Practical Training Report

Hidden Line Removal using BSP Tree and Polygon Clipping

submitted in partial fulfillment of the
requirements for the degree of

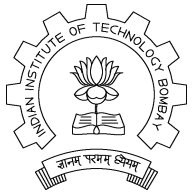
Bachelor of Technology
in
Aerospace Engineering

by

Anirudh Modi
93001015

under the guidance of

Dr. S. Gopalsamy
(NCST, Juhu)



Department of Aerospace Engineering
Indian Institute of Technology
Bombay
1996

Acknowledgment

4^{th} October, 1996

I would like to take this opportunity to express my gratitude to my guide, **Dr. S. Gopalsamy**, for his tolerance, and for the valuable time he took out of his busy schedule to guide me throughout this training. I would also like to thank Dinesh Shikhare and Deepraj Dixit for their invaluable support and suggestions. I also owe my gratitude to Prof. G. R. Shevare, Aerospace Engg. Dept., IIT-Bombay, who made this practical training possible.

Anirudh Modi

Contents

1 Binary Space Partitioning Trees **5**
- 1.1 Algorithm 5
- 1.2 Implementation 8
- 1.3 Efficiency 9
- 1.4 Scope for Improvement 9

2 Polygon Clipping **10**
- 2.1 Weiler-Atherton Algorithm 10
- 2.2 Implementation 12
- 2.3 Efficiency 12
- 2.4 Scope for Improvement 12

3 Hidden Line Removal using BSP Tree and Polygon Clipping **13**
- 3.1 Algorithm 13
- 3.2 Implementation 14
- 3.3 Efficiency 14
- 3.4 Scope for Improvement 15

4 Results **16**

List of Figures

4.1 Input - Mushroom wire frame ($N_i = 240$) 17
4.2 Output - Mushroom rendered after BSP sorting ($N_o = 481$) 17
4.3 Input - Goblet wire frame ($N_i = 470$) 18
4.4 Output - Goblet rendered after BSP sorting ($N_o = 1109$) 18
4.5 Input - Seashell wire frame ($N_i = 1680$) 19
4.6 Output - Seashell rendered after BSP sorting ($N_o = 3706$) 19
4.7 Input - Teapot wire frame ($N_i = 3751$) 20
4.8 Output - Teapot rendered after BSP sorting ($N_o = 7743$) 20
4.9 Example - Input to BSP routine ($N_i = 12$) 21
4.10 Example - Output after BSP sorting ($N_o = 12$) 21
4.11 Example - Input to Hidden Line Removal routine ($N_i = 10$) 22
4.12 Example - Output from Hidden Line Removal routine ($N_o = 7$) . . 22

Chapter 1

Binary Space Partitioning Trees

The binary space partitioning (BSP) tree algorithm was developed by Fuchs, Kedem, and Naylor [7], based on work of Schumacker [8]. The BSP tree algorithm is an efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary viewpoint. It trades off an initial time and space intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications in which the viewpoint changes but the objects do not.

1.1 Algorithm

The BSP tree algorithm [1] is based on the observation that a polygon will be scan-converted correctly (i.e., will not overlap incorrectly or be overlapped incorrectly by other polygons) if all polygons on the other side of it from the viewer are scan-converted first, followed by it, and then all polygons on the same side of it as the viewer. We need to ensure that this is so for each polygon.

The algorithm makes it easy to determine a correct order for scan conversion by building a binary tree of polygons, the **BSP** tree. The BSP tree's root is a polygon selected from those to be displayed; the algorithm works correctly no matter which is picked. The root polygon is used to partition the environment into two half-spaces. One half-space contains all remaining polygons in front of the root polygon, relative to its surface normal; the other contains all polygons behind the root polygon. Any polygon lying on both sides of the root polygon's plane is split by the plane, and its front and back pieces are assigned to the appropriate half-space. One polygon each from the root polygon's front and back half-spaces becomes its front and back children, and each child is recursively used to divide the remaining polygons in its half-space in the same fashion. The algorithm ter-

minates when each node contains only a single polygon. A C++ pseudo-code for tree-building phase is given below [5].

```
struct  BSP_tree
{
   plane     partition;
   list      polygons;
   BSP_tree  *front,
             *back;
};

void Build_BSP_Tree (BSP_tree *tree, list polygons)
{
   polygon   *root = polygons.Get_From_List ();
   tree->partition = root->Get_Plane ();
   tree->polygons.Add_To_List (root);
   list      front_list,
             back_list;
   polygon   *poly;
   while ((poly = polygons.Get_From_List ()) != 0)
   {
      int   result = tree->partition.Classify_Polygon (poly);
      switch (result)
      {
         case COINCIDENT:
            tree->polygons.Add_To_List (poly);
            break;
         case IN_BACK_OF:
            backlist.Add_To_List (poly);
            break;
         case IN_FRONT_OF:
            frontlist.Add_To_List (poly);
            break;
         case SPANNING:
            polygon   *front_piece, *back_piece;
            Split_Polygon (poly, tree->partition, front_piece, back_piece);
            backlist.Add_To_List (back_piece);
            frontlist.Add_To_List (front_piece);
            break;
      }
```

```
   }
   if ( ! front_list.Is_Empty_List ())
   {
      tree->front = new BSP_tree;
      Build_BSP_Tree (tree->front, front_list);
   }
   if ( ! back_list.Is_Empty_List ())
   {
      tree->back = new BSP_tree;
      Build_BSP_Tree (tree->back, back_list);
   }
}
```

Remarkably, the BSP tree can be traversed in a modified in-order tree walk to yield a correct priority-ordered polygon list for an arbitrary viewpoint. Consider the root polygon. It divides the remaining polygons into two sets, each of which lies entirely on one side of the root's plane. Thus, the algorithm needs only to guarantee that the sets are displayed in the correct relative order to ensure both that one set's polygons do not interfere with the other's and that the root polygon is displayed properly and in the correct order relative to the others. If the viewer is in the root polygon's front half-space, then the algorithm must first display all polygons in the root's rear half-space (those that could be obscured by the root), then the root, and finally all polygons in its front half-space (those that could obscure the root). Alternatively, if the viewer is in the root polygon's rear half-space, then the algorithm must display all polygons in the root's front half-space, then the root, and finally all polygons in its rear half-space. If the polygon is seen on edge, either display order suffices. Back-face culling may be accomplished by not displaying a polygon if the eye is in its rear half-space. Each of the root's children is recursively processed by this algorithm.

```
void Draw_BSP_Tree (BSP_tree *tree, point eye)
{
   real   result = tree->partition.Classify_Point (eye);
   if (result > 0)
   {
      Draw_BSP_Tree (tree->back, eye);
      tree->polygons.Draw_Polygon_List ();
      Draw_BSP_Tree (tree->front, eye);
   }
   else if (result < 0)
   {
```

```
      Draw_BSP_Tree (tree->front, eye);
      tree->polygons.Draw_Polygon_List ();
      Draw_BSP_Tree (tree->back, eye);
   }
   else // result is 0
   {
      // the eye point is on the partition plane...
      Draw_BSP_Tree (tree->front, eye);
      Draw_BSP_Tree (tree->back, eye);
   }
}
```

Like the depth-sort algorithm, the BSP tree algorithm performs intersection and sorting entirely at object precision, and relies on the image-precision overwrite capabilities of a raster device. Unlike depth sort, it performs all polygon splitting during a preprocessing step that must be repeated only when the environment changes. Note that more polygon splitting may occur than in the depth-sort algorithm.

1.2 Implementation

The BSP tree construction and rendering/traversing algorithm is implemented as a part of the practical training. The core code (written in *ANSI C*) has been taken from *Graphics Gems* [2] and modified to suit the needs. Certain additions have been made to accept the *3D* objects in standard *OFF* format files as the input and to display the rendered output using a simple viewer using the standard *X Window* libraries.

The algorithm implemented here takes care of only objects consisting of convex polygons, as the polygon splitting algorithm is implemented only for convex polygons. This is because convex polygons are generally easier to deal with in BSP tree construction than concave ones, because splitting them with a plane always results in exactly two convex pieces. Furthermore, the algorithm for splitting convex polygons is straightforward and robust. Splitting of concave polygons, especially self intersecting ones, is a significant problem in its own right.

The choice of the the dividing polygon is made by testing the first N polygons in the list (denoted by the global variable MAX_CANDIDATES in the program written) and checking for the number of intersections of each polygon with the other polygons in the list. The polygon giving rise to the minimum number of intersections is chosen as the dividing polygon. This is an $O(N^2)$ process. Ideally, N should be equal to the number of polygons in the list (although even this

does not guarantee the minimum number of intersections in totality, as this will also affect the decision for the choice of the next dividing polygon in the recursive algorithm), but practically it is taken anywhere from $5 - 100$ depending on the tradeoff between the number of resulting output polygons due to the artificial intersections (the intersections resulting from division of the polygons in the list by the dividing polygon) and time. Large N is recommended where the time taken for traversing of the tree (for displaying) for various viewpoints is more important that the time required for the construction of the BSP tree, as in the case of map constructors in DOOM like games where the BSP tree is the input to the renderer.

1.3 Efficiency

For hidden surface removal and ray tracing acceleration using BSP tree, the upper bound for both space and time complexity is $O(N^2)$ for N polygons. The expected case is $O(N)$ for most models.

1.4 Scope for Improvement

- Artificial intersections which are seen in the final output can be prevented from being displayed by flagging them in the intersection routine and using this information while rendering.
- Support for objects containing concave polygons can also be included by extending the plane-polygon intersection routine to concave polygons.
- Also, the algorithm can be implemented in a non-recursive way by using a stack of pointers to BSP_tree. This can be faster on some processors.

Chapter 2

Polygon Clipping

Clipping, the process of extracting a portion of a data base, is fundamental to several aspects of computer graphics. In addition to its more typical use in selecting only the specific information required to display a particular scene or view from a larger environment, it is also useful in hidden line, hidden surface, shadow, and texture algorithms. In the context of many applications, e.g., hidden surface removal, the ability to clip concave polygons is required. Discussed below is a very powerful polygon clipping algorithm due to Kevin Weiler and Peter Atherton [6].

2.1 Weiler-Atherton Algorithm

The Weiler-Atherton algorithm is capable of clipping a concave polygon with interior holes to the boundaries of another concave polygon, also with interior holes. The polygon to be clipped is called the subject polygon (SP) and the clipping region is called the clip polygon (CP). The new boundaries created by clipping the SP against the CP are identical to portions of the CP. No new edges are created. Hence, the number of resulting polygons is minimized.

The algorithm describes both the SP and the CP by a circular list of vertices. The exterior boundaries of the polygons are described clockwise, and the interior boundaries or holes are described counter-clockwise. When traversing the vertex list, this convention ensures that the inside of the polygon is always to the right. The boundaries of the SP and the CP may or may not intersect. If they intersect, the intersections occur in pairs. One of the intersections occurs when the SP edge enters the inside of the CP and one when it leaves. Fundamentally, the algorithm starts at an entering intersection and follows the exterior boundary of the SP clockwise until an intersection with a CP is found. At the intersection a right turn is made, and the exterior of the CP is followed clockwise until an intersection with the SP is found. Again, at the intersection, a right turn is made, with the SP

now being followed. The process is continued until the starting point is reached. Interior boundaries of the SP are followed counter-clockwise.

A more formal statement of the algorithm is [3]

- **Determine the intersections of the subject and clip polygons** - Add each intersection to the SP and CP vertex lists. Tag each intersection vertex and establish a bidirectional link between the SP and CP lists for each intersection vertex.
- **Process nonintersecting polygon borders** - Establish two holding lists: one for boundaries which lie inside the CP and one for boundaries which lie outside. Ignore CP boundaries which are outside the SP. CP boundaries inside the SP form holes in the SP. Consequently. a copy of the CP boundary goes on both the inside and the outside holding list. Place the boundaries on the appropriate holding list.
- **Create two intersection vertex lists** - One, the entering list, contains only the intersections for the SP edge entering the inside of the CP. The other, the leaving list, contains only the intersections for the SP edge leaving the inside of the CP. The intersection type will alternate along the boundary. Thus, only one determination is required for each pair of intersections.
- **Perform the actual clipping** -
  Polygons inside the CP are found using the following procedure.
  - Remove an intersection vertex from the entering list. If the list is empty, the process is complete.
  - Follow the SP vertex list until an intersection is found. Copy the SP list upto this point to the inside holding list.
  - Using the link, jump to the CP vertex list.
  - Follow the CP vertex list until an intersection is found. Copy the CP vertex list upto this point to the inside holding list.
  - Jump back to the SP vertex list.
  - Repeat until the starting point is again reached. At this point, the new inside polygon has been closed.

  Polygons outside the CP are found using the same procedure, except that the initial intersection vertex is obtained from the leaving list and the CP vertex list is followed in the reverse direction. The polygon lists are copied to the outside holding list.

  A suitably detailed description of the algorithm can be found in [3] and [6].

2.2 Implementation

The Weiler-Atherton algorithm implemented here works for clipping of one concave polygon against another and handles all special cases (i.e. coincident edges, coincident vertices, vertex of one polygon lying on the edge of the other, etc.). The implementation however, does not take care of self-intersecting polygons and polygons with holes.

2.3 Efficiency

The complexity of the intersection calculations is always $O(MN)$ where M and N are the number of vertices of the CP and SP respectively. The complexity of the polygon clipping problem for a non-special case is in the worst case $O(MN)$. For disjoint CP and SP, the intersection algorithm checks for bounding boxes and hence is more efficient. For special cases, the complexity may be worse due to the overheads involved in their detection. [4]

2.4 Scope for Improvement

If appropriate space partitioning techniques are used, the complexity of the intersection calculations can be reduced from $O(MN)$ to $O(M+N)$. Since not sufficient amount of literature for classifying and handling the special cases was available at the time of writing of the code, a trial and error method was adopted to arrive at an algorithm to classify all the special cases known. Since the algorithm is not rigorously tested, there might just be a very very special case for which it may fail.

Chapter 3

Hidden Line Removal using BSP Tree and Polygon Clipping

3.1 Algorithm

Here, the BSP Tree routine and Polygon Clipping routine are used in succession to perform hidden line removal. A C++ pseudo-code illustrating the algorithm is given below.

```
void HiddenLineRemoval ()
{
  OFFObject *obj;
  Point3d viewpoint;
  Poly2d subject,clip;
  Poly2d *insidelist,*outsidelist;
  Poly2d *subjectlist,*new_subjectlist;
  int i,j,k,l;

  ReadOFFObject (obj);
  Read (viewpioint);

  Sort_using_BSP (obj,viewpoint);

  Transform2d (obj,viewpoint);

  for (i = obj->nf-1; i >= 0; i--)
     {
      subject = obj->faces[i];
```

```
      subjectlist[0] = subject;
      subjectlist->n = 1;

      for (j = i+1; j <= obj->nf; j++)
         {
          clip = obj->faces[j];

          new_subjectlist->n = 0;
          for (k = 0; k < subjectlist->n; k++)
             {
              Clip2polygons(clip,subjectlist[k],insidelist,outsidelist);
              new_subjectlist += outsidelist;
              new_subjectlist->n += outsidelist->n;
              // Append 'outsidelist' to 'new_subjectlist'
             }

          subjectlist = new_subjectlist;
         }

       Output (subjectlist);
              // Print 'subjectlist' as output
     }
}
```

The input here are the set of convex polygons in *3D* space in *OFF* format and the view point, and the output is a set of non-overlapping polygons in *2D* space which when drawn in any order, give the desired hidden line removal effect with respect to the given view point.

3.2 Implementation

The above algorithm was formulated and implemented during the duration of the practical training. However, the BSP routine could not be incorporated into the code due to lack of time. Thus, the implemented code requires the output from the BSP routine as its input.

3.3 Efficiency

The algorithm as described above has complexity of $O(N^2)$, N being the number of polygons in the input. The efficiency greatly depends on the efficiency of the

clipping algorithm which itself has complexity of $O(MN)$ as described previously. But since a significant number of polygons in a typical input are disjoint (i.e. not all polygons in the list necessarily intersect each other), the polygon clipping routine on an average has a complexity of much less than $O(N^2)$ for sufficiently large N. Thus, the typical time complexity of the algorithm would be $O(N^3)$.

3.4 Scope for Improvement

There is ample scope for improvement in the algorithm mentioned above. It is almost a brute force implementation using the clipping routine to obtain hidden line removal and there exist more efficient ways (of possibly $O(NlogN)$ complexity) to obtain the same results (although implementation may be more complex). Also, dynamic memory allocation (DMA) can be used instead of the present fixed memory allocation to efficiently utilise the system resources.

Chapter 4

Results

Some sample results of the discussed algorithms from the various routines implemented in *ANSI C* are shown here.

In all the BSP tree construction calculations, MAX_CANDIDATES = 10. Also, N_i denotes the number of polygons in the input and N_o denotes the number of polygons in the output. On a *486 DX2-66* with *8 MB RAM* running *Linux v. 1.3.47*, the time taken was

- Mushroom ($N_i = 240$) $\rightarrow$ 2 *sec.* (Figure 4.1)
- Goblet ($N_i = 470$) $\rightarrow$ 5 *sec.* (Figure 4.3)
- Seashell ($N_i = 1680$) $\rightarrow$ 8 *sec.* (Figure 4.5)
- Teapot ($N_i = 3751$) $\rightarrow$ 42 *sec.* (Figure 4.7)

The artificial intersections due to BSP Tree construction are clearly seen in the output of the first few examples (Figures 4.2, 4.4, 4.6 and 4.8).

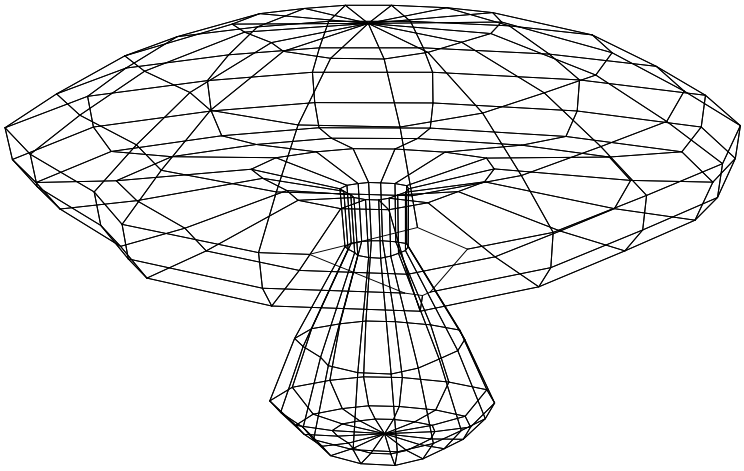


Figure 4.1: Input - Mushroom wire frame ($N_i = 240$)

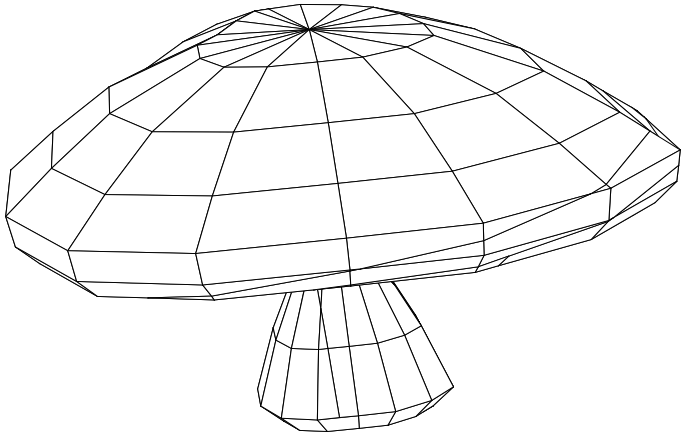


Figure 4.2: Output - Mushroom rendered after BSP sorting ($N_o = 481$)

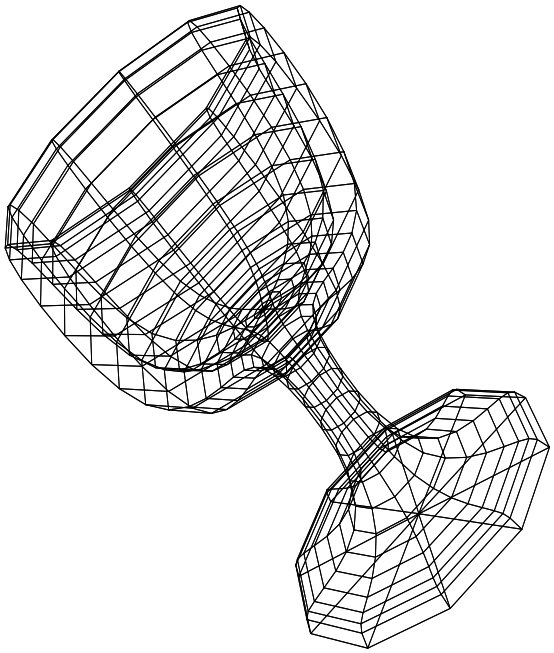


Figure 4.3: Input - Goblet wire frame ($N_i = 470$)

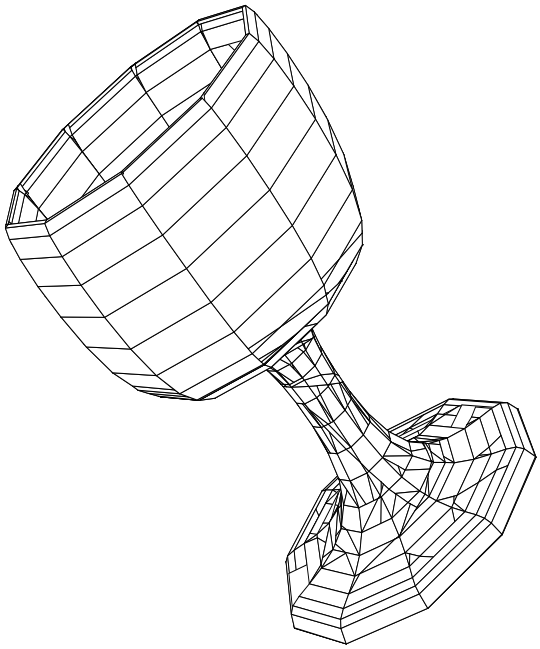


Figure 4.4: Output - Goblet rendered after BSP sorting ($N_o = 1109$)

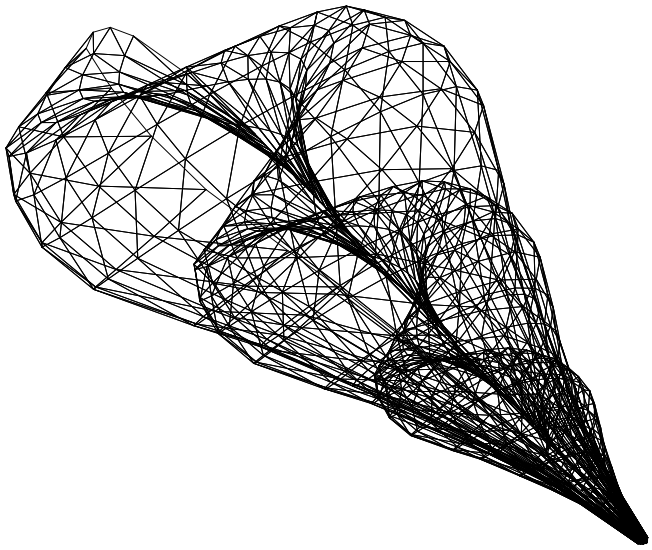


Figure 4.5: Input - Seashell wire frame ($N_i = 1680$)

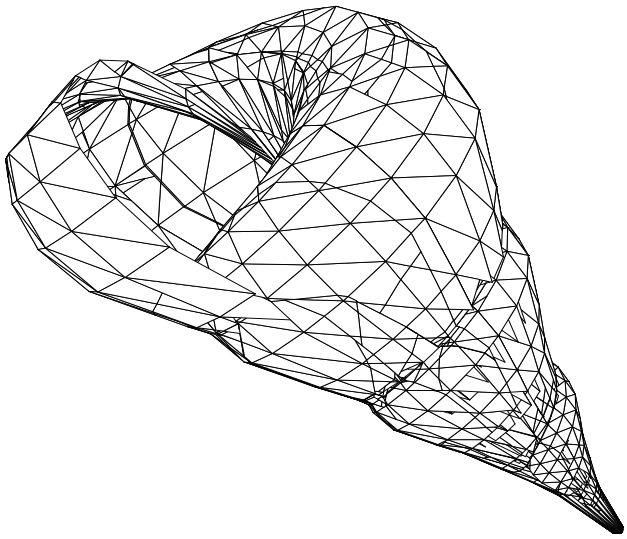


Figure 4.6: Output - Seashell rendered after BSP sorting ($N_o = 3706$)

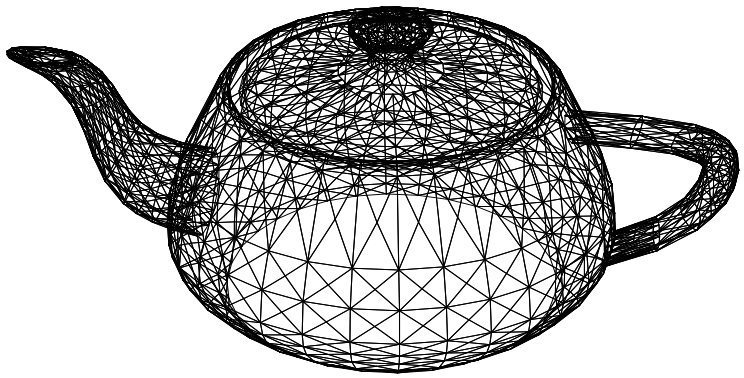


Figure 4.7: Input - Teapot wire frame ($N_i = 3751$)

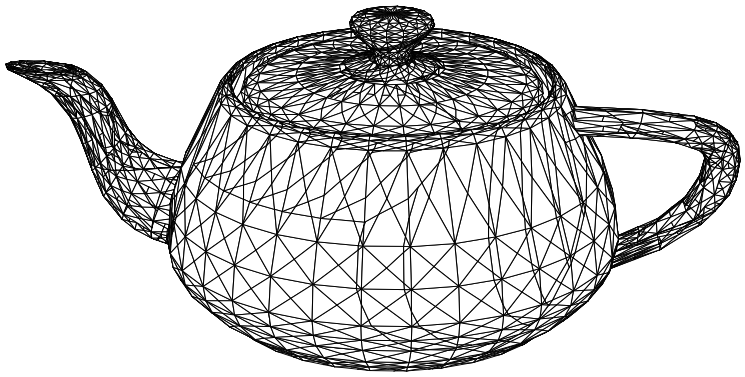


Figure 4.8: Output - Teapot rendered after BSP sorting ($N_o = 7743$)

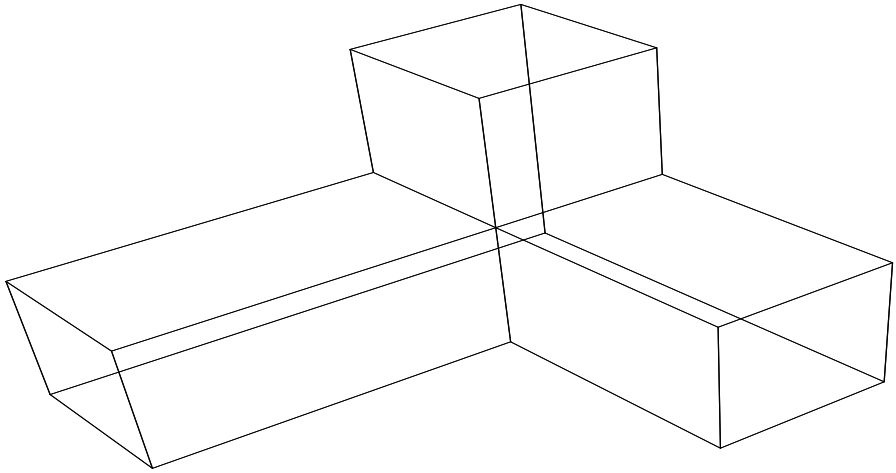


Figure 4.9: Example - Input to BSP routine ($N_i = 12$)

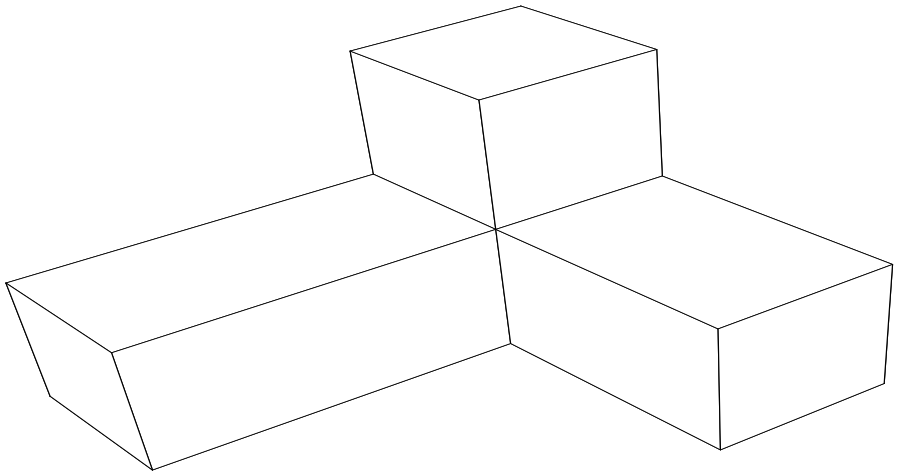


Figure 4.10: Example - Output after BSP sorting ($N_o = 12$)

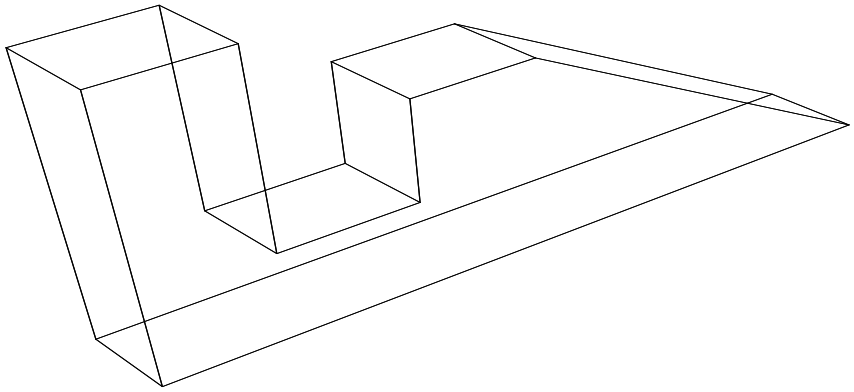


Figure 4.11: Example - Input to Hidden Line Removal routine ($N_i = 10$)

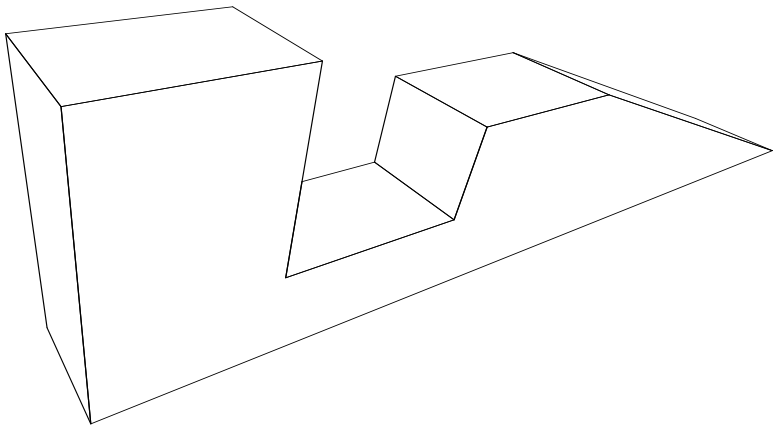


Figure 4.12: Example - Output from Hidden Line Removal routine ($N_o = 7$)

Bibliography

[1] J.D. Foley, A. Van Dam, S.K. Feiner, J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Second edition, 1990, ISBN 0-201-12110-7.

[2] Alan W. Paeth (ed.). *Graphics Gems V*. Academic Press 1995, ISBN 0-12-543455-3.

[3] David F. Rogers. *Procedural Elements for Computer Graphics*. McGraw Hill 1985, ISBN 0-07-053534-5.

[4] Schutte, Klamer (*klamer@ph.tn.tudelft.nl*). *An Edge Labeling Approach to Concave Polygon Clipping*. ACM Transactions on Graphics, July 1995 (*ftp://ftp.ph.tn.tudelft.nl/pub/klamer/clippoly.tar.gz*).

[5] Wade, Bretton (*bwade@qualia.com*). *Frequently Asked Questions (FAQ) about Binary Space Partitioning (BSP) Trees*. Qualia, Incorporated (*http://www.qualia.com/bspfaq/*) .

[6] Weiler, Kevin and Atherton, Peter. *Hidden Surface Removal Using Polygon Area Sorting*. Computer Graphics, Vol. 11, pp. 214-222, 1977 (Proc. SIGGRAPH '77).

[7] Fuchs, H., Kedem, Z., and Naylor, B.. *Visible Surface Generation by A-Priori Tree Structures*. Conf. Proc. of SIGGRAPH '80, 14(3), 124–133, Jul 1980.

[8] Schumacker, Brand, Gilliland and Sharp. *Study for Applying Computer-Generated Images to Visual Simulation*. AFHRL-TR-69-14, US AF Human Resources Lab, 1969.

[9] Sutherland, Sproull and Schumacker. *A Characterization of Ten Hidden Surface Algorithms*. ACM Computing Surveys, Vol. 6, No. 1, pp 1-55.