PBRT: Primitives and Intersection Acceleration

Tuesday, November 10, 2020 11:35 PM

4 Primitives and intersection acceleration

The *Primitive* abstraction associates with an object everything that is not geometric about it. The geometric details of an object are there in its *Shape.* At the same time, *Primitive* is an interface to an underlying *Shape.*

4.1 Primitive interface and geometric primitives

The *Primitive* base class is the ***bridge*** between the ***geometry processing*** and ***shading subsystems***.

A world-space bounding box allows a Primitive to be added to an acceleration structure (via AABB union?).

Intersection methods in *Primitive* wrap Shape's intersection methods. Primitive initializes the *Interaction* data associated with the hit.

A *Primitive* may be emissive, a light source (or not).

IMPORTANT: only Primitives with a non-null *Material* participate in ray-intersection testing. An example of a *Primitive* that doesn't have a *Material* is that which delineates a space of participating media.

A SurfaceInteraction that describes the local geometry of an object at a point of intersection includes a model of how that geometry and material scatter light: a ***BSDF*** or ***Bidirectional Scattering Distribution Function***. Organic matter is best described by a ***BSSRDF*** or ***Bidirectional Scattering-Surface Reflectance Distribution Function***, which describe subsurface scattering.

4.1.1 Geometric primitives

One per Shape. Has an associated *Material.* If it's emissive, it has and *AreaLight* associated with it.

An interesting thing is that a *GeometricPrimitive* also describes its surroundings with a *MediumInterface* (a *MediumInterface* is also found in *Interaction*); to be precise, the participating media inside and outside of it.

IMPORTANT: When a *GeometricPrimitive* is intersected by a ray, the ray's *tMax* records the intersection point parametrically. Recall that ray intersections ignore *Shapes* that lie beyond the ray's *tMax*. Now, we want a ray to find the *closest Shape* in its direction that it intersect, so *multiple Shapes* will be tested against the ray until the closest one is found. By updating *tMax* with the parametric point of the most recent intersection, we'll disqualify *Shapes* that lie beyond this point in the subsequent tests.

4.1.2 Object instancing and animated primitives

Single model, instanced multiple times with a different transformation for each one.

Instancing makes a big difference in main memory consumption.

TransformedPrimitive is an instance of a *Primitive,* which in turn is the only representation in memory of

a given model. It wraps a *Primitive* and associates with it a *Transform* that places the instance in world space. It overrides the interface's methods to account for the effects of the *Transform.*

This is not to say that a *Primitive* can't have a *Transform* that places it in world space. In the case of a *TransformedPrimitive,* the wrapped *Primitive's Transform* doesn't transform it to world space, but from object space to the instance's coordinate system (and then the *TransformedPrimitive's Transform* transforms it to world space).

4.2 Aggregates

Without acceleration structures, every ray would need to be tested against every primitive in the scene in order to find the closest hit.

Goals of acceleration structures:

- Allow the quick, simultaneous rejection of groups of primitives.
- Order the search process so that nearby intersections are likely to be found first.

2 main approaches: ***spatial subdivision*** and ***object subdivision***.

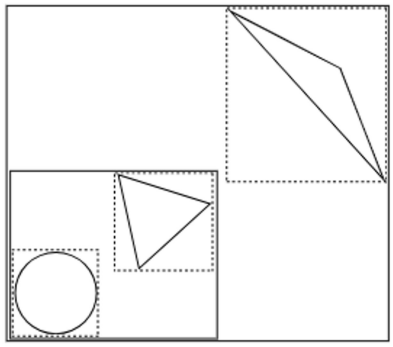
Spatial subdivision algorithms decompose 3D space into regions and record the objects that overlap each one of the regions. ***Adaptive subdivision*** further divides regions according to the number of objects that overlap with them: the more objects, the more subdivided the region is. Only the objects that overlap with regions that the ray intersects are tested for intersection. An example is PBRT's ***KdTreeAccel***.

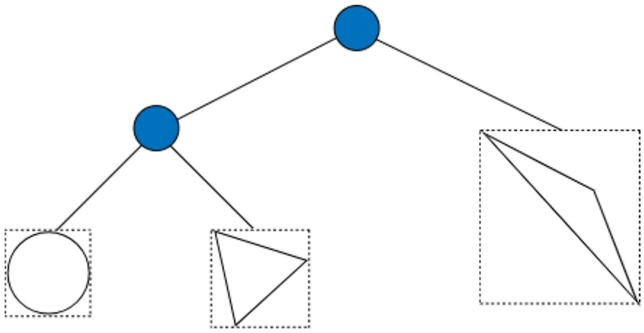
Primitive (or object) subdivision algorithms create bounding volumes around each primitive, around each constituent part of each primitive, and around groups of primitives. Only if the primitive's general volume is intersected by the ray, will its constituent parts be tested for intersection; otherwise, all of the parts are culled. Likewise, only if the group's volume is intersected by the ray, will the member primitives be tested for intersection; otherwise, they all get culled. An example is PBRT's ***BVHAccel***.

An *Aggregate* groups multiple *Primitives* together. It is itself a *Primitive,* so it inherits the intersection methods of that class.

4.3 Bounding Volume Hierarchies

A ***binary tree*** of bounding volumes with (the bounding volumes of) primitives at the leaves. The root is the bounding volume of the entire scene.





Every interior node stores a bounding box that contains all the primitives in nodes beneath it; these nodes, in turn, contain the bounding box that contains the ones beneath them.

With a BVH, a ray's traversal through the tree becomes its traversal of 3D space.

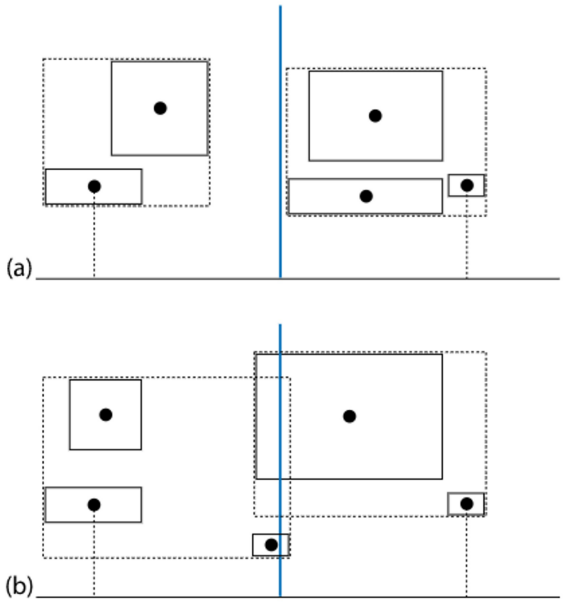
PBRT supports 4 different algorithms for building the hierarchy, that is, for subdividing primitives. They differ in effectiveness, efficiency at build-time, and parallelizability. These algorithms are referred to as ***split methods***, because they partition a set of primitives into 2 subsets: the ***surface area heuristic*** or ***SAH*** algorithm, the ***hierarchical linear bounding volume hierarchy*** or ***HLBVH***, the ***midpoint*** split method.

Primitive subdivision	***Space subdivision***
A given primitive is tested for intersection at most once (because it appears only once in the hierarchy).	A given primitive may potentially be tested in every region that it overlaps (unless ***mailboxing*** is used).
$2n - 1$ is the total number of nodes, where n is the number of primitives. This places an upper bound on memory usage.	
Faster to build.	
	Slightly faster ray intersection tests.
More numerically robust and less prone to missed intersections due to round-off errors.	

4.3.1 BVH construction

Each node has 2 children subtrees. Primitives get assigned to one child or the other based on the ***split method***. The problem with BVHs whose nodes have bounds that overlap is that a ray will intersect all of the overlapping nodes and will have to be tested against the corresponding primitives; for most of those primitives, the intersection tests will be unnecessary.

- ***Midpoint*** computes the midpoint of the interval *[minimum centroid coordinate, maximum centroid coordinate]* along the dominant axis. The midpoint creates 2 partitions of primitives.



- ***Equal counts*** puts exactly half of the primitives in one partition (the ones with smallest coordinate along the dominant axis) and the other half in the other.

- ***Surface area heuristic***. ***SAH*** is not just used in primitive subdivision algorithms; it is also used by spatial ones.

  With *midpoint* and *equal counts* there always remains the doubt of whether the resulting partitioning was optimal; there's simply no way to tell if another one might have been better. ***SAH*** estimates the computational cost of node traversal and ray-primitive intersection of a particular partitioning. By considering several ***candidate partitions***, the ***greedy*** algorithm then has the goal of minimizing the cost.

  The cost of intersecting a ***leaf node*** of a region is:

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

  where i indexes the primitives.

  The cost of intersecting an ***interior node*** of a region is ***probabilistic***:

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

  where t_{trav} is the cost of traversing the interior node to decide which child, A or B, to choose; p_A is the probability that the ray passes through the region of child A and p_B is the same but for child B.

  (Probability basics: a *random variable* is a variable whose values depend on outcomes of a random

phenomenon; *conditional probability* is a measure of the probability of an event occurring, given that another event, by assumption, presumption, assertion or evidence, has already occurred.)

Let C be an interior node whose region encompasses those of A and B. The ***conditional probability*** that a ray will intersect A given that it intersected C is the ***ratio of their surface areas***:

$$p(A|C) = \frac{s_A}{s_C}$$

$$p(B|C) = \frac{s_B}{s_C}$$

This makes sense, because the ratio approaches 1 as s_A approaches s_c. The larger the volume of C occupied by A is, the greater the conditional probability should be.

In the implementation, C is the interior node being contructed by the recursion, s_C is the surface area of the bounding box of C, and s_A and s_B are the surface areas of the bounding boxes of the 2 candidate partitions of primitives contained by the node.

The implementation computes the cost of all of the candidate partitions. It splits the extent between the 2 furthermost centroids along the dominant axis into 12 buckets and assigns primitives to buckets based on the position of their centroids along that axis. There are then 11 candidate partitions, one per interior bucket boundary, each boundary splitting the primitives into 2 partitions. The boundary that minimizes the SAH and the estimated cost of computation is chosen.

2 ***disadvantages:***

- Many passes are taken over the scene primitives to compute the SAH costs (the candidate partitions) at all of the levels of the tree.
- Top-down construction is difficult to parallelize well. There isn't much independent work to assign to parallel threads.

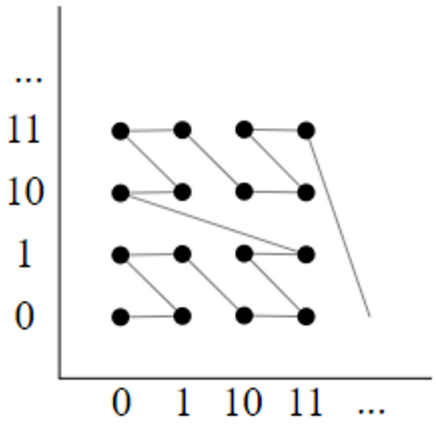
Linear BVHs are supposed to address these 2 issues.

4.3.3 Linear Bounding Volume Hierarchies

Linear BVHs address SAH's disadvantages by being linear in the number of primitives and by partitioning primitives into clusters that can be processed independently.

Points in 3D don't have an obvious sequence or order. LBVHs map them to the 1D line, where there exists an obvious order. The points that lie nearby along the line also lie nearby in 3D, forming clusters for parallel processing.

This ***sorting*** is accomplished by ***Morton codes***, which map nearby points in n dimensions to nearby points along the 1D line. The multi-line curve in 3D space that connects the points is called ***Morton curve***.

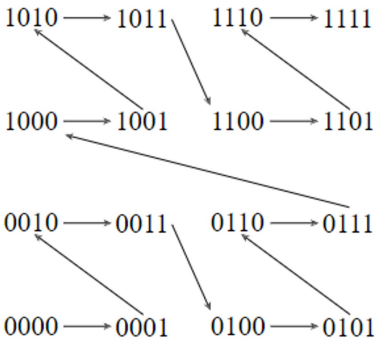


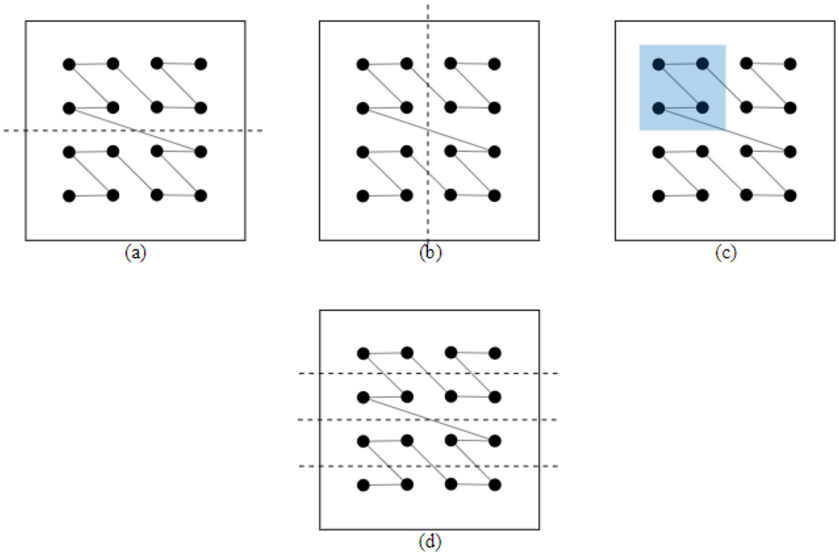
The ***Morton transformation*** takes a coordinate with ***integer*** components and ***interleaves*** the bits of the components, starting with leading bits: z's mth bit, y's mth bit, x's mth bit, z's $m-1th$ bit, y's $m-1th$ bit, etc.

$z_{m-1}y_{m-1}x_{m-1}\ldots z_0y_0x_0$

This is called a ***Morton encoding*** of the coordinate. This is accomplished by a bit-level left-shifting function.

Morton codes have nice properties that permit us to reason about the points and draw conclusions about them easily and efficiently. LBVHs use these properties to partition the primitives.





Like the ***midpoint method***, LBVHs also split primitives at the midpoint of the region. A variation called ***hierarchical*** LBVH or ***HLBVH*** does midpoint splitting by first building ***treelets*** for lower levels of the hierarchy and then building the top level using the SAH heuristic. Each treelet is a ***Morton cluster*** of nearby primitive centroids.

Take a binary string of 3 bits as example. Let the 2 high bits be 2^2 groups: 00X, 01X, 10X, and 11X . Each group has members XX0 and XX1.

Now take a binary string of 30 bits. Let the 12 high bits be 2^{12} groups. The remaining low 18 bits represent the 2^{18} members of each of the 2^{12} groups.

That is how sorted Morton codes group together coordinates into clusters. Each of the $2^{12} = 4096$ groups of coordinates is a cell of a grid imposed on 3-space. Each dimension has $2^4 = 16$ cells.

If you extend the 2^{12} bitmask to 2^{13} and keep going, you refine the granularity of the subdivision. At every extension of the bitmask, you are splitting the primitives into 2 treelets: the ones with 0 at that bit and the ones with 1 at that bit.

Bit index 17: XXXXXXXXXXXX0..... and XXXXXXXXXXXX1.....
Bit index 16: XXXXXXXXXXXXX0..... and XXXXXXXXXXXXX1.....
...
Bit index 0: XXXXXXXXXXXXXXXXXXXXXXXXXXXXX0 and XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1

The split is done at the midpoint of that bit's axis: since a Morton code interleaves the bits of the

coordinate's components ZYXZYX...ZYX, a different split axis is used every time the bit index decreases. Primitives where the bit's value is 0 go in one group and the ones with 1 go in the other group. If all of the primitives have the same value at the bit, the primitives won't be split along this axis at this level, and the algorithm proceeds to the next lower bit.

HLBVH partitioning based on Morton codes is done from bits 17 to 0. Once the HLBVH treelet structure is built at those granularity levels, the treelets are partitioned using SAH.

4.3.4 Compact BVH for traversal

Once the chosen split method is finished, the primitives are placed in a linear array in an order such that the primitives of BVH leaf nodes lie contiguously (depth-first order, an entire left subtree lies before the right subtree). This is said to ***flatten*** the tree; the flattening is done recursively on subtrees. The `LinearBVHNode` structures provide ***traversal information*****.**

4.4 Kd-Tree accelerator

- [ ] TODO. BVH will be enough for now.