Chap 9. Acceleration Algorithms

- o. Three performance goals for real-time rendering: more frames per second, higher resolution, and more (and more realistic) objects in the scene.
- o. The rendering of a Boeing-777 would include 132,500 unique parts and over 3,000,000 fastners, which would yield a polygonal model with over 500,000,000 polygons.
  ⇒ Acceleration algorithm will always be needed.
- o. A high frame rate and a steady frame rate are both important.
- o. If the frame rate is lower than the update frequency of the monitor (e.g., 85 Hz), the displayed images appear with unnatural motion. The ideal situation is to render images at the same frame rate as the monitor frequency, and that the rate should never vary.
- o. The core of many acceleration algorithms is based on spatial data structures, culling techniques, and level of detail techniques.

9.1 Spatial Data Structures

- o. Spatial data structures are usually organized hierarchically. The main reason for using a hierarchy is that different types of queries get significantly faster. The construction of most spatial data structures is expensive, and is usually done as a preprocess. Incremental updates are possible in real time.

o. Some types of spatial data structures are Bounding Volume Hierarchies (BVHs), variants of BSP-trees, and octrees. BSP-trees and octrees are data structures based on space subdivision. A bounding volume hierarchy is not a space subdivision. Rather, it encloses the regions of the space surrounding geometrical objects, and thus the BVH need not enclose all space. In addition to improving efficiency of queries, BVHs are also commonly used to describe model relationships and for controlling hierarchical animation

9.1.1 Bounding Volume Hierarchies

o. A Bounding Volume (BV) should be a much simpler geometrical shape than the contained objects, so that doing tests using a BV can be done much faster than using the objects themselves.

o. A BV does not contribute visually to the rendered image. Instead, it is commonly used to speed up rendering and different computations and queries.

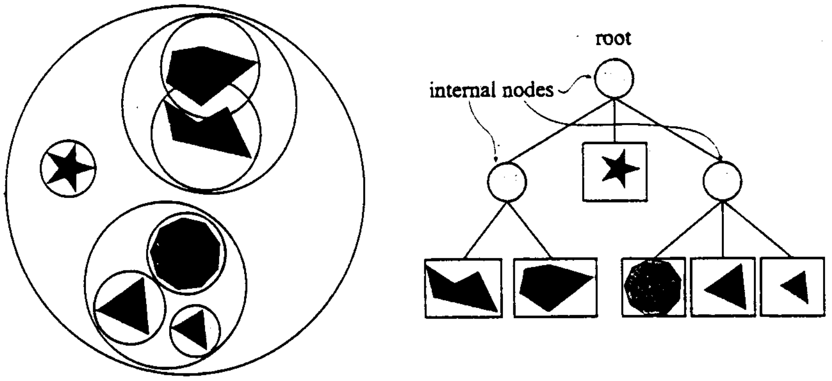

Figure 9.1. The left part shows a simple scene with six objects, each enclosed by a bounding sphere. The bounding spheres are then grouped together into larger bounding spheres until all objects are enclosed by the largest sphere. The right part shows the bounding volume hierarchy (tree) that is used to represent the object hierarchy on the left. The BV of the root encloses all objects in the scene.

9.1.2 BSP Trees

- Binary Space Partitioning (BSP) trees exist as two different variants: axis-aligned and polygon-aligned.
- If the trees are traversed in a certain way, then the geometrical contents of the tree can be sorted from any point of view. This is in contrast to BVHs, which do not include any type of sorting.

Axis-Aligned BSP Trees

- The whole scene is enclosed in an AABB and the box is recursively subdivided into smaller boxes.
- A node N is recursively visited. The plane of N is examined, and tree-traversal continues recursively on the side of the plane where the viewer is located. When this side has been traversed, we can start to traverse the other side.
- Traversal of the closer part of the tree can end when a box of a node is entirely behind the viewer.

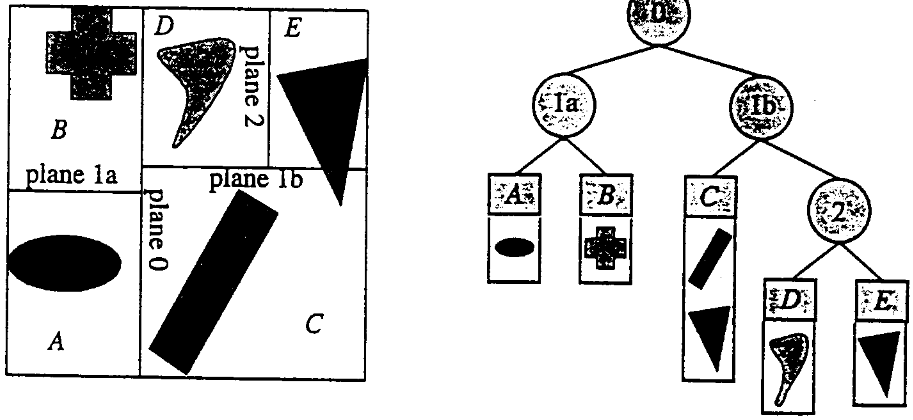

Figure 9.2. Axis-aligned BSP tree. In this example, the space partitions are allowed to be anywhere along the axis, not just at its midpoint. The spatial volumes formed are labeled A through E. The tree on the right shows the underlying BSP data structure. Each leaf node represents an area, with that area's contents shown beneath it. Note that the triangle is in the object list for two areas, C and E, because it overlaps both.

Polygon-Aligned BSP Trees

o. A polygon is chosen as the divider, splitting space into two halves. The plane in which the polygon lies is used to divide the rest of polygons in the scene into two sets. Now in each half-space of the dividing plane, another polygon is chosen as a divider, which divides only the polygons in its half-space. This is done recursively.

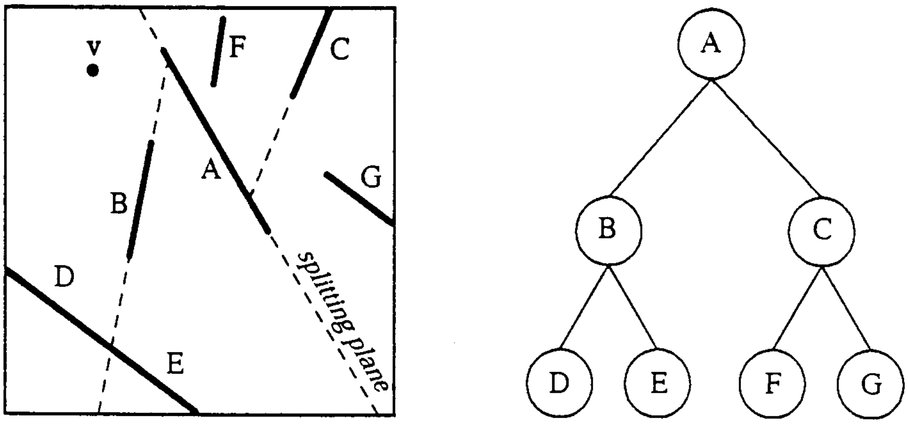

Figure 9.3. Polygon-aligned BSP tree. Polygons A through G are shown from above. Space is first split by polygon A, then each half-space is split separately by B and C. The splitting plane formed by polygon B intersects the polygon in the lower left corner, splitting it into separate polygons D and E. The BSP tree formed is shown on the right.

o. This type of BSP tree has useful properties. For a given view, the structure can be traversed strictly from back to front (or front to back).

o. The back-to-front order does not guarantee that one object is closer than another, rather it provides a strict occlusion order, a subtle difference. For example, polygon F is closer to v than polygon E, even though it is further back in occlusion order.

9.1.3 Octrees

o. An octree is constructed by enclosing the entire scene in a minimal axis-aligned box. The box is split simultaneously along all three axes, and the split point must be the center of the box. The procedure is recursive, and stop when a maximum level is reached or when there is fewer than a certain number of primitives.

o. Octrees can be used in the same manner as axis-aligned BSP-trees, and thus, handle the same type of queries. They are also used in occlusion culling algorithms.

o. Objects are stored in leaf nodes. Certain objects have to be stored in more than one leaf node. Another option is to place the object in the box that is the smallest that contains the object. This has a significant disadvantage in that a small object that is located in the center will be placed in the topmost node. One solution is to split the objects, but that introduces more objects. Another is to put a pointer to the object in each leaf box it is in, loosing efficiency and making octree editing more difficult.

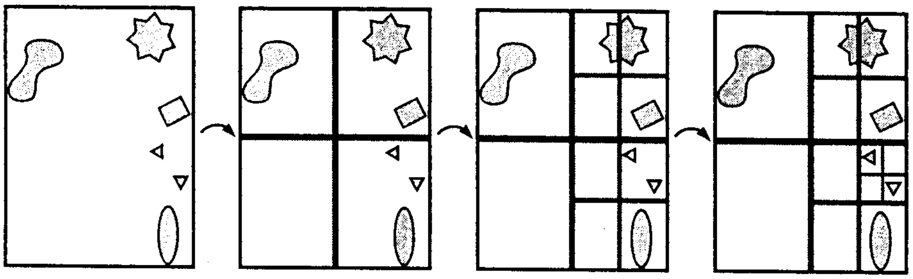

Figure 9.4. The construction of a quadtree (which is the two-dimensional version of an octree). The construction starts from the left by enclosing all objects in a bounding box. Then the boxes are recursively divided into four equal-sized boxes until each box (in this case) is empty or contains one object.

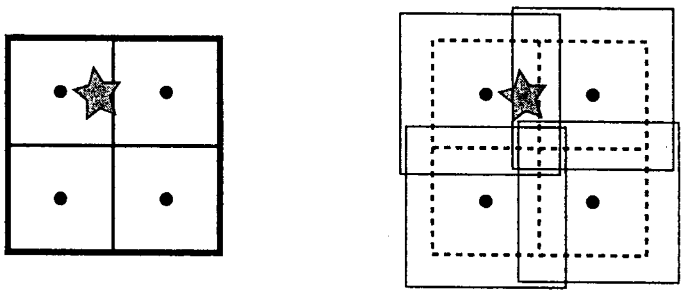
Figure 9.5. An ordinary octree compared to a loose octree. The black circles indicate the center points of the boxes (in the first subdivision). To the left, the star pierces through one splitting plane of the ordinary octree. Thus, one choice is to put the star in the largest box (that of the root). To the right, a loose octree with $k = 1.5$ (that is, boxes are 50% larger) is shown. The boxes are slightly displaced so that they can be discerned. The star can now be placed fully in the upper left box.

o、Ulrich[760] introduced a third solution, loose octree.
The choice of the size of each box is relaxed. If the side length of an ordinary box is l, then kl is used instead, where $k>1$.
The boxes' center points are the same. Using larger boxes, the number of objects that cross a splitting plane is reduced.
An object is always inserted into only one octree node, so deletion from the octree is trivial.

o、Some advantages accrue by using $k=2$. First, insertion and deletion of objects is $O(1)$. Knowing the object's size means immediately knowing the level of the octree it can successfully be inserted in, fully fitting into one loose box. The object's centroid determines which loose octree box to put it in.

o、This structure lends itself well to bounding dynamic objects, at the expense of some efficiency and the loss of a strong sort order when traversing the structure.

o、Often an object moves only slightly from frame to frame, so that the previous box still is valid the next frame.
Only a fraction of the objects in the octrees need updating each frame.

9.1.4 Scene Graphs

o. The scene graph is a higher level tree structure that is augmented with textures, transforms, levels-of-detail, render states (e.g., material properties), light sources, etc.

o. It is represented by a tree and is traversed in depth-first order to render the scene. For example, a light source can be put at an internal node, which affects only the contents of the subtree. Another example is when a texture is encountered in the tree. The texture can similarly be applied to all the geometry in that node's subtree.

o. A node in the scene graph often has a BV, and is thus quite similar to a BVH. A leaf in the scene graph stores geometry. However, one often allows this geometry to be encoded in any spatial data structure that is desired. Thus a leaf may hold an entire BSP-tree that stores the geometry of, say, a car.

o. One way of animating objects is to put transforms in internal nodes in the tree. Scene graph implementations then transform the entire contents of that node's subtree with that transform. Since a transform can be put in any internal node, hierarchical animation can be done.

o. Several nodes may point to the same child node. The tree structure is a Directed Acyclic Graph (DAG). Scene graphs are often DAGs because they allow for instantiation, i.e., we can make several copies (instances) of an object without replicating its geometry. (See Figure 9.6.). It is often only the leaf nodes, where most object data is located, that are shared. This simplifies handling of the scene graph.

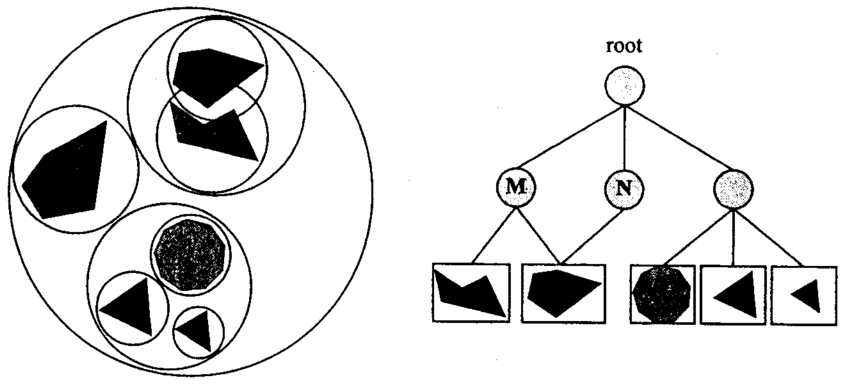

Figure 9.6. A scene graph with different transforms **M** and **N** applied to internal nodes, and their respective subtrees. Note that these two internal nodes also point to the same object, but since they have different transforms, two different objects appear (one is rotated and scaled).

- When objects are to move in the scene, the scene graph has to be updated. This can be done with a recursive call on the tree structure. Transforms are updated on the way from the root toward the leaves. The matrices are multiplied in this traversal and stored in relevant nodes. However, when transforms have been updated, the BVs are obsolete. Therefore, the BVs are updated on the way back from the leaves toward the root. A too-relaxed tree structure complicate these tasks enormously, and so DAGs are often avoided, or a limited form of DAGs are used, where only the leaf nodes are shared.
- More than one scene graph can be used for the same scene. This is the idea of spatialization, in which the user's scene graph is augmented with a separate scene graph created for a different task, e.g., faster culling and picking. The leaf nodes, where most models are located, are shared, so the expense of an additional set of internal nodes is minimized.

9.2 Culling Techniques

o. Backface culling eliminates polygons facing away from the viewer. View frustum culling eliminates groups of polygons outside the view frustum. Occlusion culling eliminates objects hidden by groups of of other objects. It is the most complex culling technique, as it requires an object or group of objects to gather and use information about other objects' locations.

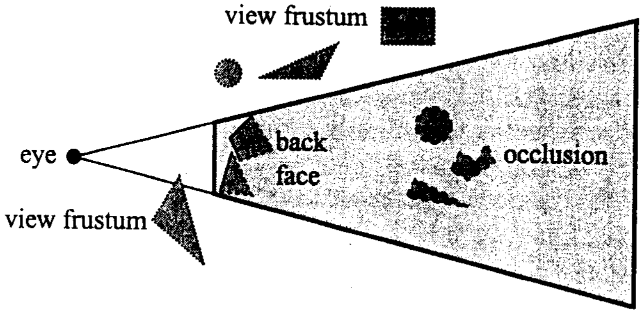

Figure 9.7. Different culling techniques. Culled geometry is dashed. *(Illustration after Cohen-Or et al. [135].)*

o. The ideal culling algorithm would send only the Exact Visible Set (EVS) of primitives through pipeline. (The EVS is defined as all primitives that are partially or fully visible.) However, it is time-consuming to determine the EVS. Instead, practical algorithms find the Potentially Visible Set (PVS), that is a prediction of the EVS. If the PVS fully includes the EVS, so that only invisible geometry is discarded, the PVS is said to be conservative. A PVS may also be approximate, in which the EVS is not fully included. This type of PVS may generate incorrect images. The goal is to make these errors as small as possible. When a PVS has been found, it is rendered using the z-buffer, which resolves the final visibility.

9.4 Hierarchical View Frustum Culling

o. Only primitives that are totally or partially inside the view frustum need to be rendered. The BV of each object is compared to the view frustum. If the BV is outside the frustum, the geometry inside can be omitted. These computations are done within the CPU. Thus the geometry inside the BV does not need to go through the rendering pipeline. If instead the BV is inside or intersecting the frustum, the contents of that BV may be visible and must be sent through the rendering pipeline.

o. By using a spatial data structure, the culling can be applied hierarchically. For a BVH, a preorder traversal from the root does the job.

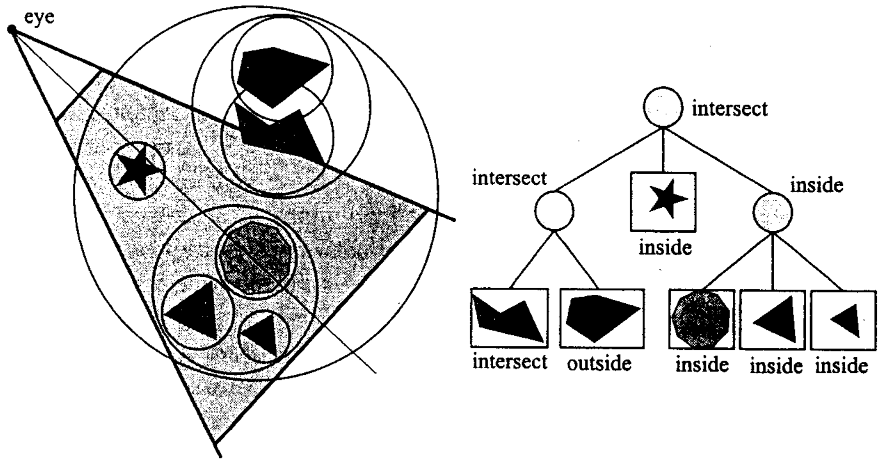

Figure 9.11. A set of geometry and its bounding volumes (spheres) are shown on the left. This scene is rendered with view frustum culling from the point of the eye. The BVH is shown on the right. The BV of the root intersects the frustum, and the traversal continues with testing its children's BVs. The BV of the left subtree intersects, and one of that subtree's children intersects (and thus is rendered), and the BV of the other child is outside and therefore is not sent through the pipeline. The BV of the middle subtree of the root is totally inside and is rendered immediately. The BV of the right subtree of the root is also fully inside, and the entire subtree can therefore be rendered without further tests.

- If the BV of any type of node is outside the frustum, that node is not processed further. The tree is pruned, since the BV's subtree is outside the view.
- If the BV intersects the frustum, the traversal continues and its children are tested. When a leaf node is found to intersect, its contents (i.e., its geometry) is sent through the pipeline. The primitives of the leaf are not guaranteed to be inside the view frustum. Clipping takes care of ensuring that only primitives inside the view frustum are being rendered.
- If the BV is fully inside the frustum, its contents must all be inside the frustum. Traversal continues, but no further frustum testing is needed for the rest of the tree.
- View frustum culling operates in the application stage (CPU). For large scenes or certain camera views, only a fraction of the scene might be visible, and it is only this fraction that needs to be sent through the rendering pipeline. In such cases, a large gain in speed can be expected. View frustum culling techniques exploit the spatial coherence in a scene, since objects are located near each other can be enclosed in a BV, and nearby BVs may be clustered hierarchically.
- Octrees and BSP trees can also be used for view frustum culling. These methods are usally not flexible enough when it comes to rendering dynamic scenes. That is, it takes too long to update the corresponding data structures when an object moves (an exception is loose octrees).

But for static scenes, these methods can perform better than BVHs.

o. BSP trees are simple to use for view frustum culling. If the box containing the scene is visible, the root node's splitting plane is tested. If the plane intersects the frustum, both branches of the BSP tree are traversed. If instead the view frustum is fully on one side of the plane, then whatever is on the other side of the plane is culled.

o. Octrees are also simple to use. Traverse the tree from the root and test each box in the tree. If a box is outside the frustum, traversal for the branch is terminated.

o. For view frustum culling, there is a simple technique for exploiting frame-to-frame coherency. If a BV is found to be outside a certain plane of the frustum in one frame, it will probably be outside that plane in the next frame too. An index to this plane is stored (cached) with the BV. In the next frame, the cached plane is tested first, and on average a speed-up can be expected.

o. If the viewer is constrained to only translation or rotation around one axis at a time from frame to frame, this can also be exploited for faster frustum culling. When a BV is found to be outside a plane of the frustum, the distance from that plane to the BV is stored with the BV. Now, if the BV translates, the distance to the BV can be updated quickly by knowing how much the viewer has translated. This can provide a generous speed-up in comparison to a naive view frustum culler.