Computation of Potentially Visible Set for Occluded Three-Dimensional Environments

Derek Carr
Advisor: Prof. William Ames
Boston College
Computer Science Department
May 2004

Abstract

This thesis deals with the problem of visibility culling in interactive three-dimensional environments. Included in this thesis is a discussion surrounding the issues involved in both constructing and rendering three-dimensional environments.

A renderer must sort the objects in a three-dimensional scene in order to draw the scene correctly. The Binary Space Partitioning (BSP) algorithm can sort objects in three-dimensional space using a tree based data structure. This thesis introduces the BSP algorithm in its original context before discussing its other uses in three-dimensional rendering algorithms.

Spatial Sorting

A naïve approach to rendering a scene is to iterate a vector of objects and draw each object in turn. Almost any environment drawn using this approach will result in an incorrect rendering. Imagine two disjoint objects along the same line of sight from the location of the eye. If the renderer draws the object furthest from the eye last, the object closest to the eye will not appear correct. The object that is furthest from the eye will appear to overlap the closer object. This basic example illustrates the need for sorting objects. A renderer must sort the objects in a three-dimensional scene in order to draw the scene correctly.

Using a depth buffer is the easiest approach to solving basic spatial sorting problems, but it is not optimal. The depth buffer approach is only applicable for those scenes that do not contain transparent objects; therefore, it is necessary for the renderer to utilize a more robust algorithm for complex meshes.

The Painter's A.

O(n log n). The renderer can achieve this level of performance by using the Binary Space Partitioning (BSP) data structure.

Binary Space Partitioning

The BSP data structure provides a computational representation of space that simultaneously acts as a search structure and a representation of geometry. The efficiency gain reaped by using the BSP algorithm occurs because it provides a type of "spatial sorting". The BSP tree is an extension of the basic binary search tree to dimensions greater than one. A BSP solves a number of problems in computational geometry by exploiting two simple properties that occur when a plane separates two or more objects. First, an object on one side of a plane cannot intersect any object on the opposite side. Second, given a viewpoint location, all objects on the same side as the eye can have their images rendered on top of the images of all objects on the opposite side. This second property allows the BSP tree to implement the Painter's Algorithm.

The creation of a BSP tree is a time intensive process that a BSP compiler produces once offline for a rigid environment. By pre-computing the BSP tree, an animation or interactive application can reap the savings as substantially faster algorithms for computing visibility orderings. Before diving into the algorithm that produces a BSP tree, it is necessary to enumerate the group of prerequisite functions.

The BSP compiler takes a set of polygons as input and recursively divides that set into two subsets until each subset contains a convex grouping of polygons. In order for a set of polygons to be convex, each polygon in the set must be in front of every other polygon in the set.

Figure 1. A convex vs. non-convex set of polygons.

The three following functions presented in pseudo-code test if a set of polygons is convex. All references to an epsilon are necessary in order to provide numerical stability when working with floating point precision representations of vertices.

```
FUNCTION classifyPoint(Plane plane, Vector3
```

all polygons that are in front of the partition plane as input for creating the nodes front child. The process works the same for the back child. The compiler splits those polygons that span the partition plane into two pieces, and places each piece in the correct subset of space.

The compiler must decide if it is better to have a balanced tree structure or a tree that minimizes polygon splitting. If the compiler creates too many additional polygons by choosing partitioning planes that split too many polygons, the hardware will have a difficult time rendering all the polygons. On the other hand, if a compiler strives to minimize polygon splitting, the tree data structure will be unbalanced to a certain side. Tree traversal costs will increase if the depth of the tree structure is not balanced. On average, a balanced tree will perform better than an unbalanced tree. Creating an optimal BSP tree, by either balancing the tree structure or minimizing polygon splitting, is an NP-

The following figures illustrate how the tree construction process works on a sample scene. The sample will work in two dimensions, but the same process extends to three dimensions. In two dimensions, the partitioning structure is the line. In three dimensions, the partitioning structure is the plane. For these images, each line segment represents a polygon whose surface normal points in the direction of the arrow.

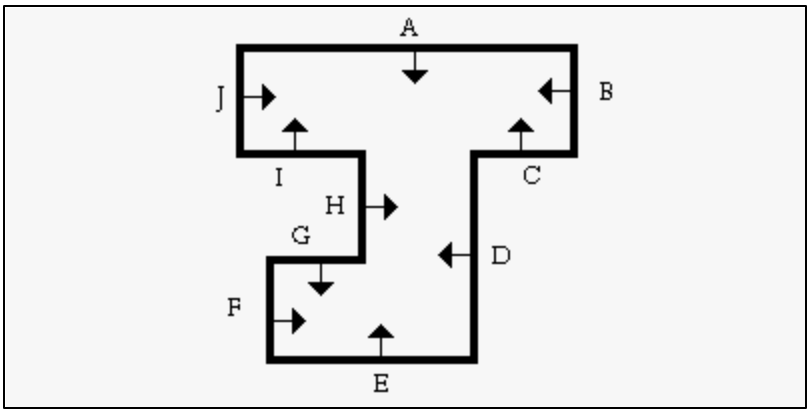

Figure 2. A simple scene.

Let us assume that the *selectPartitionPlane* function selected polygon G as the partitioning structure. The compiler classified all polygons against G. Since polygon D spanned the partition plane, the compiler split it into two segments. Polygons G, F, E, and D' are in front of G. Polygons A, B, C, D'', H, I, J are behind G.

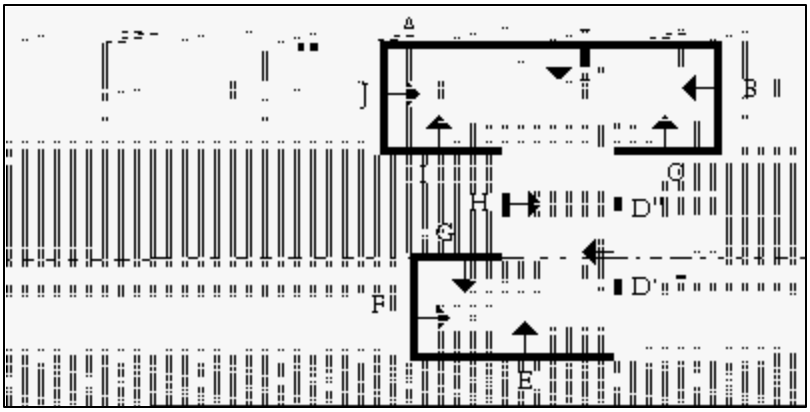

Figure 3. G is chosen as a partition.

The compiler continues with the set of polygons behind G. It selected polygon I as a partitioning structure.

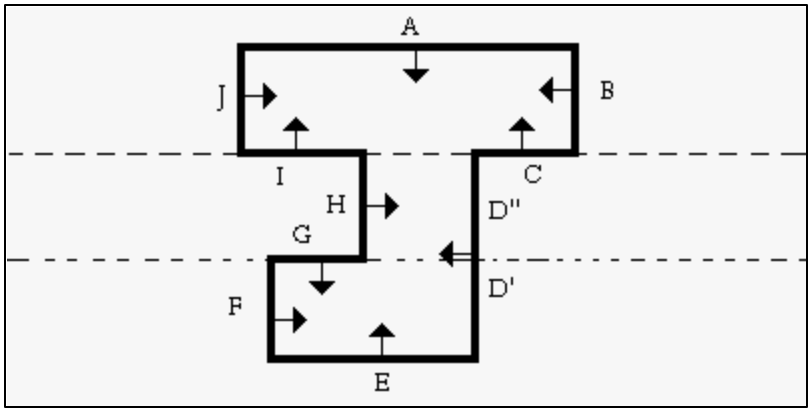

Figure 4. I selected as a partition.

The process continues until the partition is completed.

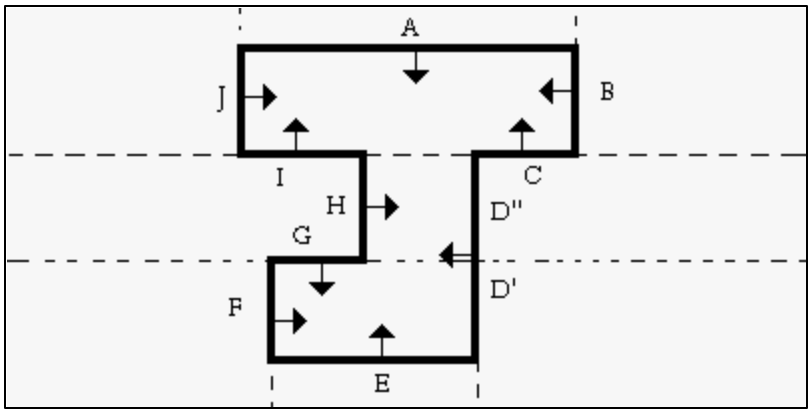

Figure 5. The final BSP tree.

The resulting data structure is a tree that has three leaves. Each leaf represents a convex grouping of polygons. Every polygon is in front of all other polygons in its leaf. Internal nodes store the partitioning plane, and the leaf nodes store the actual polygons.

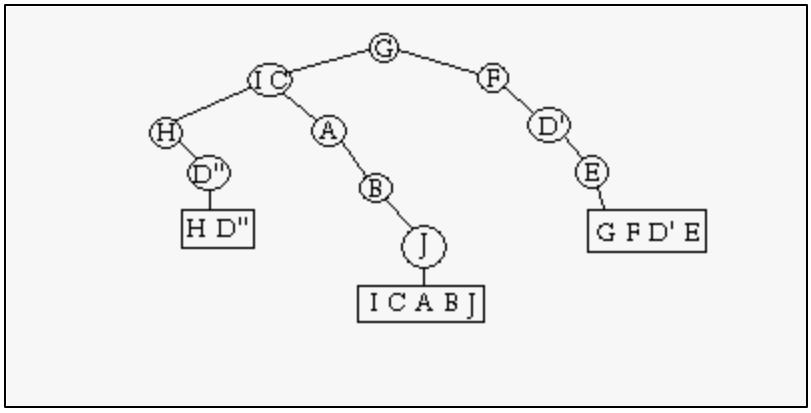

Figure 6. A BSP representation of the scene.

When a renderer has a BSP representation of the scene, the renderer can draw the scene in a back-to-front ordering that implements the Painter's Algorithm without having to perform the brute force spatial sorting algorithm discussed earlier. The rendering method takes advantage of the fact that given a viewpoint location, all objects on the same side as the eye can have their images rendered on top of the images of all objects on the opposite side.

The following is a rendering function that illustrates this concept.

```
FUNCTION renderTree (BspNode node, Point position)
        if ( isLeaf (node) )
          renderPolygons (node.polygons);
        else
          value = classifyPoint (node.plane, position);
          if (value == INFRONT || value == COINCIDENT)
            renderTree (node.backChild, position);
            renderTree (node.frontChild, position);
          else
            renderTree (node.frontChild, position);
            renderTree (node.backChild, position);
        return
```

Rendering our sampl

Users build CSG models by performing set operations and linear transformations on basic primitives such as spheres, cones, cubes, etc. A tree is used to represent the model. The leaves contain the basic primitives, while the nodes store operators or linear transformations. Union, subtraction, difference, and intersection are all valid operations. This thesis will focus on union and subtraction operations.

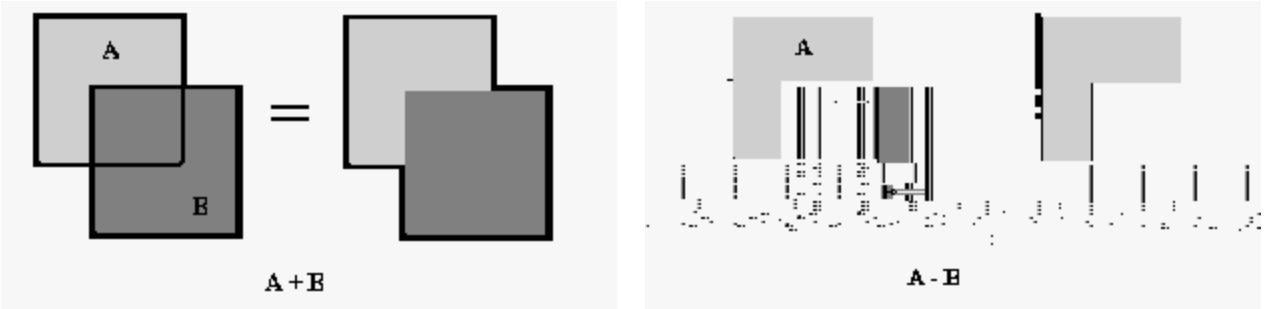

Figure 7. Sample of adding and subtraction two basic shapes.

Let us refer to A and B as brushes. In the example above, both brushes are convex shapes. The CSG implementatio

B that are not within the closed volume of

```
Polygon pos, neg, retFront, retBack;
splitPolygon (polygon, node.partitionPlane, pos, neg);
if (node.frontChild)
```

Potentially Visible Set (PVS) is the set of polygons that might be visible from a given location and field of view.

In their paper, *Portals and Mirrors*, Luebke and Georges describe a method of rendering occluded three-dimensional environments using a technique known as portal rendering. Portal rendering is an intuitive algorithm for rendering large architectural models that are closed, and exhibit a large amount of occlusion.

A portal-rendering engine works with sectors connected by portals. A sector is a polyhedral volume of space. It is useful to think of a sector as a room in a building. Each sector will contain a list of polygons. Most of the polygons will represent visible surfaces in the environment, such as walls, but a few polygons, called portals, will represent invisible regions of space that connect two adjacent sectors. Open doors and windows are good representatives of portal regions. Sectors can only "see" other sectors through the portals that connect them.

Representing architectural models using this data structure has some immediate benefits. All data related to a room is restricted to a specific part of the dataset, i.e. an individual sector. It is only possible to move from one sector to another by passing through a portal. Large environments, made up of many sectors, do not need to be completely in memory at one time. Advanced caching strategies are possible since not all sectors will be relevant, or visible, at any one time. A renderer only needs to render those sectors that are visible through a series of portals.

Figure 8.
View from a bedroom. White Boxes represent portals within the sector. Doorways and mirrors are portals [1].

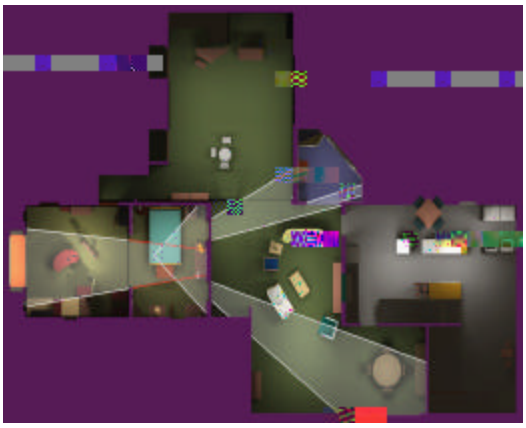

Figure 9.
This is an overhead view of the house containing the room in Figure 8. View frustum is clipped against each portal in order to calculate what sectors are visible [1].

We proceed to describe an algorithm that automatically divides an environment into a network of sectors and portals. It takes a BSP tree as input. This process is completed once for a given input. The following code is presented in C++ format, and is extracted from the implementation program that goes along with this paper.

```
void CSectorMgr::create(CBspNode* pBspNode)
{
        pBspRoot = pBspNode;
        createSectors();
        CPortalList* pPortalList = createPortalList();
        addPortalsToSectors(pPortalList);
        pPortalList-
```

Che ileavs if i iSP itee idefindan jontVex igrouping if ipolygons i ileavs coresepod mdirctoly po poe aectors pn pBrtalL] TJ 0

```
void CSectorMgr::createSectors()
{
        unsigned int x;
        numSector = pBspRoot->getNumLeaves();
        pSector = new CSector[numSector];
        for (x = 0; x < numSector; x++)
                pSector[x].addPolygon(pBspRoot->getLeafNode(x)-
>getPolygons());
}
```

Potential portals can only exist along the partition planes of the BSP data structure. We proceed by enumerating a list of all potential portals by creating a large portal polygon along each partition plane, and clipping that portal against all partitions in the BSP tree.

```
CPortalList* CSectorMgr::createPortalList()
{
	unsigned int x;
	unsigned int numPartitions = pBspRoot->getNumPartitions();
	unsigned int numPortals = numPartitions;
	CPortalList* pPortalList = new CPortalList();
	for (x = 0; x < numPartitions; x++)
	{
		CBspNode* pBspNode = pBspRoot->getPartitionNode(x);
		CPortal* pPortal = createLargePortal(pBspNode);
		pPortal->partitionId = x;
		pPortalList->add(pPortal);
	}
	for (x = 0; x < numPartitions; x++)
	{
		CBspNode* pNode = pBspRoot->getPartitionNode(x);
		unsigned int i = numPortals-1;
		bool hasMorePortals = true;
		while (hasMorePortals)
		{
			CPortal* pPortal = pPortalList->get(i);
			int retval = classifyPortal(pPortal, pNode->plane);
			if (retval == ALPHA_SPANNING)
			{
				CPortal* pFront = new CPortal();
				CPortal* pBack = new CPortal();
				bool retsplit = pPortal->splitPolygon(pNode->plane, pFront,
pBack);
				if (retsplit)
				{
					pFront->partitionId = pPortal->partitionId;
					pBack->partitionId = pPortal->partitionId;
					pPortalList->add(pFront);
					pPortalList->add(pBack);
					pPortalList->remove(i);
					numPortals = numPortals+1;
				}
				else
				{
					delete pFront;
					delete pBack;
				}
			}
			if (i != 0)
				i--;
			else
				hasMorePortals=false;
		}
	}
	for (x = 0; x < numPortals; x++)
	{
		CPortal* pPortal = pPortalList->get(x);
		pPortal->id = x;
	}
	return pPortalList;
```

```
}
```

```
CBspNode* CBspNode::getLeafNode(CVector3 &location)
{
	if (isLeaf())
		return this;
	int retval = plane.classifyPoint(location);
	if (retval == ALPHA_BEHIND)
	{
		if (pBack)
			return pBack->getLeafNode(location);
		else
			return 0;
	}
	if (pFront)
		return pFront->getLeafNode(location);
	else
		return 0;
}
```

```
void CSectorMgr::renderSector(int flags, unsigned int sectorId, unsigned int prevSectorId,
CFrustum* pFrustum, CVector3 &location)
{
	if (sectorId >= numSector)
		return;
	if (pSector[sectorId].beingVisited)
		return;
	unsigned int i;
	pSector[sectorId].beingVisited = true;
	for (i=0; i < pSector[sectorId].portalList.size(); i++)
	{
		CPortal* pPortal = pSector[sectorId].portalList.get(i);
		if (pFrustum->polygonInFrustum(pPortal))
		{
			if (pPortal->toSectorId != prevSectorId)
```

Another useful extension to representing an environment via a network of cells and portals is efficient collision detection. Given a vector of motion defined by a start and end position, we test for collision via a recursive method. Before beginning the method, we must first find the sector that contains the origin of the motion vector. The BSP tree can be used to find this sector similar to how it is used in the first step of the rendering function.

Given the sector that contains the origin of the motion vector, the simulation engine first tests if the vector intersects any of the portals in that sector. If an intersection exists, the collision procedure continues by testing for collision with the adjacent sector to the intersected portal. Since each sector is convex, if a motion vector intersects a portal polygon, it cannot possibly intersect any other polygon in that sector. If a collision with a portal polygon is not present, the simulation engine then tests for collision against the motion vector and all other polygons in the sector. If a collision is found, the motion vector is restricted to not allow the motion to extend outside of a wall.

Using this approach as a solution for collision detection in architectural models, it is necessary for only a small subset of polygons to be tested for collision against a given motion vector.

be created with CSG expressions, and then fed to the portal-sector algorithm to allow a real-

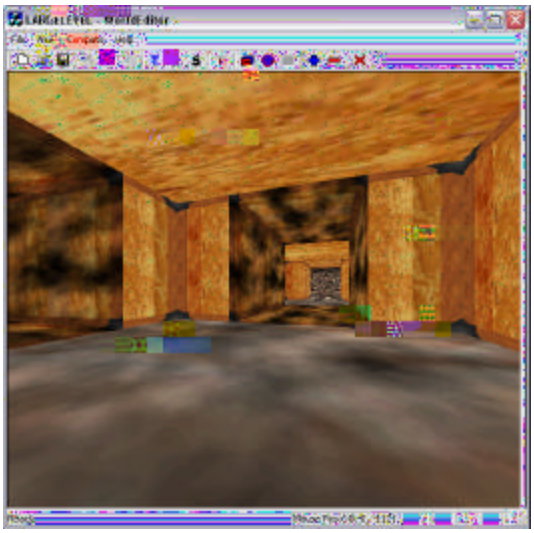

Figure 10.
A rendering of a BSP tree.

Figure 11.
Wireframe rendering showing large amount of unnecessary polygons being rendered.

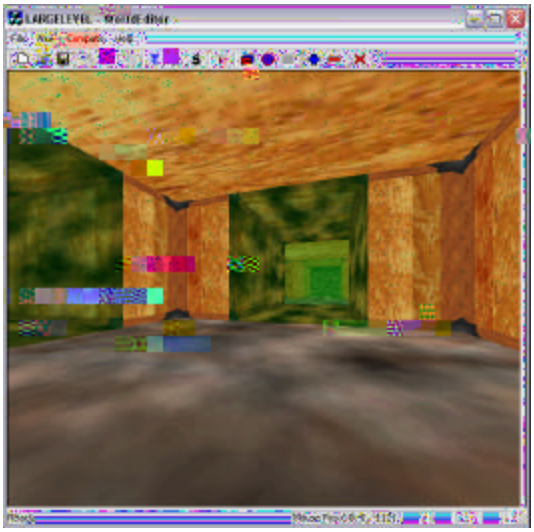

Figure 12.
Same rendering as above but using portal technique. Green Regions are portal polygons.

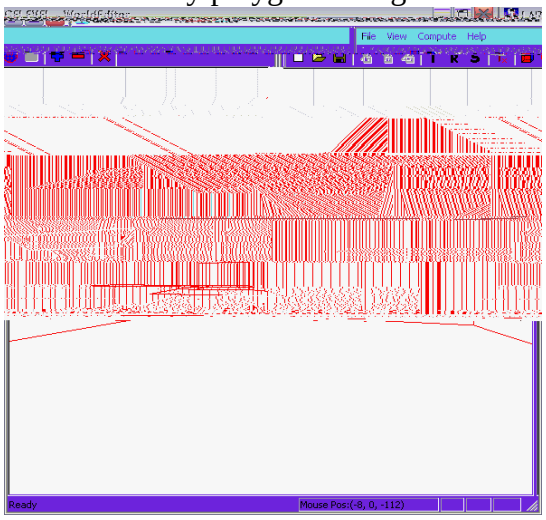

Figure 13.
Wireframe rendering showing the reduction in polygons rendered using portal technique.

Bibliography

1. Luebke, D. and Georges, C. *Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets.* Department of Computer Science, University of North Carolina at Chapel Hill, Apri.5 4 Tc 12 0 0 12 126 6574tT 12 mh5 Tm/9.75 T-15 (and) -15 (0-16 nd -