (solid blue lines); whichever one gives the minimum cost per the surface area heuristic is selected.

```
<<Allocate BucketInfo for SAH partition buckets>>=
constexpr int nBuckets = 12;
struct BucketInfo {
    int count = 0;
    Bounds3f bounds;
};
BucketInfo buckets[nBuckets];
```

对于区间内的每个图元，首先决定的是包围盒中心点落在了哪个桶里，然后扩展桶的范围用以装下这个图元。

```
<<Initialize BucketInfo for SAH partition buckets>>=
for (int i = start; i < end; ++i) {
    int b = nBuckets *
        centroidBounds.Offset(primitiveInfo[i].centroid)[d
im];
    if (b == nBuckets) b = nBuckets - 1;
    //记录每个桶内会有的图元的数量，以及扩展桶的包围盒，注意这里并没
有把图元直接放在bucket这个数据结构中，只在primitiveInfo中存储
    buckets[b].count++;
    buckets[b].bounds = Union(buckets[b].bounds, primitive
Info[i].bounds);
}
```

对于每个桶，我们有装入其中的一系列图元，以及它们所有的包围盒范围信息。我们需要用SAH去估算切割每个桶的范围的时间开销。这里我们设定计算相交的开销为1，遍

历的开销是1/8。原因在于计算节点遍历（本质上是射线包围盒检测计算来决定走左子树还是右子树）的计算量，仅仅比射线与shape的检测低那么一点点，但射线与图元检测需要通过两个虚函数调用，这里会出现比较大的开销，因此我们估计这里的开销大概是射线包围盒检测的8倍左右。

```
<<Compute costs for splitting after each bucket>>=
Float cost[nBuckets - 1];
for (int i = 0; i < nBuckets - 1; ++i) {
    Bounds3f 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;
    }
    //这里的b0.SurfacreArea()就是这个子树的面积，这个计算实际上是
计算开销的数学期望
    cost[i] = .125f + (count0 * b0.SurfaceArea() +
                       count1 * b1.SurfaceArea()) / bound
s.SurfaceArea();
}
```

得到所有的cost，找到所有cost中最小的切分方法

```
<<Find bucket to split at that minimizes SAH metric>>=
Float minCost = cost[0];
int minCostSplitBucket = 0;
for (int i = 1; i < nBuckets - 1; ++i) {
    if (cost[i] < minCost) {
        minCost = cost[i];
        minCostSplitBucket = i;
    }
}
```

如果选中的分割方式的开销比用已有的图元构建一个节点的开销要低，或者一个节点所拥有的的图元已经超过了最大允许的数量，则需要使用std::partition()函数来记录在primitiveInfo数组中节点的信息。之前说了std::partition()函数将比较函数返回为真的元素

放前面，假的元素放后面，并且返回首个假元素的位置。因为我们之前假设了计算一次相交的开支是1，因此创建一个叶子节点的开支约等于里面图元的数量：nPrimitives。

```
<<Either create leaf or split primitives at selected SAH b
ucket>>=
Float leafCost = nPrimitives;
if (nPrimitives > maxPrimsInNode || minCost < leafCost) {
    //这里根据最优分割的桶，将primitiveInfo数组的内容重新排序
    BVHPrimitiveInfo *pmid = std::partition(&primitiveInfo
[start],
        &primitiveInfo[end-1]+1,
        [=](const BVHPrimitiveInfo &pi) {
            //对应第几个桶
            int b = nBuckets * centroidBounds.Offset(pi.ce
ntroid)[dim];
            //超过了最后一个桶，算到最后一个桶里，这种情况应该发生
在当前图元的包围盒中心在整个的包围盒的外面
            if (b == nBuckets) b = nBuckets - 1;
            //所在桶是否小于等于最优切割的那个桶，如果是的话，就在
那个切割的桶内或者左侧，否则在右侧
            return b <= minCostSplitBucket;
        });
    mid = pmid - &primitiveInfo[0];
} else {
    <<Create leaf BVHBuildNode>>
}
```

总结一下，上面的代码的核心功能是给你primitiveInfo数组，这个数组里是你能拿到的所有图元，要把这些图元分割成两个子树，每个子树都进行递归分割。无论mid，equal size还是SAH，都是为了确定分割的策略。分割的结果是将primitiveInfo数组重新排序，并给出分割点索引mid，使得mid左边进左子树，右边进右子树。分割方式可以是两边图元数量相同，也可以是分割线恰好在所有图元的中心点，也可以是通过开销估计来平衡两边。之后就是左子树和右子树的递归构建。另外，很明显的，分割操作是以每个primitive的包围盒中心点来进行分类的，很显然地，这并不意味着两个子树不重叠。

4.3.3 Linear Bounding Volume Hierarchies

上面构造BVH的方法有两个缺陷：

1. 需要对场景中所有级别的树计算SAH开销，而这需要使用很多遍循环

2. 自上而下的BVH计算难以并行，毕竟要有足够可分割的子树，必须至少是前几层的树已经构建完毕才有可能做到。

因此**线性包围盒结构(Linear Bounding Volume Hierarchies, LBVH)**就被发明了出来。这个算法的厉害之处在于构造树只需要几个轻量级的用于处理图元的pass，并且构造时间相对于图元数量是线性的，算法能迅速将图元分割成**簇(clusters)**，以便于并行处理。这个算法的关键思想是将构造BVH的过程转变成为一个排序问题。因为并没有一个现成的函数能对多维数据进行排序，因此需要用到**莫顿码Morton codes**，它可以将n维空间的相邻点映射到1维空间的相邻点，这样排序算法就能派上用场了。
莫顿码基于一个简单的转换：给你n维空间的整型坐标值，莫顿码的表示方法是将坐标的2进制交错表示每个bit。例如一个2D坐标(x,y)其中x和y的2进制比特的每一位表示为x_i和y_i，因此莫顿码的值为：

$$\ldots y_3 x_3 y_2 x_2 y_1 x_1 y_0 x_0$$

如图所示，

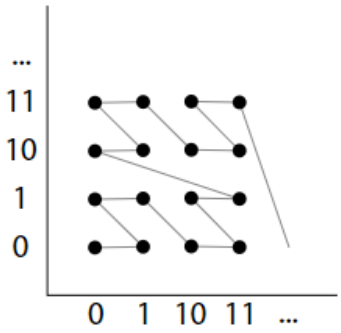
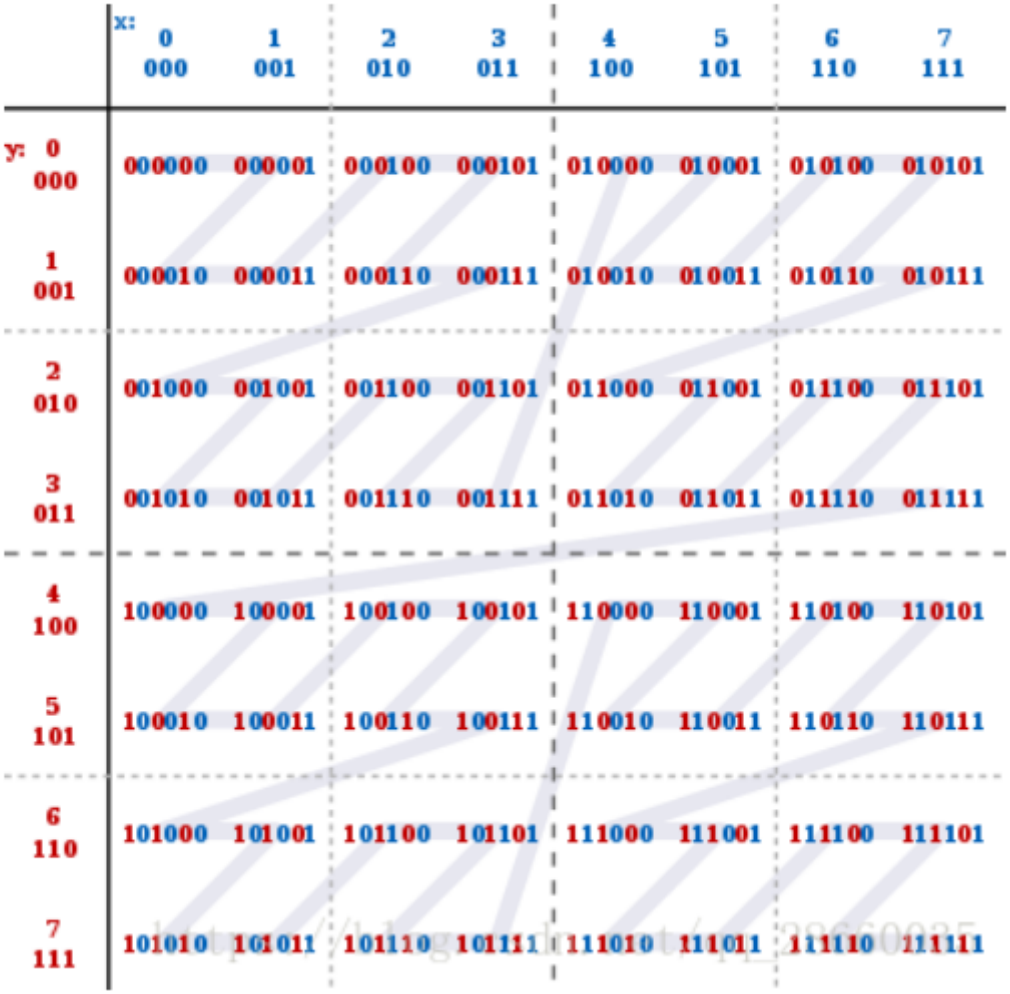


Figure 4.7: The Order That Points Are Visited along the Morton Curve. Coordinate values along the x and y axes are shown in binary. If we connect the integer coordinate points in the order of their Morton indices, we see that the Morton curve visits the points along a hierarchical "z"-shaped path.

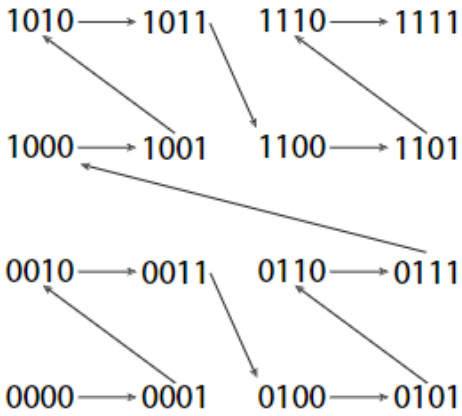
这张图更清楚点

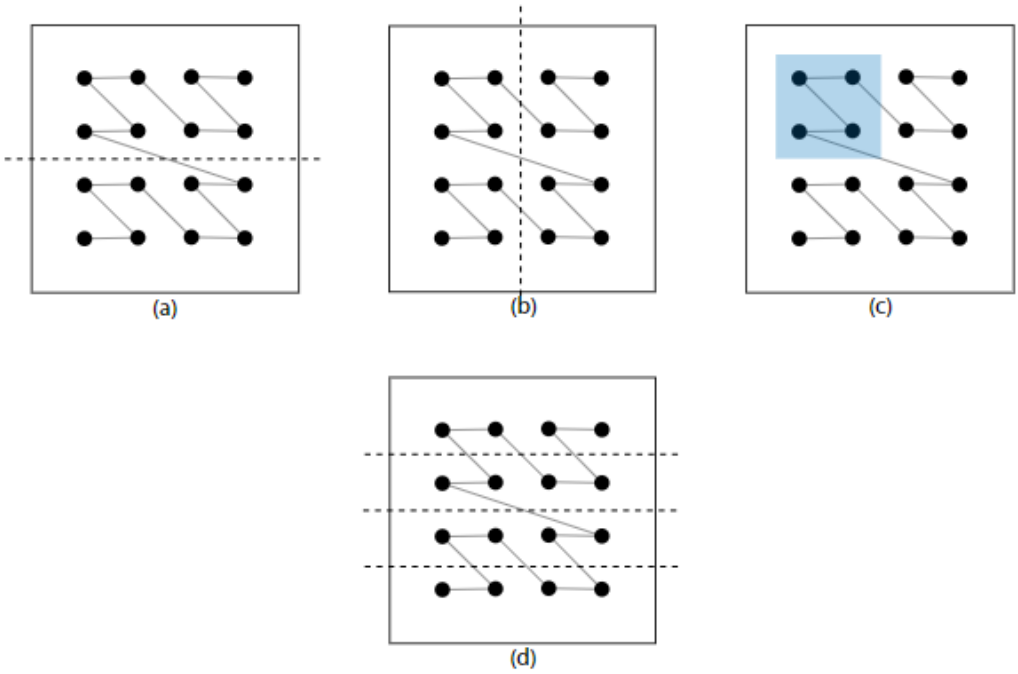


y: \ x:	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

这里的z形曲线又叫做**莫顿曲线(Morton curve)**。
举个例子，坐标(5,5)，其编码为110011b，对应十进制为51，坐标(6,6)，其编码为111100b，对应十进制为60，坐标(5,0)，其编码为010001b，对应十进制为17，直观地讲，就是坐标挨的越近的，编码的值也越接近。
莫顿码也可以编码进去一些关于它要表示的点的位置的非常有用的信息 。考虑一对4bit的2D坐标，x和y坐标的范围是[0,15]，莫顿码有8个bit：$y_3x_3y_2x_2y_1x_1y_0x_0$。在编码过程中，有很多有趣的属性：

- 莫顿码的8个bit中，高位的y_3一旦设置，我们则可以肯定是y坐标对应的最高位被设置了，因此则有$y \geq 8$(图例4.8(a))。同时也能发现这个bit实际上将图分割成了上下两边
- 接下来的一个bit，x_3将x分割为左右两边，这个bit为1时，点落在右半边，否则落在左半边(图例4.8(b))
- 接下来的y_2，把y轴分割成了4个部分(图例4.8(d))





另一种解读莫顿码基于bit的特性是基于它的值。例如图例4.8(a)的上半部分的范围就是[8,15]，而4.8(c)的范围是[8,11]。因此给定一组莫顿码的索引，我们通过二分法就能找到对应区域的点的范围。

LBVH是一种BVH，它用在空间区域中点的分割平面来分割图元（等价于之前用的SplitMethod::Middle方法）。分割非常有效，基于莫顿码。

我们现在开始构建**层级线性包围盒层级(hierarchical linear bounding volume hierarchy, HLBVH)**(这啥名字)。在这个方法中，基于莫顿曲线的簇用于构建树的底层的层级，在这里被称为”treelet”，而树的上层，则使用SAH方法。HLBVHBuild()方法则用于执行整个构建过程。

```
<<BVHAccel Method Definitions>>+=
BVHBuildNode *BVHAccel::HLBVHBuild(MemoryArena &arena,
        const std::vector<BVHPrimitiveInfo> &primitiveInf
o,
        int *totalNodes,
```

```
        std::vector<std::shared_ptr<Primitive>> &orderedPr
ims) const {
    <<Compute bounding box of all primitive centroids>>
    <<Compute Morton indices of primitives>>
    <<Radix sort primitive Morton indices>>
    <<Create LBVH treelets at bottom of BVH>>
    <<Create and return SAH BVH from LBVH treelets>>
}
```

前面也说过，构建BVH只用了图元包围盒的中心点做排序，这也意味着它不管每个图元在空间中的真正大小。这种简化对于HLBVH方法的执行效率有着关键影响，但这也意味着如果场景中有超大体积的图元，相比于SAH方法，通过这种方法构建的树没法很好处理这种变化(这句啥意思？先往后看)。
莫顿编码只是针对整型变量，第一件事是把所有图元包围盒的中心点组成的包围盒计算出来

```
<<Compute bounding box of all primitive centroids>>=
Bounds3f bounds;
for (const BVHPrimitiveInfo &pi : primitiveInfo)
    bounds = Union(bounds, pi.centroid);
```

有了整体的包围盒，下面计算莫顿码。计算过程本身是轻量级的，但由于图元众多，因此需要并行。

```
<<Compute Morton indices of primitives>>=
std::vector<MortonPrimitive> mortonPrims(primitiveInfo.siz
e());
//这里的512表示一个线程要处理512个图元
ParallelFor(
    [&](int i) {
        <<Initialize mortonPrims[i] for ith primitive>>
    }, primitiveInfo.size(), 512);
```

在这里，会构造MortonPrimitive结构体

```
<<BVHAccel Local Declarations>>+=
struct MortonPrimitive {
    ///在primitiveInfo数组中的索引
    int primitiveIndex;
```

```
        ///莫顿码
        uint32_t mortonCode;
    };
```

我们对于x,y,z三个维度的坐标分别用10bit来表示，因此莫顿码一共30bit。这么设计是为了让一个32bit的变量能放得下。相对于包围盒的浮点偏移在[0,1]区间内，因此我们将其缩放2^{10}，用来适配到10bit。（对于边际情况偏移量恰好等于1，这就有可能导致溢出，这时用函数LeftShift3()处理）

```
<<Initialize mortonPrims[i] for ith primitive>>=
//2^10
constexpr int mortonBits = 10;
constexpr int mortonScale = 1 << mortonBits;
//图元索引
mortonPrims[i].primitiveIndex = primitiveInfo[i].primitive
Number;
//Offset函数返回的是[0,1]区间的值，是个百分比
Vector3f centroidOffset = bounds.Offset(primitiveInfo[i].c
entroid);
mortonPrims[i].mortonCode = EncodeMorton3(centroidOffset *
 mortonScale);
```

下一步需要解决的问题，是计算三维坐标的莫顿码。

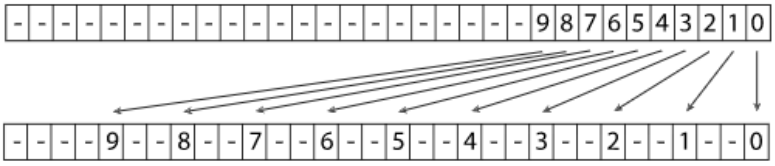


Figure 4.9: Bit Shifts to Compute 3D Morton Codes. The `LeftShift3()` function takes a 32-bit integer value and for the bottom 10 bits, shifts the ith bit to be in position $3i$—in other words, shifts it $2i$ places to the left. All other bits are set to zero.

单独移动每个bit操作太慢，因此这里用一个更好的办法：

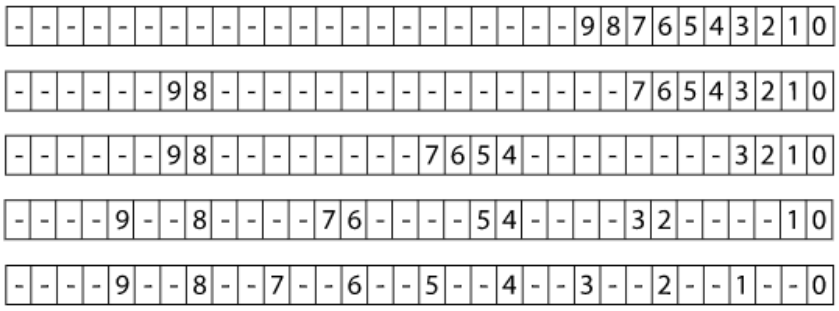


Figure 4.10: Power-of-Two Decomposition of Morton Bit Shifts. The bit shifts to compute the Morton code for each 3D coordinate are performed in a series of shifts of power-of-two size. First, bits 8 and 9 are shifted 16 places to the left. This places bit 8 in its final position. Next bits 4 through 7 are shifted 8 places. After shifts of 4 and 2 places (with appropriate masking so that each bit is shifted the right number of places in the end), all bits are in the proper position. This computation is implemented by the `LeftShift3()` function.

```
<<BVHAccel Utility Functions>>=
inline uint32_t LeftShift3(uint32_t x) {
    if (x == (1 << 10)) --x;
    x = (x | (x << 16)) & 0b00000011000000000000000011111111;
    x = (x | (x <<  8)) & 0b00000011000000001111000000001111;
    x = (x | (x <<  4)) & 0b00000011000011000011000011000011;
    x = (x | (x <<  2)) & 0b00001001001001001001001001001001;
    return x;
}
```

其实看看就行了，反正就是个平移bit的新方案。
EncodeMorton3把一个三维整型变量打包到一个32bit的整型数里。

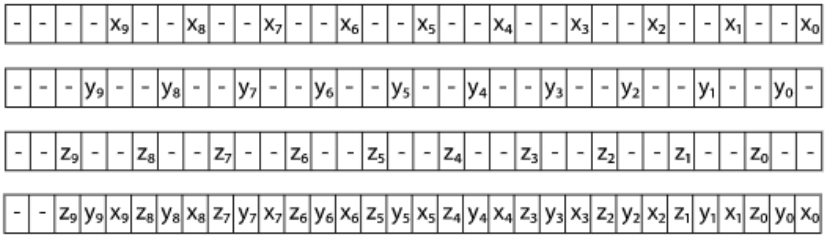


Figure 4.11: Final Interleaving of Coordinate Values. Given interleaved values for x, y, and z computed by `LeftShift3()`, the final Morton-encoded value is computed by shifting y and z one and two places, respectively, and then ORing together the results.

```
<<BVHAccel Utility Functions>>+=
```

```
inline uint32_t EncodeMorton3(const Vector3f &v) {
    return (LeftShift3(v.z) << 2) | (LeftShift3(v.y) << 1)
 |
           LeftShift3(v.x);
}
```

有了莫顿码，接下来就可以对mortonPrims进行排序，在这里，基数排序是最快的。

```
<<Radix sort primitive Morton indices>>=
RadixSort(&mortonPrims);
```

补充：基数排序
基数排序利用了桶的概念，LSD的算法流程如下：

1. 把0-9每个数字作为桶，桶按0-9的顺序，并且每个bucket是一个队列
2. 按照数字个位等于桶号的规则，所有数字分别入队分别入队
3. 再把0-9每个桶分别出队排在一个数组中，数组的结果是，只看个位是有序的
4. 把数组从头开始遍历，把每个数字按照十位再进桶，可以预见的是，每个桶中已经是个位按序了
5. 把0-9每个桶分别出队排在一个数组中，现在是个位十位都按序了
6. 循环往复，直到最大那个数的位数排完

```
//LSD Radix Sort
var counter = [];
function radixSort(arr, maxDigit) {
    var mod = 10;
    var dev = 1;
    for (var i = 0; i < maxDigit; i++, dev *= 10, mod *= 1
0) {
        for(var j = 0; j < arr.length; j++) {
            var bucket = parseInt((arr[j] % mod) / dev);
            if(counter[bucket]==null) {
                counter[bucket] = [];
            }
            counter[bucket].push(arr[j]);
        }
        var pos = 0;
        for(var j = 0; j < counter.length; j++) {
```

```
            var value = null;
            if(counter[j]!=null) {
                while ((value = counter[j].shift()) != nul
l) {
                    arr[pos++] = value;
                }
          }
        }
    }
    return arr;
}
```

基数排序如上所述，可用于将整型数据进行排序，每次是排一位，对于二进制数据，当然可以一次排序多个比特位，来降低总共的排序次数。下面的代码就是一次排序6个bit，对于30个bit的数据，5遍就能搞定了。

```
<<BVHAccel Utility Functions>>+=
static void RadixSort(std::vector<MortonPrimitive> *v) {
    std::vector<MortonPrimitive> tempVector(v->size());
    constexpr int bitsPerPass = 6;
    constexpr int nBits = 30;
    constexpr int nPasses = nBits / bitsPerPass;
    for (int pass = 0; pass < nPasses; ++pass) {
        <<Perform one pass of radix sort, sorting bitsPerP
ass bits>>
    }
    <<Copy final result from tempVector, if needed>>
}
```

当前pass处理bitsPerPass个bit，从lowBit开始

```
<<Perform one pass of radix sort, sorting bitsPerPass bits
>>=
int lowBit = pass * bitsPerPass;
<<Set in and out vector pointers for radix sort pass>>
<<Count number of zero bits in array for current radix sor
t bit>>
```

```
<<Compute starting index in output array for each bucket>>

<<Store sorted values in output array>>
```

双数组，一个输入一个输出，每遍循环来回切换

```
<<Set in and out vector pointers for radix sort pass>>=
std::vector<MortonPrimitive> &in  = (pass & 1) ? tempVecto
r : *v;
std::vector<MortonPrimitive> &out = (pass & 1) ? *v : temp
Vector;
```

下面其实就是基数排序的算法，没有必要多说了，上面已经讲过原理

```
<<Count number of zero bits in array for current radix sor
t bit>>=
constexpr int nBuckets = 1 << bitsPerPass;
int bucketCount[nBuckets] = { 0 };
constexpr int bitMask = (1 << bitsPerPass) - 1;
for (const MortonPrimitive &mp : in) {
    int bucket = (mp.mortonCode >> lowBit) & bitMask;
    ++bucketCount[bucket];
}
```

```
<<Compute starting index in output array for each bucket>>
=
int outIndex[nBuckets];
outIndex[0] = 0;
for (int i = 1; i < nBuckets; ++i)
    outIndex[i] = outIndex[i - 1] + bucketCount[i - 1];
```

```
<<Store sorted values in output array>>=
for (const MortonPrimitive &mp : in) {
    int bucket = (mp.mortonCode >> lowBit) & bitMask;
    out[outIndex[bucket]++] = mp;
}
```

```
<<Copy final result from tempVector, if needed>>=
if (nPasses & 1)
    std::swap(*v, tempVector);
```

现在图元数组已经经过排序了，下面的工作就是根据图元的包围盒中心点分簇(cluster)，以便创建LBVH。

```
<<Create LBVH treelets at bottom of BVH>>=
<<Find intervals of primitives for each treelet>>
<<Create LBVHs for treelets in parallel>>
```

每个簇都用LBVHTreelet来表示。

```
<<BVHAccel Local Declarations>>+=
struct LBVHTreelet {
    //图元起始索引，以及包含了多少个图元
   int startIndex, nPrimitives;
   BVHBuildNode *buildNodes;
};
```

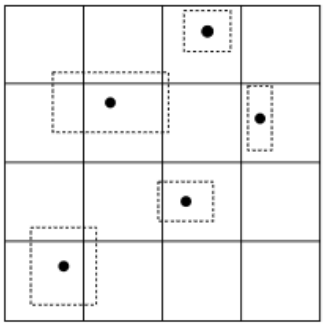


Figure 4.12: Primitive Clusters for LBVH Treelets. Primitive centroids are clustered in a uniform grid over their bounds. An LBVH is created for each cluster of primitives within a cell that are in contiguous sections of the sorted Morton index values.

上图中，一系列莫顿码的高位相同，因此被框到了一个2的n次幂对齐的子空间内。而也正因为是用莫顿码排序的，这些图元的莫顿码在数组中也是连续存放的。
接下来，我们通过在30个bit中的高12个bit是否发生变化来区分簇，即是用高12bit来分簇。对于三维空间来说，$2^{12} = 4096$个3D格子，对于每一个轴来说，就是$2^4 = 16$个一维区间。

```
<<Find intervals of primitives for each treelet>>=
std::vector<LBVHTreelet> treeletsToBuild;
```

```
for (int start = 0, end = 1; end <= (int)mortonPrims.size
(); ++end) {
    uint32_t mask = 0b00111111111111000000000000000000;
    if (end == (int)mortonPrims.size() ||
        ((mortonPrims[start].mortonCode & mask) !=
         (mortonPrims[end].mortonCode & mask))) {
        <<Add entry to treeletsToBuild for this treelet>>

        start = end;
    }
}
```

得到一簇图元之后，就可以调用BVHBuildNodes来创建treelet，因为BVH节点数量是叶子节点数量的2倍，因此可以预先分配内存。
一个实现细节是MemoryArena::Alloc()第二个参数，如果传入false，意思是虽然分配对象，但对象的构造函数不予执行。因为BVHBuildNode的所有成员都要手工初始化，因此调用构造函数并没有什么价值。

```
<<Add entry to treeletsToBuild for this treelet>>=
int nPrimitives = end - start;
int maxBVHNodes = 2 * nPrimitives - 1;
BVHBuildNode *nodes = arena.Alloc<BVHBuildNode>(maxBVHNode
s, false);
//在这里填充数组，每个元素都是需要构造treelet的所有图元，以及分配给
它的内存
treeletsToBuild.push_back({start, nPrimitives, nodes});
```

确定了treelet的所有图元，就可以开始构造LBVH了，构造完毕后，LBVHTreelet对应成员就会指向相关LBVH的根。
有两个地方需要进行线程同步，一个是需要返回一个总共LBVH节点的数量，另一个是一旦叶子节点创建出来，orderedPrims数组的邻接块数据需要用于记录叶子节点的图元索引。

```
<<Create LBVHs for treelets in parallel>>=
std::atomic<int> atomicTotal(0), orderedPrimsOffset(0);
//用来存储输出的的列表
orderedPrims.resize(primitives.size());
//在这里通过线程并行执行
ParallelFor(
    [&](int i) {
```

```
            <<Generate ith LBVH treelet>>
        }, treeletsToBuild.size());
    *totalNodes = atomicTotal;
```

真正构造treelet的是emitLBVH()函数，它获得一些区域空间的图元的包围盒中心点，通过分割平面进行分割，沿着三根轴的某一根轴将当前区域空间分割成两半。
emitLBVH()更新的是非原子操作的本地变量，直到一个treelet构造完毕才去更新一次atomicTotal。

```
<<Generate ith LBVH treelet>>=
int nodesCreated = 0;
//在上一步中，把图元分配到了不同的treelet中，已经占用了12个bit，接下来的是使用剩下的bit
const int firstBitIndex = 29 - 12;
//处理当前的数据来构造treelt
LBVHTreelet &tr = treeletsToBuild[i];
tr.buildNodes =
    emitLBVH(tr.buildNodes, primitiveInfo, &mortonPrims[tr.startIndex],
             tr.nPrimitives, &nodesCreated, orderedPrims,
             &orderedPrimsOffset, firstBitIndex);
atomicTotal += nodesCreated;
```

之前已经用了12个bit，接下来用剩下的bit，首先可以肯定的是，每个treelet的图元都在一个cluster中。

```
<<BVHAccel Method Definitions>>+=
BVHBuildNode *BVHAccel::emitLBVH(BVHBuildNode *&buildNodes,
        const std::vector<BVHPrimitiveInfo> &primitiveInfo,
        MortonPrimitive *mortonPrims, int nPrimitives, int *totalNodes,
        std::vector<std::shared_ptr<Primitive>> &orderedPrims,
        std::atomic<int> *orderedPrimsOffset, int bitIndex) const {
    //bitIndex就是开始的bit位，如果没有了，就说明分割结束了，创建叶子节点
    if (bitIndex == -1 || nPrimitives < maxPrimsInNode) {
```

```
            <<Create and return leaf node of LBVH treelet>>
        } else {
            //如果还有可以参与分割的bit位，则继续分割
            //这个mask
            int mask = 1 << bitIndex;
            <<Advance to next subtree level if there's no LBVH
 split for this bit>>
            <<Find LBVH split point for this dimension>>
            <<Create and return interior LBVH node>>
        }
    }
```

orderedPrimOffset对应的是orderedPrims数组，这个数组用来输出节点，而这个offsest变量则指向的是下一个可以用来填充的位置。

```
<<Create and return leaf node of LBVH treelet>>=
(*totalNodes)++;
BVHBuildNode *node = buildNodes++;
Bounds3f bounds;
//fetch_add是原子操作，自身增加，并返回增加之前的值
int firstPrimOffset = orderedPrimsOffset->fetch_add(nPrimi
tives);
//遍历每个图元，与morton prims同样的顺序添加到orderedPrims数组
中，注意这一步就是实际在进行排序操作，之前的构造的mortonPrims数组，
仅仅是mortonPrims本身按照莫顿码有序了，并没有动原始的primitive数
组
for (int i = 0; i < nPrimitives; ++i) {
    int primitiveIndex = mortonPrims[i].primitiveIndex;
    orderedPrims[firstPrimOffset + i] = primitives[primiti
veIndex];
    bounds = Union(bounds, primitiveInfo[primitiveIndex].b
ounds);
}
//构造叶子节点
node->InitLeaf(firstPrimOffset, nPrimitives, bounds);
return node;
```

上面的代码片段是构造叶子节点的，下面开始考虑进行空间再分割的情况。
有一种情况，就是所有图元都在切割平面的同一边，因为所有图元都是用莫顿码来排序

的，因此可以检查第一个和最后一个图元的bit值，是否与切割面的对应bit位相同。如果相同，判断下一个bit。

```
<<Advance to next subtree level if there's no LBVH split f
or this bit>>=
//如果最高位bit相同的话，则前进一位，没必要把相同的最高位考虑进去了
if ((mortonPrims[0].mortonCode & mask) ==
    (mortonPrims[nPrimitives - 1].mortonCode & mask))
    return emitLBVH(buildNodes, primitiveInfo, mortonPrim
s, nPrimitives,
                    totalNodes, orderedPrims, orderedPrims
Offset,
                    bitIndex - 1);
```

如果图元在分割平面的两边的话，通过二分查找搜索分割点，即bitIndex从0到1的变化位置

```
<<Find LBVH split point for this dimension>>=
int searchStart = 0, searchEnd = nPrimitives - 1;
while (searchStart + 1 != searchEnd) {
    int mid = (searchStart + searchEnd) / 2;
    if ((mortonPrims[searchStart].mortonCode & mask) ==
        (mortonPrims[mid].mortonCode & mask))
        searchStart = mid;
    else
        searchEnd = mid;
}
int splitOffset = searchEnd;
```

这里最终拿到的结果是splitOffset，即mortonPrims数组中的分割点的位置，拿到了分割点就可以一分为二，并且创建内部节点

```
<<Create and return interior LBVH node>>=
(*totalNodes)++;
BVHBuildNode *node = buildNodes++;
BVHBuildNode *lbvh[2] = {
    emitLBVH(buildNodes, primitiveInfo, mortonPrims, split
Offset,
```

```
                totalNodes, orderedPrims, orderedPrimsOffset,
 bitIndex - 1),
    emitLBVH(buildNodes, primitiveInfo, &mortonPrims[split
Offset],
                nPrimitives - splitOffset, totalNodes, ordere
dPrims,
                orderedPrimsOffset, bitIndex - 1)
};
int axis = bitIndex % 3;
node->InitInterior(axis, lbvh[0], lbvh[1]);
return node;
```

一旦所有的LBVH treelet创建完毕，调用builderUpperSAH()函数，就可以为所有的treelet创建BVH。

```
<<Create and return SAH BVH from LBVH treelets>>=
std::vector<BVHBuildNode *> finishedTreelets;
//经过上面的所有运算，treeletsToBuild的每个元素的buildNodes都已
经被填充完毕了
for (LBVHTreelet &treelet : treeletsToBuild)
    finishedTreelets.push_back(treelet.buildNodes);
return buildUpperSAH(arena, finishedTreelets, 0,
                     finishedTreelets.size(), totalNodes);
```

```
<<BVHAccel Private Methods>>=
BVHBuildNode *buildUpperSAH(MemoryArena &arena,
    std::vector<BVHBuildNode *> &treeletRoots, int start,
int end,
    int *totalNodes) const;
```

4.3.4 Compact BVH For Traversal

BVH树构造完毕之后，下面一步就是把它打包到一个数组当中。即二叉树的数组表示。

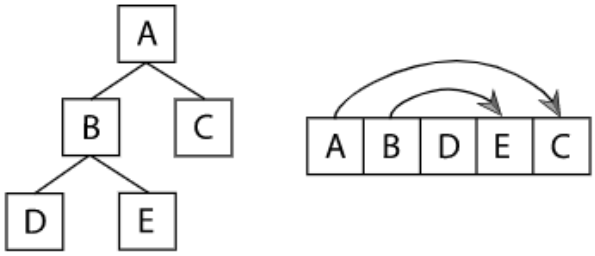


Figure 4.13: Linear Layout of a BVH in Memory. The nodes of the BVH (left) are stored in memory in depth-first order (right). Therefore, for any interior node of the tree (A and B in this example), the first child is found immediately after the parent node in memory. The second child is found via an offset pointer, represented here with lines with arrows. Leaf nodes of the tree (D, E, and C) have no children.

采用的数据结构名为LinearBVHNode

```
<<BVHAccel Local Declarations>>+=
struct LinearBVHNode {
    Bounds3f bounds;
    union {
        int primitivesOffset;    // leaf
        int secondChildOffset;   // interior
    };
    uint16_t nPrimitives;  // 0 -> interior node
    uint8_t axis;          // interior node: xyz
    uint8_t pad[1];        // ensure 32 byte total size
};
```

其中pad是为了保证32字节对齐。
flattenBVHTree()函数就是用来将树结构转换为LinearBVHNode表示，采用深度优先的方法。

```
<<Compute representation of depth-first traversal of BVH t
ree>>=
nodes = AllocAligned<LinearBVHNode>(totalNodes);
int offset = 0;
flattenBVHTree(root, &offset);
```

其中nodes是BVHAccel的成员变量

```
<<BVHAccel Private Data>>+=
LinearBVHNode *nodes = nullptr;
```

下面就是二叉树平展到数组中，很简单，不说了

```
<<BVHAccel Method Definitions>>+=
int BVHAccel::flattenBVHTree(BVHBuildNode *node, int *offs
et) {
    LinearBVHNode *linearNode = &nodes[*offset];
    linearNode->bounds = node->bounds;
    int myOffset = (*offset)++;
    if (node->nPrimitives > 0) {
        linearNode->primitivesOffset = node->firstPrimOffs
et;
        linearNode->nPrimitives = node->nPrimitives;
    } else {
        <<Create interior flattened BVH node>>
    }
    return myOffset;
}
```

```
<<Create interior flattened BVH node>>=
linearNode->axis = node->splitAxis;
linearNode->nPrimitives = 0;
flattenBVHTree(node->children[0], offset);
linearNode->secondChildOffset =
    flattenBVHTree(node->children[1], offset);
```

4.3.5 Traversal

首先回忆一下公式:

$$t_1 = \frac{x_1 - o_x}{d_x}$$

因此，这里会有invDir这么个变量。
接下来是射线检测函数

```
<<BVHAccel Method Definitions>>+=
bool BVHAccel::Intersect(const Ray &ray,
        SurfaceInteraction *isect) const {
    //注意传进来的ray是世界空间的
    bool hit = false;
```

```
    Vector3f invDir(1 / ray.d.x, 1 / ray.d.y, 1 /
ray.d.z);
    int dirIsNeg[3] = { invDir.x < 0, invDir.y < 0, invDi
r.z < 0 };
    <<Follow ray through BVH nodes to find primitive inter
sections>>
    return hit;
}
```

```
<<Follow ray through BVH nodes to find primitive intersect
ions>>=
int toVisitOffset = 0, currentNodeIndex = 0;
//不用递归了，外边定义栈
int nodesToVisit[64];
while (true) {
    const LinearBVHNode *node = &nodes[currentNodeIndex];
    <<Check ray against BVH node>>
}
```

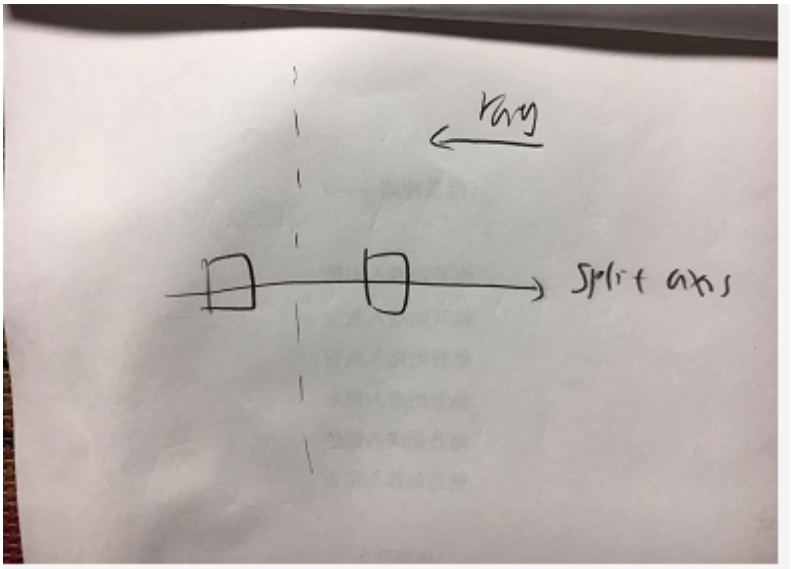
```
<<Check ray against BVH node>>=
if (node->bounds.IntersectP(ray, invDir, dirIsNeg)) {
    //包围盒和射线相交
    if (node->nPrimitives > 0) {
        //叶子节点
        <<Intersect ray with primitives in leaf BVH node>>

    } else {
        //内部节点
        <<Put far BVH node on nodesToVisit stack, advance
 to near node>>
    }
} else {
    //包围盒和射线没有相交
    if (toVisitOffset == 0) break;
    currentNodeIndex = nodesToVisit[--toVisitOffset];
}
```

```
<<Intersect ray with primitives in leaf BVH node>>=
```

```
//检测每个图元是否相交
for (int i = 0; i < node->nPrimitives; ++i)
    if (primitives[node->primitivesOffset + i]->Intersect
(ray, isect))
        hit = true;
if (toVisitOffset == 0) break;
currentNodeIndex = nodesToVisit[--toVisitOffset];
```

如果一个内部节点与射线发生了碰撞，那么就需要遍历两个子。如上面所述，先访问第一个子，把ray的max缩短，很可能第二个子就碰不到了，这种思路就很好。
一种有效的从前往后的遍历而不发生于子节点的相交检测，以及比较距离的方法，就是使用射线方向向量与分割图元的坐标轴的点乘的符号。如果是负号，我们应该先访问右子树，再访问左子树，因为进入右子树的图元在分割点的前边。



一目了然。

```
<<Put far BVH node on nodesToVisit stack, advance to near
 node>>=
if (dirIsNeg[node->axis]) {
   nodesToVisit[toVisitOffset++] = currentNodeIndex + 1;
   currentNodeIndex = node->secondChildOffset;
} else {
   nodesToVisit[toVisitOffset++] = node->secondChildOffse
t;
   currentNodeIndex = currentNodeIndex + 1;
}
```

BVHAccel::IntersectP是其简化版本，只要碰到一个相交点，检测就会停止。

4.4 KDTree Accelerator

二叉空间分割(binary space partitioning, BSP)树是用平面将空间分割，当一个包围盒内图元数量超限时，就需要用平面将其一分为二。图元与其覆盖的半空间所关联，当然如果图元足够大，覆盖两个半空间，则与两个半空间同时关联。（与BVH不同，BVH中图元只属于一个子树，当然，子树与子树是可以互相重叠的，图元的包围盒中心点在哪个子树下，那就属于那个子树。）
构建BSP的停止条件是子空间内的图元数量足够少，或者BSP的深度足够深。因为分割点是空间中任意的，并且空间的任意部分对应树的深度也是不同的，因此BSP可以很简单地处理物体分布不均匀的空间。
BSP的两种变体，一种是kd-tree(kd树)，另一种是octree(八叉树)。kd-tree要求切分空间的平面与某一根坐标轴垂直。octree用三根坐标轴将包围盒分割成八个区域。这个章节我们需要实现的就是KdTreeAccel类。

```
<<KdTreeAccel Declarations>>=
class KdTreeAccel : public Aggregate {
public:
    <<KdTreeAccel Public Methods>>
private:
    <<KdTreeAccel Private Methods>>
    <<KdTreeAccel Private Data>>
};
```

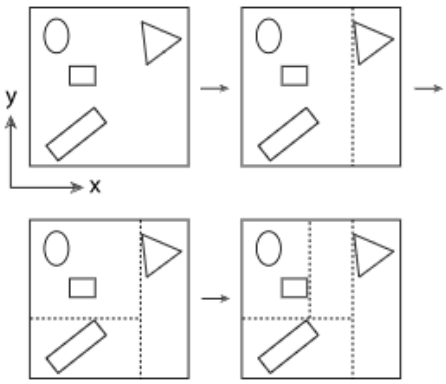


Figure 4.14: The kd-tree is built by recursively splitting the bounding box of the scene geometry along one of the coordinate axes. Here, the first split is along the x axis; it is placed so that the triangle is precisely alone in the right region and the rest of the primitives end up on the left. The 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.

```
<<KdTreeAccel Method Definitions>>=
```

```
KdTreeAccel::KdTreeAccel(
        const std::vector<std::shared_ptr<Primitive>> &p,
        int isectCost, int traversalCost, Float emptyBonu
s,
        int maxPrims, int maxDepth)
    : isectCost(isectCost), traversalCost(traversalCost),
      maxPrims(maxPrims), emptyBonus(emptyBonus), primitiv
es(p) {
    <<Build kd-tree for accelerator>>
}
<<KdTreeAccel Private Data>>=
const int isectCost, traversalCost, maxPrims;
const Float emptyBonus;
std::vector<std::shared_ptr<Primitive>> primitives;
```

4.4.1 Tree Representation

kd-tree是一种二叉树，叶子节点保存了与其重叠的图元信息，内部节点包含以下信息：

- 分割轴：x,y,z中的哪个轴来分割当前节点
- 分割位置：分割平面的位置
- 子节点

每个叶子节点记录了哪些图元与其重叠。
值得注意的是，大多数叶子节点和内部节点都只有8个字节大小，因为这么做的话，8个节点就能组成64字节，符合系统缓存要求。我们的初始实现一个节点有16个字节，接下来降低到8字节能带来20%的性能提升。

```
<<KdTreeAccel Local Declarations>>=
struct KdAccelNode {
    <<KdAccelNode Methods>>
    union {
        Float split;                 // Interior
        int onePrimitive;            // Leaf
        int primitiveIndicesOffset;  // Leaf
    };
    union {
        int flags;          // Both
        int nPrims;         // Leaf
        int aboveChild;     // Interior
    };
```

```
};
```

在这里，flags的低位的2个bit，用来区分是内部节点还是叶子节点。内部节点分为3种，分别是由x,y,z三个轴来分割，这里用0,1,2来表示，或者如果是叶子节点，则用3来表示，用满了前两个bit位。对于叶子节点而言，前两个低位bit已经用了，那么剩下的30个高位bit，亦即KdAccelNode::nPrims的30个高位bit，用来表示这个叶子节点下有多少个图元。如果只有一个图元在叶子节点内，那么一个整形索引指向的KdTreeAccel::primtives数组元素就是该图元。如果多于1个图元，则它们的索引就会被单独存储在KdTreeAccel::primitiveIndices中，并且在这个索引数组的偏移量offset就保存在叶子节点中的 KdAccelNode::primitiveIndicesOffset变量中。

```
<<KdTreeAccel Private Data>>+=
std::vector<int> primitiveIndices;
```

下面是初始化叶子节点

```
<<KdTreeAccel Method Definitions>>+=
void KdAccelNode::InitLeaf(int *primNums, int np,
        std::vector<int> *primitiveIndices) {
    //初始化低位为3
    flags = 3;
    //图元数量放高位
    nPrims |= (np << 2);
    <<Store primitive ids for leaf node>>
}
```

对于一个没有或者只有1个图元的叶子节点，不需要额外分配内存空间，但如果有多个图元的话，就需要分配在primitiveIndices数组中。

```
<<Store primitive ids for leaf node>>=
if (np == 0)
    onePrimitive = 0;
else if (np == 1)
    //只有一个图元，onePrimitive存储图元的索引
    onePrimitive = primNums[0];
else {
    //有多个图元，则要存储在primitiveIndices中的索引偏移
    primitiveIndicesOffset = primitiveIndices->size();
    for (int i = 0; i < np; ++i)
```

```
            primitiveIndices->push_back(primNums[i]);
    }
```

上面的代码要补充一点的是，np已经被存储到了nPrims的高位，因此可以随便用onePrimitive这个变量的union，不会产生混乱。

对于内部节点，让它的尺寸保持在8个字节大小也是非常直接的。浮点型变量本身存储空间是32个bit，用来保存沿着分割轴方向的位置，其中分割轴是KdAccelNode::flags的低位2个bit。剩下的事情就是存储足够的信息以便于在遍历的时候能找到它的两个子节点。

我们这里不用保存两个子节点的指针或者偏移，而只需要保存一个就够了：所有节点都是保存在一个数组中，而内部节点的第一个子节点肯定是保存在当前内部节点的下一个位置上(见二叉树的数组保存格式)，而第二个子节点则不一定在哪，因此KdAccelNode::aboveChild就存储第二个子节点的位置。

因此以内部节点的方式初始化就显而易见了：

```
<<KdAccelNode Methods>>=
void InitInterior(int axis, int ac, Float s) {
    split = s;
    flags = axis;
    aboveChild |= (ac << 2);
}
```

剩下的就是获取需要的值的方法

```
<<KdAccelNode Methods>>+=
Float SplitPos() const { return split; }
int nPrimitives() const { return nPrims >> 2; }
int SplitAxis() const { return flags & 3; }
bool IsLeaf() const { return (flags & 3) == 3; }
int AboveChild() const { return aboveChild >> 2; }
```

4.4.2 Tree Construction

kd-tree的构建方法是自上而下的，一开始有一个轴对齐空间，以及一系列图元在这个空间内，然后要么将空间一分为二形成两个子空间，要么直接生成叶子节点，然后递归操作。

所有KdAccelNodes节点都放在一个数组当中，因此KdTreeAccel::nextFreeNode记录了下一个可用的节点，以及KdTreeAccel::nAllocedNodes记录了一共分配了多少个节点。都为0则是一个节点都没有开始分配。

接下来需要决定树的最大深度，这样就能限制最大创建节点的数量，以保证内存不会被撑爆。pbrt使用的值为$8 + 1.3 \log(N)$。

```
<<Build kd-tree for accelerator>>=
nextFreeNode = nAllocedNodes = 0;
if (maxDepth <= 0)
    maxDepth = std::round(8 + 1.3f * Log2Int(primitives.si
ze()));
<<Compute bounds for kd-tree construction>>
<<Allocate working memory for kd-tree construction>>
<<Initialize primNums for kd-tree construction>>
<<Start recursive construction of kd-tree>>

<<KdTreeAccel Private Data>>+=
KdAccelNode *nodes;
int nAllocedNodes, nextFreeNode;
```

因为后面构造树的时候要频繁用到包围盒，因此先把所有图元的包围盒放到vector中

```
<<Compute bounds for kd-tree construction>>=
std::vector<Bounds3f> primBounds;
for (const std::shared_ptr<Primitive> &prim : primitives)
 {
    Bounds3f b = prim->WorldBound();
    bounds = Union(bounds, b);
    primBounds.push_back(b);
}

<<KdTreeAccel Private Data>>+=
Bounds3f bounds;
```

构造树节点还需要一个图元的索引列表，对于根节点来说，索引列表包含全部图元。

```
<<Initialize primNums for kd-tree construction>>=
std::unique_ptr<int[]> primNums(new int[primitives.size
()]);
for (size_t i = 0; i < primitives.size(); ++i)
    primNums[i] = i;
```

同时对于每个节点都会调用buildTree()，在这里需要决定当前节点是叶子节点还是内部节点，最后3个参数在后面会讲。

```
<<Start recursive construction of kd-tree>>=
buildTree(0, bounds, primBounds, primNums.get(), primitive
s.size(),
          maxDepth, edges, prims0.get(), prims1.get());
```

buildTree()的参数中

- nodeNum：KdAccelNodes数组偏移量
- nodeBounds：所有图元的最大包围盒
- allPrimBounds：每个图元的各自包围盒
- primNums：索引列表
- 其他的会在后面提到

```
<<KdTreeAccel Method Definitions>>+=
void KdTreeAccel::buildTree(int nodeNum, const Bounds3f &n
odeBounds,
        const std::vector<Bounds3f> &allPrimBounds, int *p
rimNums,
        int nPrimitives, int depth,
        const std::unique_ptr<BoundEdge[]> edges[3],
        int *prims0, int *prims1, int badRefines) {
    <<Get next free node from nodes array>>
    <<Initialize leaf node if termination criteria met>>
    <<Initialize interior node and continue recursion>>
}
```

分配节点用的是比较常规的内存池方案

```
<<Get next free node from nodes array>>=
if (nextFreeNode == nAllocedNodes) {
    int nNewAllocNodes = std::max(2 * nAllocedNodes, 512);
    KdAccelNode *n = AllocAligned<KdAccelNode>(nNewAllocNo
des);
    if (nAllocedNodes > 0) {
        memcpy(n, nodes, nAllocedNodes * sizeof(KdAccelNod
e));
        FreeAligned(nodes);
```

```
        }
        nodes = n;
        nAllocedNodes = nNewAllocNodes;
    }
    ++nextFreeNode;
```

作为叶子节点初始化的条件很明显，如果图元数量足够少，或者已经达到最大深度，不允许增加深度了，就直接转成叶子节点。

```
    <<Initialize leaf node if termination criteria met>>=
    if (nPrimitives <= maxPrims || depth == 0) {
        nodes[nodeNum].InitLeaf(primNums, nPrimitives, &primit
    iveIndices);
        return;
    }
```

如果是创建内部节点，那就麻烦的多

```
    <<Initialize interior node and continue recursion>>=
    <<Choose split axis position for interior node>>
    <<Create leaf if no good splits were found>>
    <<Classify primitives with respect to split>>
    <<Recursively initialize children nodes>>
```

首先是用SAH选择分割方式。即选择一个节点的分割平面，使其执行开销最低。相交测试开销t_{isect}与遍历开销t_{trav}被设定为80和1，
相比于BVH，kd-tree的SAH公式发生了一些变化，因为选择切割平面的时候，如果子空间内没有图元，则可以做一些调整，因为射线进入这些区域，可以直接跳转到下一个kd-tree节点，而不必做任何求交测试。因此最终公式为

$$t_{isect}N$$

以及

$$t_{trav} + (1 - b_e)(p_B N_B t_{isect} + p_A N_A t_{isect})$$

其中，
b_e是奖励值，一直为0，除非两个区域的其中一个区域是空的，这种情况下，b_e的值在0和1之间。
之前已经讲过怎么计算开销模型的概率，现在唯一的问题，就是寻找切分点，以及如何有效计算开销。很显然的，开销最小的方式，一定是分割面就在某一个图元的包围盒的

某一个面上，这样一来就不用考虑中间的位置，因此我们只考虑所有包围盒的面就行了。
下面介绍检查所有可能分割面的开销的算法，

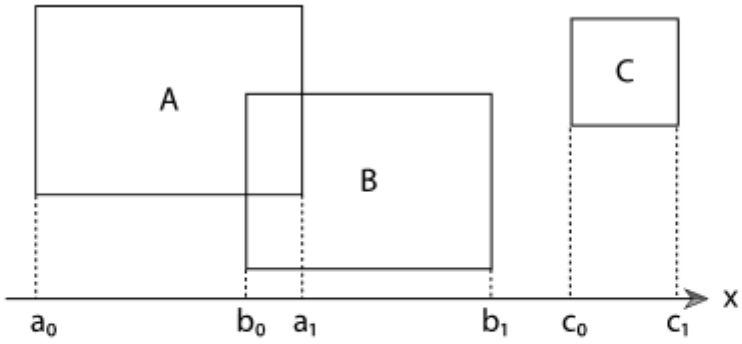


Figure 4.15: Given an axis along which we'd like to consider possible splits, the primitives' bounding boxes are projected onto the axis, which leads to an efficient algorithm to track how many primitives would be on each side of a particular splitting plane. Here, for example, a split at a_1 would leave 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.

如图，将ABC3个包围盒投影到某根坐标轴上，每个包围盒都能得到两个面在x轴上的投影，例如A对应的是a_0和a_1，我们用BoundEdge来存储这个信息。

```
<<KdTreeAccel Local Declarations>>+=
enum class EdgeType { Start, End };

<<KdTreeAccel Local Declarations>>+=
struct BoundEdge {
    <<BoundEdge Public Methods>>
    //边的位置
    Float t;
    ///对应第几号图元
    int primNum;
    ///是包围盒在轴上投影的开始边还是结束边
    EdgeType type;
};

<<BoundEdge Public Methods>>=
BoundEdge(Float t, int primNum, bool starting)
    : t(t), primNum(primNum) {
    type = starting ? EdgeType::Start : EdgeType::End;
}
```

每根轴上的edge数量是包围盒数量的两倍

```
<<Allocate working memory for kd-tree construction>>=
std::unique_ptr<BoundEdge[]> edges[3];
for (int i = 0; i < 3; ++i)
    edges[i].reset(new BoundEdge[2 * primitives.size()]);
```

确定了构建叶子节点需要的开销后，KdTreeAccel::buildTree()选择一个分割轴，然后为每个可能的分割面计算开销。

```
<<Choose split axis position for interior node>>=
//记录挑选的坐标轴，以及目前计算出来的最好的分割位置，用包围盒的边缘索引表示
int bestAxis = -1, bestOffset = -1;
//目前最优的开销的值
Float bestCost = Infinity;
//old cost是用对每个图元进行相交操作的开销来初始化的
Float oldCost = isectCost * Float(nPrimitives);
//最外层包围盒总面积
Float totalSA = nodeBounds.SurfaceArea();
Float invTotalSA = 1 / totalSA;
Vector3f d = nodeBounds.pMax - nodeBounds.pMin;
<<Choose which axis to split along>>
int retries = 0;
retrySplit:
<<Initialize edges for axis>>
<<Compute cost of all splits for axis to find best>>
```

接下来，寻找分割轴还是老办法，包围盒最大的那一维。

```
<<Choose which axis to split along>>=
int axis = nodeBounds.MaximumExtent();
```

首先，把所有**需要处理的**包围盒拿出来，把边拿到，存到数组edges中

```
<<Initialize edges for axis>>=
for (int i = 0; i < nPrimitives; ++i) {
    int pn = primNums[i];
    const Bounds3f &bounds = allPrimBounds[pn];
    edges[axis][2 * i] =     BoundEdge(bounds.pMin[axis],
 pn, true);
```

```
        edges[axis][2 * i + 1] = BoundEdge(bounds.pMax[axis],
     pn, false);
    }
    <<Sort edges for axis>>
```

下一步就是排序，用c++标准库的sort()，排序规则是BoundEdge::t，即边缘的位置。另外一点要注意的是，即使t相同，包围盒也是区分左右边的，要把左右边考虑进去。同时，要注意到，当且仅当$a < b$以及$a > b$都为false时，$a == b$。

```
    <<Sort edges for axis>>=
    std::sort(&edges[axis][0], &edges[axis][2*nPrimitives],
        [](const BoundEdge &e0, const BoundEdge &e1) -> bool {
            if (e0.t == e1.t)
                return (int)e0.type < (int)e1.type;
            else return e0.t < e1.t;
        });
```

到现在为止，拿到了所有排序后的边。之后依然是计算概率，方法依然是计算面积，分割后两边的图元数量用nBelow和nAbove来保存，在每次循环的手，nBelow和nAbove都是实时更新的。

```
    <<Compute cost of all splits for axis to find best>>=
    //一开始所有的图元都在分割面右侧，然后开始寻找分割面。注意，这里的nB
    elow和nAbove说的是图元数量，不是边的数量
    int nBelow = 0, nAbove = nPrimitives;
    for (int i = 0; i < 2 * nPrimitives; ++i) {
        //遍历每条边
        //如果跨过一个图元的右边，那么nAbove-1
        if (edges[axis][i].type == EdgeType::End) --nAbove;
        Float edgeT = edges[axis][i].t;
        if (edgeT > nodeBounds.pMin[axis] &&
            edgeT < nodeBounds.pMax[axis]) {
            //如果边缘在合法范围内，则计算切分的开销
            <<Compute cost for split at ith edge>>
        }
        //如果跨过一个图元的左边，则nBelow+1
        if (edges[axis][i].type == EdgeType::Start) ++nBelow;
    }
```

```
<<Compute child surface areas for split at edgeT>>=
int otherAxis0 = (axis + 1) % 3, otherAxis1 = (axis + 2) %
 3;
//belowSA和aboveSA是分别是分割点下面和上面包围盒的面积，d是vector
3类型的包围盒尺寸
//d[otherAxis0] * d[otherAxis1]是第一个面的面积，d[otherAxis
0] + d[otherAxis1])是两根轴包围盒长度相加，再乘以edgeT - nodeBo
unds.pMin[axis]，就得到剩余两个面的面积。最后乘以2，就是6个面的面
积。
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]));
```

然后，就是算cost，回顾上面的公式

$$t_{trav} + (1 - b_e)(p_B N_B t_{isect} + p_A N_A t_{isect})$$

```
Compute cost for split at ith edge>>=
<<Compute child surface areas for split at edgeT>>
Float pBelow = belowSA * invTotalSA;
Float pAbove = aboveSA * invTotalSA;
Float eb = (nAbove == 0 || nBelow == 0) ? emptyBonus : 0;
Float cost = traversalCost +
             isectCost * (1 - eb) * (pBelow * nBelow + pAb
ove * nAbove);
<<Update best split if this is lowest cost so far>>
```

最终，找到best cost

```
<<Update best split if this is lowest cost so far>>=
if (cost < bestCost)  {
    bestCost = cost;
    bestAxis = axis;
    bestOffset = i;
}
```

如果出现了没法进行切分的情况，如下图，首先会尝试换个轴去切分，3个轴都试过了还不行，只能放弃，创建叶子节点。

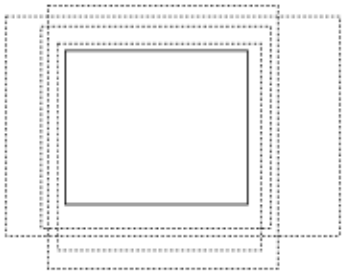


Figure 4.16: If multiple bounding boxes (dotted lines) overlap a kd-tree node (solid lines) as shown here, there is no possible split position that can result in fewer than all of the primitives being on both sides of it.

另外一种可能性就是，进行空间分割后的开销反而比分割前的开销还要大，这个时候就干脆创建个叶子节点就完事了。

```
<<Create leaf if no good splits were found>>=
//如果没有找到当前
if (bestAxis == -1 && retries < 2) {
    ++retries;
    axis = (axis + 1) % 3;
    goto retrySplit;
}
if (bestCost > oldCost) ++badRefines;
//最好情况下的开销要比遍历每个图元的开销的4倍还要大，并且图元数量足够小
//或者根本没有找到最好的分割轴的时候，就直接创建叶子节点
if ((bestCost > 4 * oldCost && nPrimitives < 16) ||
    bestAxis == -1 || badRefines == 3) {
    nodes[nodeNum].InitLeaf(primNums, nPrimitives, &primitiveIndices);
    return;
}
```

有了分割点之后，就可以以此为依据，用包围盒的边缘信息知道哪些图元在分割点上方，哪些在下方，或者跨过了分割点。这里用的方法和之前的nAbove和nBelow差不多。需要注意的是，这里故意跳过了bestOffset那条边，这意味着用于分割的图元实际上只能位于一侧。

```
<<Classify primitives with respect to split>>=
int n0 = 0, n1 = 0;
```

```
//以bestOffset为分界线，如果在best offset左边，只要这条边是包围盒左侧的边，则把这个图元算作左边记录下来，如果在best offset右边，只要这条边是包围盒右侧的边，则把这个图元作为算作右边记录下来
for (int i = 0; i < bestOffset; ++i)
    if (edges[bestAxis][i].type == EdgeType::Start)
        prims0[n0++] = edges[bestAxis][i].primNum;
for (int i = bestOffset + 1; i < 2 * nPrimitives; ++i)
    if (edges[bestAxis][i].type == EdgeType::End)
        prims1[n1++] = edges[bestAxis][i].primNum;
```

这就达到了预想的目标：如果一个图元包围盒的左边在分割点左侧，右边在分割点右侧，则这个图元会被同时记录在两个子树内。
回顾一下，kd_tree的数组存储结构中，在当前节点下面的子（即左子树的第一个节点），其索引是当前kdtree节点+1，在构建完左子树之后，接下来，nextFreeNode的偏移量，就是给分配右子树的。

```
<<Recursively initialize children nodes>>=
//确定分割位置
Float tSplit = edges[bestAxis][bestOffset].t;
//首先初始化左右两个子树的包围盒
Bounds3f bounds0 = nodeBounds, bounds1 = nodeBounds;
//左子树的包围盒的max和右子树包围盒的min都是这个分割点，这样左右两个子树的包围盒就都正确了。
bounds0.pMax[bestAxis] = bounds1.pMin[bestAxis] = tSplit;
//构造左子树，第一个参数是用于构造左子树的节点索引
//需要注意的是，倒数第二个参数是nprims1 + nPrimitives，因此才有下边的要给prims1留足足够多的空间
//在传参的时候，prims0不但要作为primNums传入，还要作为输出左子树的缓冲区使用。primNums在buildTree函数内创建完edge列表之后就没用了，接下来的流程就会把prims0和prims1作为输出修改掉。在左子树构造完毕以后，prims1实际内容发生了改变，但在构造右子树的时候，还得将prims1作为全部图元的输入调用buildTree，因此prims1内容需要在调用左子树的buildTree以后保留，因此就出现了prims1 + nPrimitives这种做法，把作为输出的缓冲区，和作为primNums输入的缓冲区指针给岔开。
buildTree(nodeNum + 1, bounds0, allPrimBounds, prims0, n0,
          depth - 1, edges, prims0, prims1 + nPrimitives,
 badRefines);
//构造完左子树，下一个可用节点，就是右子树的开始节点
int aboveChild = nextFreeNode;
//构造当前节点为中间节点
nodes[nodeNum].InitInterior(bestAxis, aboveChild, tSplit);
```

```
//构造右子树
buildTree(aboveChild, bounds1, allPrimBounds, prims1, n1,
          depth - 1, edges, prims0, prims1 + nPrimitives,
 badRefines);
```

prims1为了预防最糟糕的分割可能性，需要分配足够多的空间

```
<<Allocate working memory for kd-tree construction>>+=
std::unique_ptr<int[]> prims0(new int[primitives.size()]);
std::unique_ptr<int[]> prims1(new int[(maxDepth+1) * primi
tives.size()]);
```

4.4.3 Traversal

射线如果没跟包围盒相交，则return false，如果相交了，则分别判断两个子树。就是经典的二叉树求交的流程。

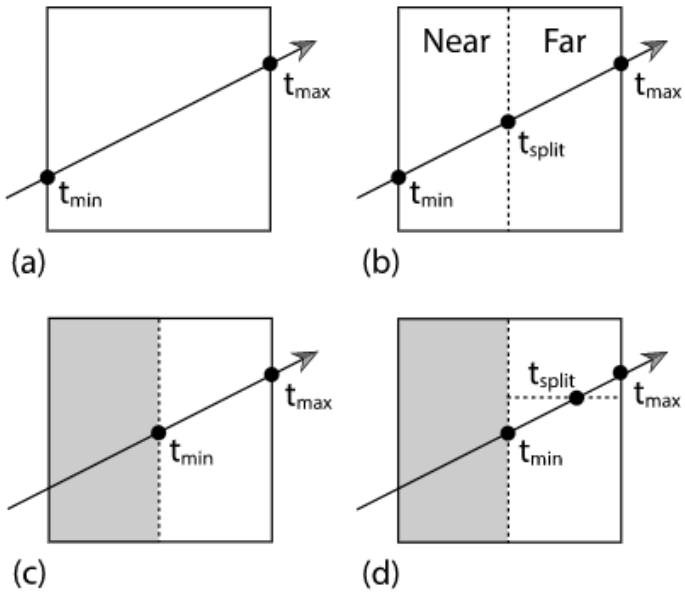


Figure 4.17: Traversal of a Ray through the kd-Tree. (a) The ray is intersected with the bounds of the tree, giving an initial parametric $[t_{\min}, t_{\max}]$ range to consider. (b) Because this range is nonempty, it is necessary to consider the two children of the root node here. The ray first enters the child on the right, labeled "near," where it has a parametric range $[t_{\min}, t_{\text{split}}]$. 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 $[t_{\min}, t_{\text{split}}]$, 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.

```
<<KdTreeAccel Method Definitions>>+=
bool KdTreeAccel::Intersect(const Ray &ray,
        SurfaceInteraction *isect) const {
```

```
    <<Compute initial parametric range of ray inside kd-tr
ee extent>>
    <<Prepare to traverse kd-tree for ray>>
    <<Traverse kd-tree nodes in order for ray>>
}
```

这个函数最终目标是寻找tMin和tMax.

```
<<Compute initial parametric range of ray inside kd-tree e
xtent>>=
Float tMin, tMax;
if (!bounds.IntersectP(ray, &tMin, &tMax))
    return false;
```

```
<<Prepare to traverse kd-tree for ray>>=
Vector3f invDir(1 / ray.d.x, 1 / ray.d.y, 1 / ray.d.z);
///数组数量就是树的最大深度
constexpr int maxTodo = 64;
KdToDo todo[maxTodo];
int todoPos = 0;
///KdToDo用于记录需要处理的节点
<<KdTreeAccel Declarations>>+=
struct KdToDo {
    const KdAccelNode *node;
    Float tMin, tMax;
};
```

```
<<Traverse kd-tree nodes in order for ray>>=
bool hit = false;
//遍历节点，每次循环检测一个节点
const KdAccelNode *node = &nodes[0];
while (node != nullptr) {
    <<Bail out if we found a hit closer than the current n
ode>>
    if (!node->IsLeaf()) {
        <<Process kd-tree interior node>>
    } else {
        <<Check for intersections inside leaf node>>
```

```
            <<Grab next node to process from todo list>>
        }
    }
    return hit;
```

如果出现了之前判断的图元已经把射线的最远距离缩短得很短的情况，即大包围盒套了很多小包围盒这种情况，则不用再往深处遍历了。当前子树直接返回。

```
    <<Bail out if we found a hit closer than the current node>
    >=
    if (ray.tMax < tMin) break;
```

下面处理内部节点，通过相交点，我们来决定两个子树到底哪个或者全部都参与接下来的射线检测

```
    <<Process kd-tree interior node>>=
    <<Compute parametric distance along ray to split plane>>
    <<Get node children pointers for ray>>
    <<Advance to next child node, possibly enqueue other child
    >>
```

根据节点的split axis，和split pos，计算分割平面。
首先回顾一下计算射线与平面交点的公式

$$t_1 = \frac{x_1 - o_x}{d_x}$$

```
    <<Compute parametric distance along ray to split plane>>=
    int axis = node->SplitAxis();
    //tPlane就是射线与分割平面相交的点
    Float tPlane = (node->SplitPos() - ray.o[axis]) * invDir[a
    xis];
```

之前讲过的一种优化，就是先访问左子树和右子树哪个子树的问题，这里也面临同样问题。

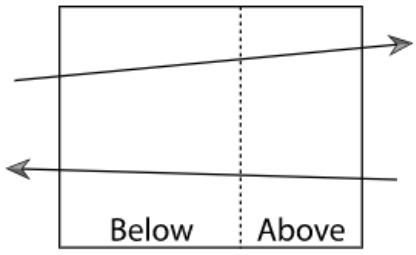


Figure 4.18: The position of the origin of the ray with respect to the splitting plane can be used to determine which of the node's children should be processed first. If the origin of a ray like r_1 is on the "below" side of the splitting plane, we should process the below child before the above child, and vice versa.

```
<<Get node children pointers for ray>>=
const KdAccelNode *firstChild, *secondChild;
//射线起点在split pos左边，或者如果与split pos重合的话，但是方向
是反的，也先处理below节点，即左子树
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;
}
```

左右两个子树都要处理的情况也许并不必要。下图展示了所有的情况

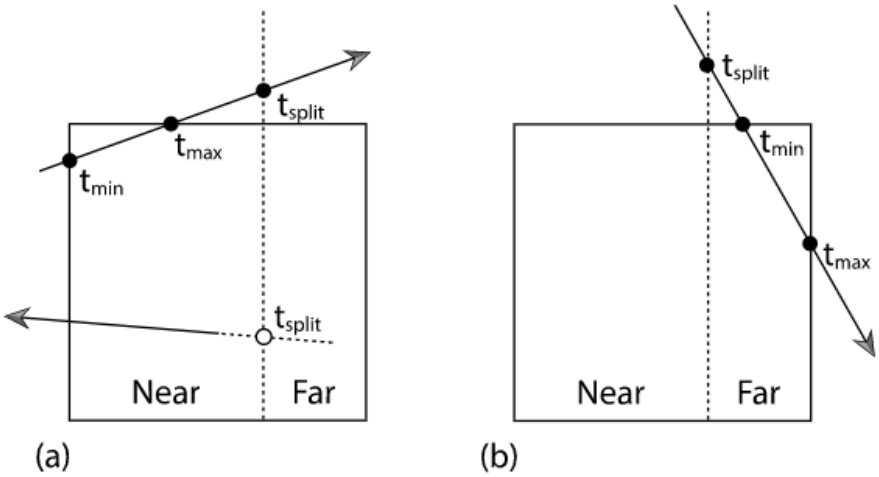


Figure 4.19: Two cases where both children of a node don't need to be processed because the ray doesn't overlap them. (a) The top ray intersects the splitting plane beyond the ray's t_{max} position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated 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.

```
<<Advance to next child node, possibly enqueue other child
>>=
//上图(a)中的情况，要么就是分割面比tMax还要远，或者射线原点在分割面
左侧，并且向左指
if (tPlane > tMax || tPlane <= 0)
    node = firstChild;
else if (tPlane < tMin)
    //图(b)的情况
    node = secondChild;
else {
    //两个子树全部都要进行处理
    <<Enqueue secondChild in todo list>>
    node = firstChild;
    tMax = tPlane;
}
<<Enqueue secondChild in todo list>>=
todo[todoPos].node = secondChild;
todo[todoPos].tMin = tPlane;
todo[todoPos].tMax = tMax;
++todoPos;
```

如果当前节点是叶子节点，则拿出图元进行相交测试

```
<<Check for intersections inside leaf node>>=
```

```
int nPrimitives = node->nPrimitives();
if (nPrimitives == 1) {
    const std::shared_ptr<Primitive> &p = primitives[node-
>onePrimitive];
    <<Check one primitive inside leaf node>>
} else {
    for (int i = 0; i < nPrimitives; ++i) {
        int index = primitiveIndices[node->primitiveIndice
sOffset + i];
        const std::shared_ptr<Primitive> &p = primitives[i
ndex];
        <<Check one primitive inside leaf node>>
    }
}
```

处理独立图元仅仅是一个很简单的调用

```
<<Check one primitive inside leaf node>>=
if (p->Intersect(ray, isect))
    hit = true;
```

执行完左节点，下面执行右节点

```
<<Grab next node to process from todo list>>=
if (todoPos > 0) {
    --todoPos;
    node = todo[todoPos].node;
    tMin = todo[todoPos].tMin;
    tMax = todo[todoPos].tMax;
}
else
    break;
```

与BVHAccel相似，KdTreeAccel也有一个用于shadow ray的检测函数IntersectionP()

```
<<KdTreeAccel Public Methods>>=
bool IntersectP(const Ray &ray) const;
```