# A Shaft Culling Tool

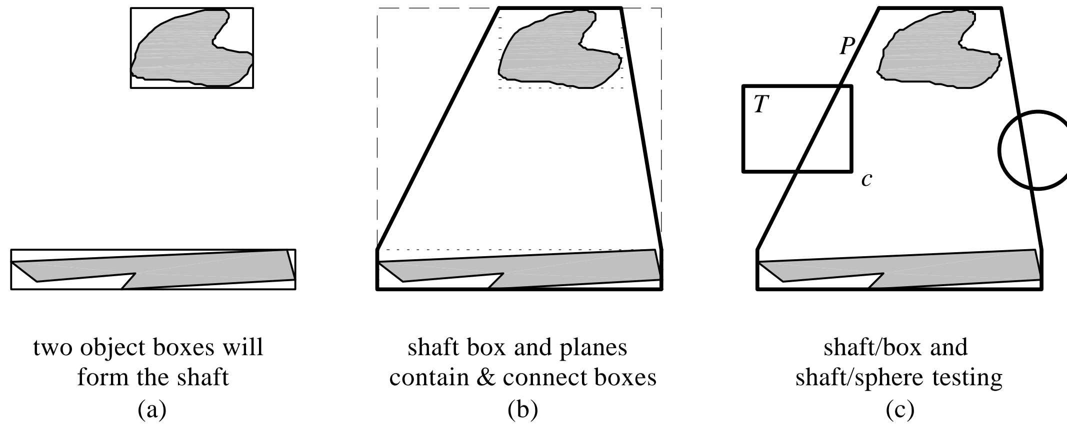
Eric Haines
Autodesk, Inc.
Ithaca, New York
erich@acm.org

**Abstract:** Shaft culling is a means to accelerate the testing of visibility between two objects. This paper briefly describes an algorithm for shaft culling and various implementation options. The entire code and test harness for the algorithm is available online.

## Introduction

Shaft culling is the process of examining two objects and determining what other objects are potentially in between them. The algorithm developed by Haines & Wallace [1] was originally used to efficiently find the visibility between two objects for radiosity calculations. However, the method is a purely geometrical operation that can be used in other rendering techniques and for collision detection. The space swept out by a moving object (e.g. a catapult boulder) between one frame and the next forms a shaft, which can then be tested against the rest of the scene for collision. Another use is for quickly identifying all the objects between a point light source and some set of receivers. This can be helpful in more rapidly generating drop shadows by identifying only those objects that could project onto the receivers.

Putting an axis-aligned bounding box around both objects and connecting these two objects' boxes by a set of planes forms a shaft (Figure 1). Other bounding volumes can then be tested against this shaft to see if they are fully outside. If not, additional testing can be done to see if they are fully inside. Testing a box against a shaft is a fast operation, as only one corner of the box needs to be compared to each particular plane in the shaft. For example, only the lower right corner $c$ of the test box $T$ in Figure 1(c) needs to be tested against the shaft plane $P$ to determine if the box is outside of the plane. The plane's normal determines in advance which box corner to test. The signs of the normal's coordinates determine the octant that corresponds to which box corner to test.




two object boxes will form the shaft (a)

shaft box and planes contain & connect boxes (b)

shaft/box and shaft/sphere testing (c)

Figure 1.

Code in C for this tool, along with a test harness that gathers statistics and shows the shafts formed, is available online at http://www.acm.org/jgt/. The original algorithm tested only boxes against the shaft; the code here includes support for testing spheres against the shaft. The sphere test is imperfect, but conservative: on occasion it will miscategorize a sphere as overlapping that is actually fully outside the shaft. For most applications of this tool this will lead to a little inefficiency, but not incorrect results.

## Shaft Formation

The most involved part of shaft culling code is in forming the shaft itself. Efficiently generating the planes between the two object boxes is not an obvious process. The approach taken here can be thought of as follows. Each of the 6 faces of the shaft box is touched by one or both of the object boxes. Say we paint red each face touched by only the first object's box, and paint blue each face touched only by the second's. A shaft plane is formed for any edge whose two neighboring faces differ in color. The corresponding edges on the two object boxes are used to generate the plane. There will be from 0 to 8 shaft planes generated. As an example, in Figure 1(b) say the top object box is red and the bottom blue. The shaft box's top is then painted red and the other 3 faces are made blue. This leads to the formation of two shaft planes corresponding to the two upper corners, where the different colors meet.

The code gives two forms of this algorithm, one that finds these edges during run-time and another that uses a precomputed table which stores the relevant edges. The precomputed table has 64 entries, formed by six bits. Each bit represents the color of the corresponding shaft face, e.g. 0 for red, 1 for blue. Interestingly, on a Pentium II the precomputed table code forms a shaft in only 3% less time than the run-time code.

The run-time code has a further optimization. If *both* object boxes touch a shaft face, color that face purple. This corresponds to the object box faces laying in the same plane, meaning that no additional shaft planes are needed to join these faces. Again, only edges with one red and one blue neighbor will form a shaft; purple faces are ignored. This optimization avoids generating unnecessary shaft planes. It could be added to the table

approach, but leads to a 729 entry table (3 to the $6^{th}$). Pseudocode for run-time shaft formation is given in Listing 1.

```
For all faces f = (lo/hi.x/y/z) of the two object boxes
  /* color the face purple if box extents are the same */
  If the two box coordinates are equal:
    match[f] = true
  /* else color the face red or blue */
  Else:
    match[f] = false
    Note which object box coordinate is further out
    by saving color[f] = object box 1 or 2.

/* loop through and check all faces against each
 * other to determine if a plane should join them */
For face f1 = lo.x/y/z through hi.x/y (0 through 4)
  If match[f1] is false:
  /* else face is purple, and no edge forms a shaft */
    For face f2 = f1+1 through hi.z (through 5)
      If match[f2] is false:
      /* else face is purple */
        /* check if faces share an edge; opposite face
         * pairs such as lo.y and hi.y do not */
        If f1+3 is not equal to f2:
          /* test if faces are different colors */
          If color[f1] is not equal to color[f2]:
            Create and store a plane joining the two
            object box faces f1 & f2 along their common
            edge.
```

Listing 1 – shaft formation pseudocode

Once the shaft is formed, another optimization can be done. Each shaft plane chops off a certain volume of the shaft box. Logically, the shaft plane that cuts off the largest volume should be tested first, since it is most likely to cull out an arbitrary test box and so return more quickly. Code is included to sort the shaft's planes by the amount of volume they cull. In practical benchmarks, this optimization was found to save an average of from 0.1 to 1.1 plane tests per non-trivial box/shaft comparison (i.e. ignoring cases where the test box was fully outside the shaft box). This savings is relatively minor, but if a single shaft is going to be tested against many boxes then this additional sorting step may pay off. More elaborate sorts could also be done; the code as given does not take into account the amount of overlap of volumes trimmed by the planes. For example, shaft planes for diagonally opposite edges could be good to pair up, as the trimmed volumes for these will never overlap.

A number of other variations on the code are outlined in the code comments, with tradeoffs in speed, memory, and flexibility discussed. Most of these code variations can be switched on or off and so can be tried by the user of this tool. Thanks go to Martin Blais for independent proofreading and linting of this code.

## References

1. Haines, Eric A., and John R. Wallace, "Shaft Culling for Efficient Ray-Traced Radiosity," *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering),* Springer-Verlag, New York, 1994, p.122-138. Also in *SIGGRAPH '91 Frontiers in Rendering course notes*. Available online at http://www.acm.org/tog/editors/erich/