

Computation of Potentially Visible Set for Occluded Three-Dimensional Environments

Author: Derek W. Carr

Persistent link: <http://hdl.handle.net/2345/4426>

This work is posted on [eScholarship@BC](#),
Boston College University Libraries.

Boston College Electronic Thesis or Dissertation, 2004

Copyright is held by the author, with all rights reserved, unless otherwise noted.

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.

Constructive Solid Geometry (CSG) is an efficient interactive modeling technique that enables an artist to create complex three-dimensional environments by performing Boolean set operations on convex volumes. After providing a general overview of CSG, this thesis describes an efficient algorithm for computing CSG expression trees via the use of a BSP tree.

When rendering a three-dimensional environment, only a subset of objects in the environment is visible to the user. We refer to this subset of objects as the Potentially Visible Set (PVS). This thesis presents an algorithm that divides an environment into a network of convex cellular volumes connected by invisible portal regions. A renderer can then utilize this network of cells and portals to compute a PVS via a depth first traversal of the scene graph in real-time. Finally, this thesis discusses how a simulation engine might exploit this data structure to provide dynamic collision detection against the scene graph.

Representation of Three-Dimensional Objects

The optimal method of representing a three-dimensional object in computer graphics remains an unsolved problem. Researchers have developed a variety of representation schemes. Polygonal meshes, bi-cubic parametric patches, spatial subdivision techniques, and implicit mathematical representation of an object are all techniques used in representing three-dimensional objects.

The polygonal mesh has become the de facto standard for machine representation of three-dimensional objects in most interactive simulations. Its popularity is a result of both efficient algorithms and hardware that are able to consume and render a large amount of polygonal facets at sufficient frame rates. All references in this thesis to a three-dimensional environment are synonymous with a set of polygonal meshes.

Polygonal meshes approximate the shape of an object as a net of planar polygonal facets. It is possible to approximate an object of any shape using this approach. A mesh can represent a cube or pyramid efficiently, but as shapes exhibit greater curvature or detail, the amount of facets needed to reflect the shape of the object drastically increases. It is common for rich environments to require millions of polygons in order to achieve a reasonable level of realism. An increase in scene complexity will typically result in an increase in the number of polygons needed to represent the environment. It is common for modern day simulations to utilize environments that consist of millions of polygons. User expectations continue to outpace the hardware's capability to render interactive complex scenes. For this reason, a rendering engine utilizes a variety of visibility culling and spatial subdivision algorithms to meet user expectations.

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 Algorithm is the general technique of rendering those objects furthest from the eye first. Those objects closer to the eye will write over any objects previously drawn; this is similar to how a painter paints the background first before painting the foreground objects on top. In order to produce a final image, a painter may paint a part of the canvas multiple times.

A brute force computation of spatial relations between n polygons requires the renderer to compare every pair of polygons; hence, it is an $O(n^2)$ operation. For large scenes that contain 10^5 polygons, a proper sorting would require 10^{10} operations. This is not optimal; the renderer can substantially reduce the number of operations to

$O(n \log n)$. 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.

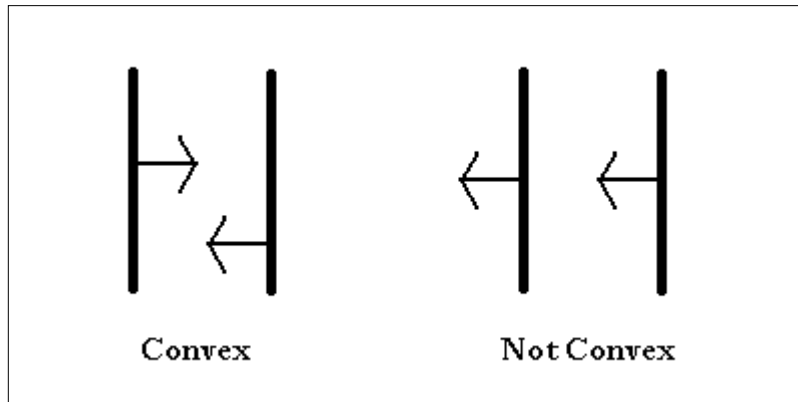


Figure 1. A convex vs. non-convex set of polygons.

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 point)
    value = plane.normal.dotProduct(point) + plane.distance;
    If (value < -epsilon) then return BEHIND;
    If (value > epsilon) then return INFRONT else return BEHIND;

```

```

FUNCTION classifyPolygon(Plane plane, Polygon polygon)
    numPositive = numNegative = 0;
    For each point p in polygon
        value = classifyPoint(plane, polygon);
        If (value == INFRONT) then numPositive++;
        If (value == BEHIND) then numNegative++;
    If (numPositive > 0 && numNegative == 0) then return INFRONT;
    If (numPositive == 0 && numNegative > 0) then return BEHIND;
    If (numPositive > 0 && numNegative > 0) then return SPANNING;
    return COINCIDENT;

```

```

FUNCTION isConvexSet(PolygonSet polygonSet)
    For i = 0 to polygonSet.length
        For j = 0 to polygonSet.length
            If (classifyPolygon(polygonSet[i].plane, polygonSet[j]) == BEHIND) then
                return false;
    return true;

```

During each recursive step of the BSP tree creation process, the compiler chooses a partitioning plane to divide the input set of polygons into two sets. The compiler feeds 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-complete problem. Since the BSP creation process is executed once offline for every scene to be rendered, an efficient compiler might choose to enumerate a large amount of possible tree structures in order to find a more optimal data structure.

In practice, compilers use heuristics in order to produce trees that tend to be balanced or minimize polygon splitting. The function *selectPartitionPlane(PolygonSet p)* returns a plane to partition the set of polygons. A simple approach that will most often produce an efficient tree is to select the plane of the first polygon in the set as the partition plane. In general, most compilers pool of potential partitioning planes is the planes of the polygons in the polygon set. By using this approach, a BSP tree can be used to perform collision

detection and clipping procedures efficiently since the partition planes are coincident with the polygons.

Before presenting the BSP tree creation function, it is necessary to understand how the tree can divide polygons into subsets. There are two commonly used methods. The first, and simplest approach, is to store all polygons coincident with a node's partitioning plane in the internal nodes of the tree. Each node will contain at least one polygon, and each path down the tree describes a convex set of polygons. The second approach is to store all polygons only at leaf nodes. The compiler flags those polygons that are coincident with a node's partitioning plane, and pushes those nodes down the front side of the tree. The *selectPartitionPlane* function makes sure that it never uses that polygon's partition plane again. This approach creates a leafy BSP tree. The leaves of the BSP tree contain a convex set of polygons enclosed by the planes defined by the path from the root of the tree to a leaf node.

The following is pseudo-code that creates the BSP tree.

```

FUNCTION createBspTree (BspNode node, PolygonSet polygons)
  If (isConvexSet(polygons)) then
    node.polygonSet = polygons;
  Else
    node.partitionPlane = selectPartitionPlane(polygons);
    frontSet = null;
    backSet = null;
    For each polygon P in polygons
      value = classifyPolygon(node.partitionPlane, P);
      If (value == INFRONT || value == COINCIDENT) then
frontSet.union(P);
      If (value == BEHIND) then backSet.union(P);
      If (value == SPANNING) then
        splitPolygon(P, partitionPlane, frontPart, backPart);
        frontSet.union(frontPart);
        backSet.union(backPart);
    createBspTree(node.frontChild, frontSet);
    createBspTree(node.backChild, backSet);

```



```
return;
```

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.

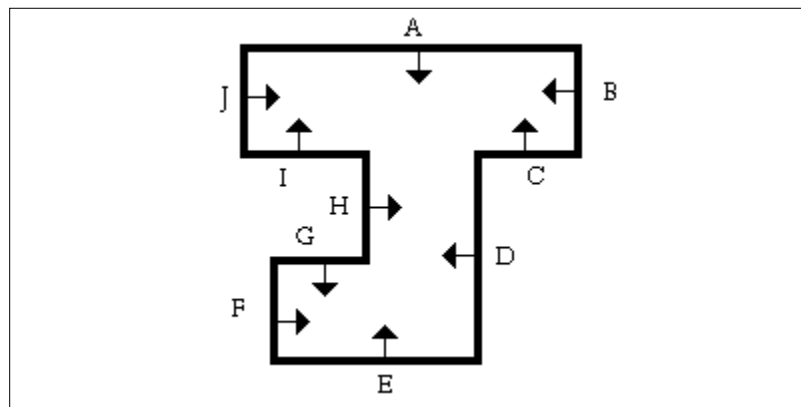


Figure 2. A simple scene.

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.

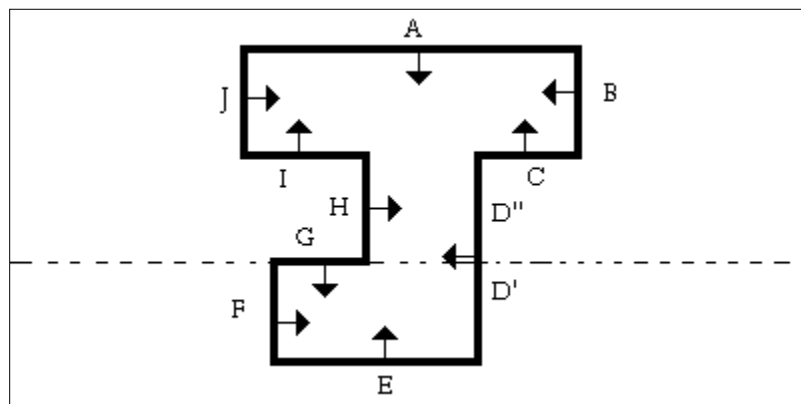


Figure 3. G is chosen as a partition.

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

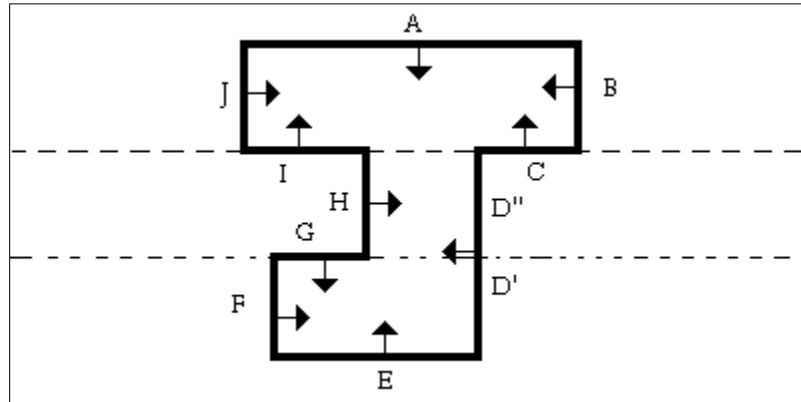


Figure 4. I selected as a partition.

The process continues until the partition is completed.

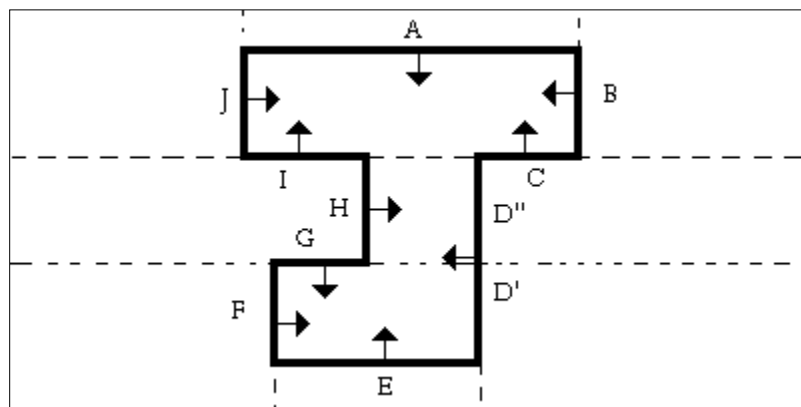


Figure 5. The final BSP tree.

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.

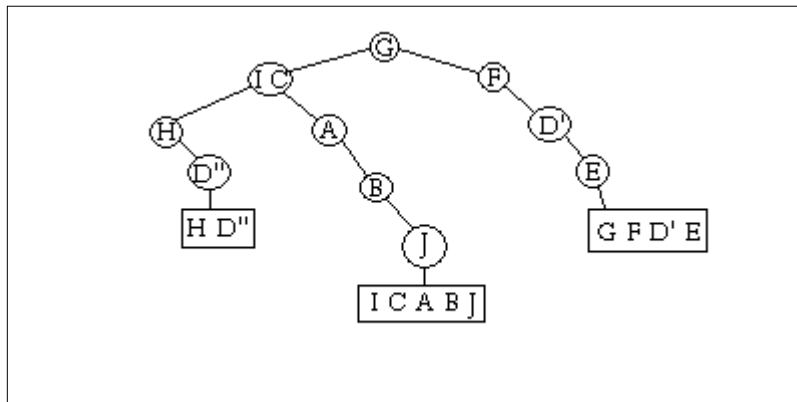


Figure 6. A BSP representation of the scene.

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 sample scene, let us imagine the eye to be in the sector containing G, and looking into the other two sectors. The render function begins with the root and compares the position of the eye against G. Since the eye is in front of G, the rendering function attempts to render all content behind G first, before rendering those polygons in front of G. Since the eye is behind I, the function renders all content in front of I before rendering the nodes behind it. Hence, the BSP tree spatially sorted polygons with respect to the viewpoint location. The renderer will render the polygons in the following order: I, C, A, B, J, H, D'', G, F, D', E.

The renderer must draw all polygons in the scene, even if those polygons are not visible. It would be better never to send those polygons that are not in the viewing frustum to the renderer at all. Visibility culling is the process of removing polygons from the entire set that are not visible before passing the set of polygons to a renderer. This is a non-trivial task. Before addressing this problem, it is important to know how a user can design three-dimensional environments efficiently.

Constructive Solid Geometry

Constructive Solid Geometry (CSG) is an efficient interactive modeling technique that enables an artist to create complex three-dimensional environments by performing Boolean set operations on convex volumes. CAD programs have long used CSG modeling techniques. CSG does not require the user to work with polygons directly; instead, it produces a polygon model after the user completes their design. CSG is a

high-level representation of an object. It describes the shape of an object and a history of how that shape was constructed.

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.

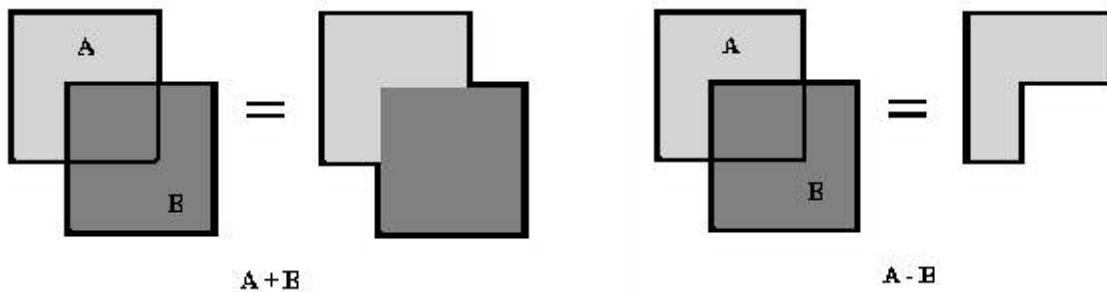


Figure 7. Sample of adding and subtraction two basic shapes.

Let us refer to A and B as brushes. In the example above, both brushes are convex shapes. The CSG implementation represents brushes as groups of polygons. Let us define some rules for how a CSG implementation could perform basic set operations on these convex shapes.

Union:

The union of brush A and B is equivalent to clipping all polygons of brush A that are within the closed volume of brush B and clipping all polygons of brush B that are within the closed volume of brush A.

Subtraction:

Subtracting brush B from A is a three-part process. First, clip all polygons of brush A that are within the closed volume of brush B. Second, clip all polygons of brush B that are not within the closed volume of brush A. Third, reverse the surface normal of the polygons that survive the clip operation from step two.

Similar rules for intersection and difference exist, but when modeling large, closed three-dimensional environments, these operations are not as useful to the artist. The primary function needed to implement these operations is a clipping function. CSG implementations need to be able to clip brushes against each other. Since the result of an operation on two convex primitives might not be convex, a CSG package needs a way of performing clipping operations against these results. In order to do this, a CSG package creates a BSP representation of each brush and performs the clipping procedure against a BSP tree.

The following function clips a polygon against a BSP tree. It takes as input an argument specifying if polygons within or without a BSP volume should be clipped.

```
FUNCTION clip (BspNode node, Polygon polygon, int side)
  If (isLeaf() )
    If (side == INFRONT)
      return null;
    else
      return polygon;
  int value = classifyPolygon(node.partitionPlane, polygon);
  if (value == COINCIDENT)
    value = side;
  switch (value)
  case INFRONT:
    if (node.frontChild)
      return clip(node.frontChild, polygon, side);
    else if (side == INFRONT)
      return null;
    else return polygon;
  break;
  case BEHIND:
    if (node.backChild)
```

```

    return clip(node.backChild, polygon, side);
else if (side == BEHIND)
    return null;
else return polygon;
break;
case SPANNING:
    Polygon pos, neg, retFront, retBack;
    splitPolygon (polygon, node.partitionPlane, pos, neg);
    if (node.frontChild)
        retFront = clip(node.frontChild, pos, side);
    else if (side == INFRONT) retFront = null;
    else retFront = pos;
    if (node.backChild)
        retBack = clip(node.backChild, neg, side);
    else if (side == BEHIND) retBack = null;
    else retBack = neg;
    return retBack + retFront;
break;
return null;

```

The modeling program completed to go along with this thesis implements a BSP based method of performing CSG operations. The application allows a user to work with geometric primitives, such as spheres and boxes, and the set operations of union and subtraction to build large complex environments. Users can apply scaling, rotation, and translation operations on the shapes. Once the user constructs an environment, the application computes the CSG expression defining the model and produces a BSP representation of the environment suitable for an interactive simulation.

Potentially Visible Set

A BSP tree alone is not a suitable data structure for world representation. As a search structure, the BSP tree is optimal, and highly scalable, but as a rendering data structure, BSP trees are not scalable for large environments. Rendering with a BSP tree requires all polygons in a scene to be drawn. For large environments, it is not computationally feasible for all polygons to enter the rendering pipeline. In order to

render three-dimensional environments efficiently, we need a data structure whose rendering performance is not related to the size of the world.

The secret to rendering large three-dimensional environments efficiently is to render only those polygons that are potentially visible from any field of view. The 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].

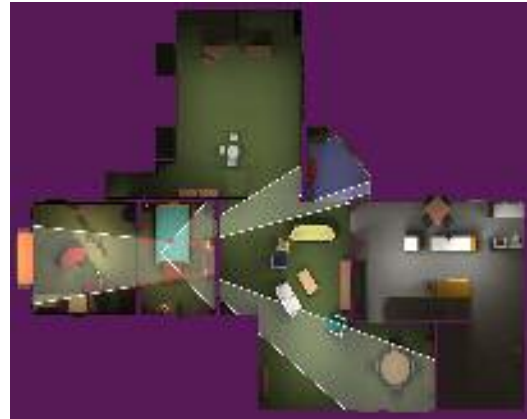


Figure 9.
This is an overhead view of the house containing the room in Figure 8. View frustum is clipped against each portal in order to calculate what sectors are visible [1].

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->clear();
    delete pPortalList;
    findTruePortals();
}
```

The leaves of a BSP tree define a convex grouping of polygons. These leaves correspond directly to the sectors in a portal-cell graph. The BSP creation process creates the sectors for us.

```
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;
}

```

Once a list of potential portals has been created, each with an unique id, portals are added to their corresponding sectors by pushing each portal down the BSP tree to a leaf node.

```

void CSectorMgr::addPortal(CPortal* pPortal, CBspNode* pBspNode)
{
    if (pBspNode->isLeaf())
    {
        pSector[pBspNode->id].addPortal(pPortal);
        return;
    }

    int retval = classifyPortal(pPortal, pBspNode->plane);
    if (retval == ALPHA_INFRONT)
    {
        if (pBspNode->pFront)
            addPortal(pPortal, pBspNode->pFront);
    }
    if (retval == ALPHA_BEHIND)
    {
        if (pBspNode->pBack)
            addPortal(pPortal, pBspNode->pBack);
    }
    if (retval == ALPHA_COINCIDING)
    {
        if (pBspNode->pFront)
            addPortal(pPortal, pBspNode->pFront);

        if (pBspNode->pBack)
            addPortal(pPortal->clone(), pBspNode->pBack);
    }
}

void CSectorMgr::addPortalsToSectors(CPortalList* pPortalList)
{
    unsigned int numPortals = pPortalList->size();

    for (unsigned int x = 0; x < numPortals; x++)
    {
        CPortal* pPortal = pPortalList->get(x)->clone();
        addPortal(pPortal, pBspRoot);
    }
}

```

Once portals are added to their corresponding sectors, sector ids correspond with BSP tree ids, there will be a large amount of invalid portals in each sector.

```
void CSectorMgr::findTruePortals()
{
    checkForSinglePortals();
    makePortalsInward();
    removeExtraPortals();
}
```

The first step to removing unnecessary portals is to cycle through each sector's list of portals and test if another sector contains a portal with the same id. If a matching portal exists, then we know that these two sectors are connected by these portals, otherwise, the portal is invalid. This is implemented in the *checkForSinglePortals()* method call.

Once all unnecessary portals are removed, it is necessary to make sure that each portal's surface normal faces into the sector that contains it. This is performed via the *makePortalsInward()* call. Finally, there are some extra portals that are in multiple sectors, but are not valid portals. These portals are coplanar with the walls in the mesh. The *removeExtraPortals()* method removes these extraneous portals.

In order to render using this data structure, we first find the leaf in the BSP tree that contains the location of the eye. We then render that sector via a recursive method that will render all sectors visible from the view frustum. With each recursive step through a portal, the renderer clips the viewing frustum against the portal. Using this method, only those sectors that are visible will be drawn. We first define a method of locating what sector contains a given point in space by using the following BSP method.

```

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;
}

```

This is our main rendering function. It takes a frustum and eye location as input. It uses the BSP tree to locate what sector the eye is located in, and then it calls the recursive *renderSector()* function with the current sector of the eye as input.

```

void CSectorMgr::render(int flags, CFrustum* pFrustum, CVector3 &location)
{
    CBspNode* pNode = pBspRoot->getLeafNode(location);
    if (!pNode)
        pBspRoot->render(flags, location);
    else
        renderSector(flags, pNode->id, pNode->id, pFrustum, location);
}

```

```

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)
            {
                CFrustum* pNewFrustum = pFrustum->adjust(pPortal, location);
                renderSector(flags, pPortal->toSectorId, sectorId, pNewFrustum,
location);
                delete pNewFrustum;
            }
        }
    }
    // render polygons
    CPolygon* pPolygon = pSector[sectorId].pPolygon;
    while (pPolygon != 0)
    {
        pPolygon->render(flags);
        pPolygon = pPolygon->pNext;
    }
    pSector[sectorId].beingVisited = false;
}

```

The above recursive method works by first setting a *beingVisited* flag that will be used to stop any possibility of infinite recursion. The method proceeds to loop through all portals in the current sector and tests if that portal is in the current view frustum. If the portal is in the frustum, the frustum is clipped against the portal, and the method is called again on the adjacent sector with the new frustum. This procedure continues until no portals are in view, at which point the polygons in each sector are rendered. Notice that this implements the Painter's Algorithm because those sectors furthest from the eye will be rendered first.

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. Like rendering, collision detection against the environment is localized only to those sectors that will have a potential collision, and the vast majority of sectors in the environment can be ignored.

Conclusion

The algorithms discussed in this thesis allow an application to allow a user to model and render an environment efficiently. The basic structure of an environment can

be created with CSG expressions, and then fed to the portal-sector algorithm to allow a real-time simulation to provide for both efficient rendering and collision detection.

By no means are the algorithms discussed in this paper sufficient for a professional simulation engine. Given more time, I would have liked to implement dynamic lighting, support for detail objects (tables, chairs, etc.), and a robust particle engine that would allow simulated effects for weather and explosions. In addition, support for a skybox that would allow designers to simulate exterior environments would allow broader design capabilities during environment creation.

The application that implemented the algorithms discussed in this paper, except for dynamic collision detection, was written in C++ using the MFC and OpenGL libraries. The application allows users to create large environments using CSG operations. It then can take the result of the CSG expression and create a portal-cell data structure for the environment. Finally, it renders the world efficiently by using that data structure. Environments created in the application can be saved to disk. The core architecture for the rendering engine is encapsulated in a DLL file that can be reused in other applications.

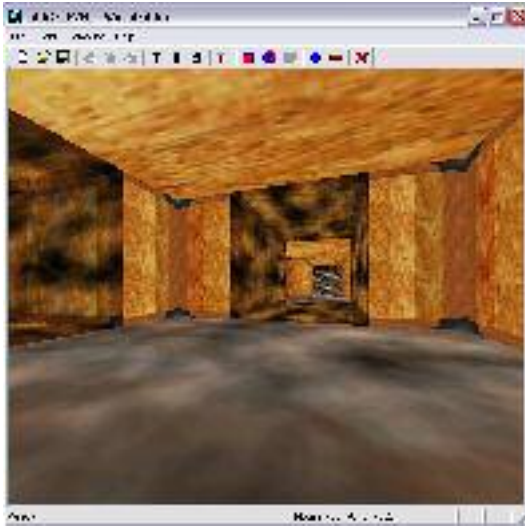


Figure 10.
A rendering of a BSP tree.

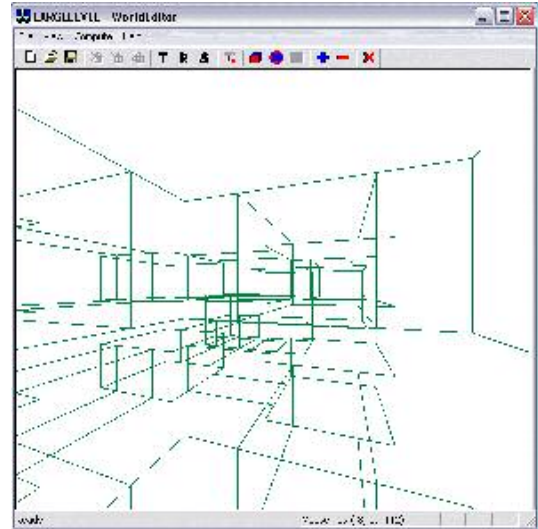


Figure 11.
Wireframe rendering showing large amount of unnecessary polygons being rendered.

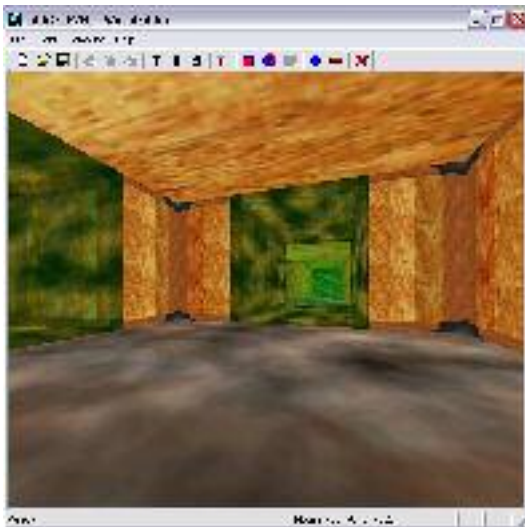


Figure 12.
Same rendering as above but using portal technique. Green Regions are portal polygons.

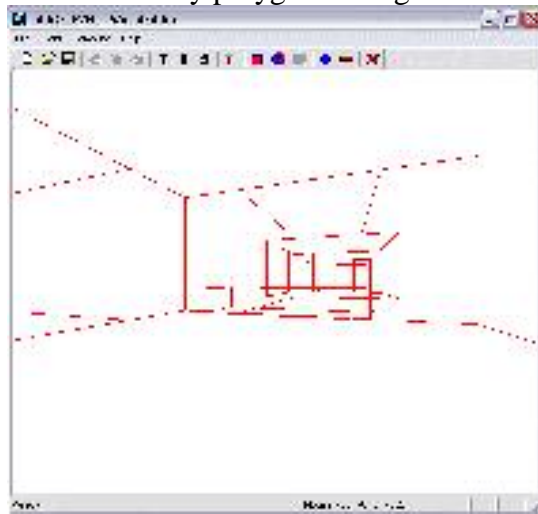


Figure 13.
Wireframe rendering showing the reduction in polygons rendered using portal technique.

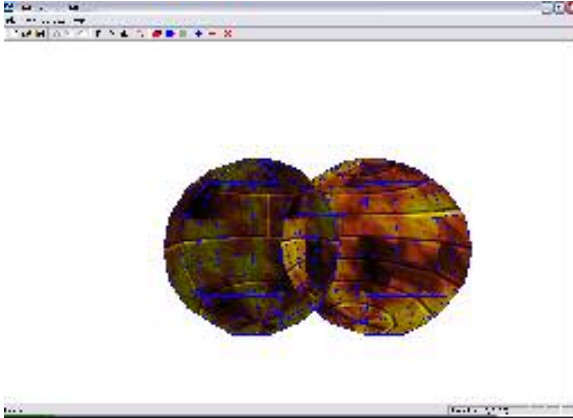


Figure 14.
Two spheres prior to CSG union operation.

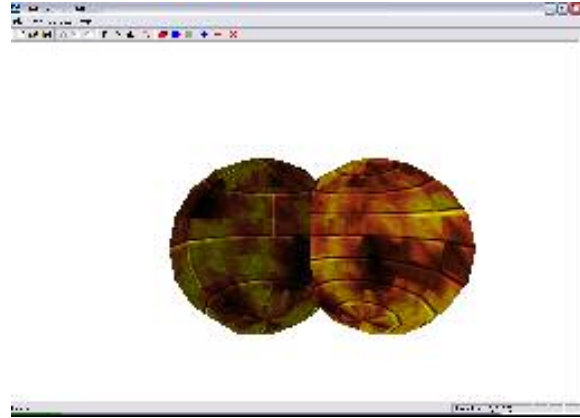


Figure 15.
Result of CSG union operation from figure 14.

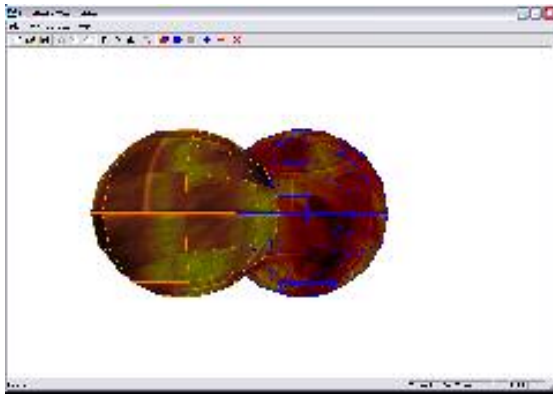


Figure 16.
Two spheres prior to CSG subtract operation.

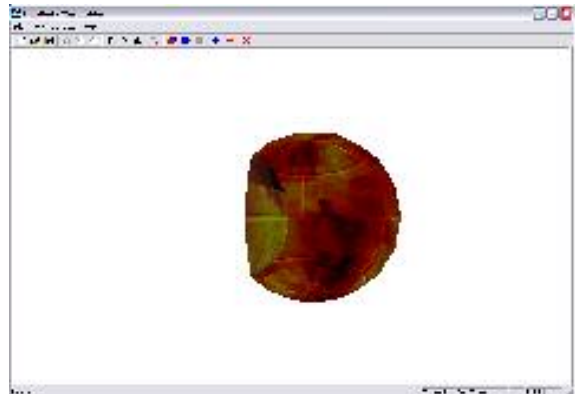


Figure 17.
Result of CSG subtract operation from Figure 16.

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, April 1995, <http://www.cs.virginia.edu/~luebke/publications/portals.html>
2. Berg M. *Trends and Developments in Computational Geometry*, Department of Computer Science, Utrecht University, May 1995.
3. S. F. Buchele. *Three-Dimensional Binary Space Partitioning Tree and Constructive Solid Geometry Tree Construction from Algebraic Boundary Representations*. Ph.D. Dissertation, Department of Computer Science, The University of Texas at Austin, 1999.
4. S. F. Buchele and A. C. Roles. *Binary Space Partition Tree and Constructive Solid Geometry Representations for Objects Bounded by Curved Surfaces*. Department of Mathematics and Computer Science, Southwestern University, 2000.
5. W. Thibault and B. Naylor. *Set Operations on Polyhedra Using Binary Space Partitioning Trees*, Computer Graphics, Vol. 21, No. 4, 1987
6. S. Ranta-Eskola. *Binary Space Partitioning Trees and Polygon Removal in Real Time 3D Rendering*. Information Technology Computing Science Department, Uppsala University, 2001.
7. Fuchs H, Kedem Z, Naylor B. *On Visible Surface Generation By A Priori Tree Structures*. Computer Graphics (Proceedings of SIGGRAPH '80), 14:124-133, 1980.
8. Teller SJ, *Visibility Computations in Densely Occluded Polyhedral Environments*. Ph.D. Thesis, Computer Science Department, University of California at Berkeley, 1992.
9. Royer, Dan. *CSG - Constructive Solid Geometry* Date Accessed: April 20, 2004, <http://www.cfxweb.net/~aggrav8d/tutorials/csg.html>
10. Baylis, Alan. *Portals Tutorial*. Al's OpenGL Game Development. January 30, 2002. Date Accessed: April 20, 2004, http://members.net-tech.com.au/alaneb/portals_tutorial.html
11. Whitaker, Nathan. *Extracting Connectivity Information From a BSP Tree*. Exaflop. Date Accessed: April 20, 2004, <http://www.exaflop.org/docs/naifgfx/naifebsp.html>
12. Naylor et. al. *Binary Space Partitioning Trees FAQ*. July 5, 1995. Date Accessed: April 20, 2004, <http://www.graphics.cornell.edu/bspfaq/>
13. Elmqvist, Niklas. *Introduction to Portal Rendering: Indoor Scene Rendering Made Easy*. March 21, 1999. Date Accessed: 4/20/2004, <http://www.medialab.chalmers.se/c2c/doc/portals.html>