

Fast Projected Area Computation for Three-Dimensional Bounding Boxes

Dieter Schmalstieg and Robert F. Tobler
Vienna University of Technology

Abstract. The area covered by a three-dimensional bounding box after projection onto the screen is relevant for view-dependent algorithms in real-time and photorealistic rendering. We describe a fast method to compute the accurate two-dimensional area of a three-dimensional oriented bounding box, and show how it can be computed equally fast or faster than its approximation with a two-dimensional bounding box enclosing the projected three-dimensional bounding box.

1. Introduction

Computer graphics algorithms using heuristics, like level of detail (LOD) selection algorithms, make it sometimes necessary to estimate the area an object covers on the screen after perspective projection [Funkhouser, Sequin 93]. Doing this exactly would require first drawing the object and then counting the covered pixels, which is quite infeasible for real-time applications. Instead, oftentimes a bounding box (bbox) is used as a rough estimate: The bbox of the object is projected to the screen, and its size is taken.

Another application area is view-dependent hierarchical radiosity, where a fast method for calculating the projection area can be used to estimate the importance of high level patches, obviating the need to descend the hierarchy in places of little importance.

The reason for favoring bounding boxes over bounding spheres is that they provide a potentially tighter fit (and hence a better approximation) for the object while offering roughly the same geometric complexity as spheres. How-

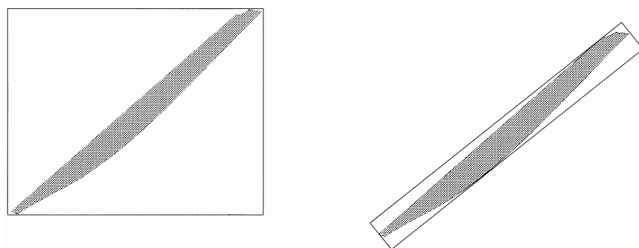


Figure 1. Axis-aligned bounding boxes (left) are often inferior to oriented bounding boxes (right).

ever, usually axis-aligned bboxes are used, which can also be a poor fit for the enclosed object. In contrast, an oriented bounding box (OBB) requires an additional transformation to be applied, but allows a comparatively tight fit (Figure 1). The speed and applicability of OBBs in other areas has been shown by Gottschalk et al. [Gottschalk et al. 96]. For the construction of an OBB, refer to [Wu 92]. To estimate the two-dimensional area of a three-dimensional object when projected to the screen, one can find the perspective projection of the corners of an axis-aligned bbox, and then use the area of the rectangle (two-dimensional bbox) enclosing the three-dimensional bbox to estimate the area of the object on the screen. This procedure entails two nested approximations which are not necessarily a tight fit, and the error can be large.

Instead, we directly project an OBB and compute the area of the enclosing two dimensional polygon. This procedure yields significantly better approximations. Moreover, we will show that the procedure can be coded to require fewer operations than the nested bbox approach.

2. Algorithm Overview

In this section, we will show how a simple viewpoint classification leads to an approach driven by a lookup table, followed by an area computation based on a contour integral. Both steps can be coded with few operations and are computationally inexpensive. When a three-dimensional box is projected to the screen either one, two, or three adjacent faces are visible, depending on the viewpoint (Figure 2):

- **Case 1:** one face visible, two-dimensional hull polygon consists of four vertices
- **Case 2:** two faces visible, two-dimensional hull polygon consists of six vertices

Schmalstieg and Tobler: Fast Projected Area Computation for 3D Bounding Boxes

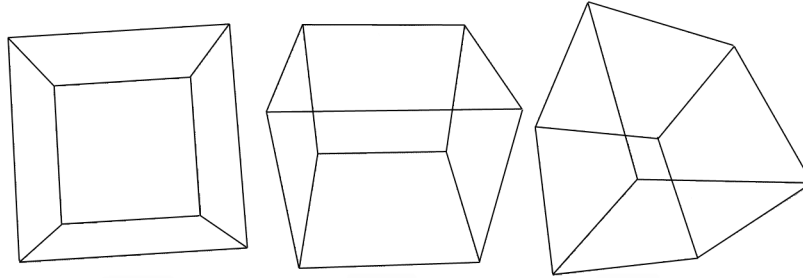


Figure 2. One, two, or three faces of a box may be visible.

- **Case 3:** three faces visible, two-dimensional hull polygon consists of six vertices

Whether a particular placement of the viewpoint relative to the bbox yields Case 1, 2, or 3, can be determined by examining the position of the viewpoint with respect to the six planes defined by the six faces of the bbox. These six planes subdivide Euclidean space into $3^3 = 27$ regions. The case where the viewpoint is inside the box does not allow meaningful area computation, so 26 valid cases remain.

By classifying the viewpoint as left or right of each of the six planes, we obtain $2^6 = 64$ theoretical cases, of which 26 are valid. For each of these cases we describe the hull polygon as an ordered set of vertex indices which can be precomputed and stored in a two-dimensional lookup table, called the hull vertex table.

An efficient implementation of the classification is to transform the viewpoint in the local coordinate system of the OBB, where each of the planes is parallel to one of the major planes, and the classification can be made by comparing one scalar value.

After the classification, the area of the hull polygon must be computed from the bbox vertices given in the hull vertex table. Our sample implementation uses a fast contour integral [Foley et al. 90].

3. Implementation

For an efficient implementation, the central data structure is the hull vertex table. It stores the ordered vertices that form the outline of the hull polygon after projection to two dimensions, as well as the number of vertices in the outline (four or six, with zero indicating an invalid case). The table is indexed with a 6-bit code according to Table 1.

Bit	5	4	3	2	1	0
Code	back	front	top	bottom	right	left

Table 1. Bit code used to index into the hull vertex table.

By precomputing this table, many computational steps can be saved when a bounding box area is computed at runtime. The hull vertex table used in the sample implementation is shown in Table 2.

Using this hull vertex table (`hullvertex`), the following C function `calculateBoxArea` computes the projected area of an OBB from the viewpoint given in parameter `eye` and the bounding box `bbox` given as an array of eight vertices (both given in local bbox coordinates). We assume an auxiliary function `projectToScreen`, which performs perspective projection of an OBB vertex to screen space.

```
float calculateBoxArea(Vector3D eye, Vector3D bbox[8])
{
    Vector2D dst[8]; float sum; int pos, num, i;
    int pos = ((eye.x < bbox[0].x)      ) // 1 = left   | compute 6-bit
              + ((eye.x > bbox[7].x) << 1) // 2 = right  |   code to
              + ((eye.y < bbox[0].y) << 2) // 4 = bottom |   classify eye
              + ((eye.y > bbox[7].y) << 3) // 8 = top   |with respect to
              + ((eye.z < bbox[0].z) << 4) // 16 = front| the 6 defining
              + ((eye.z > bbox[7].z) << 5); // 32 = back |   planes
    if (!num = hullvertex[pos][6]) return -1.0; //look up number of vertices
                                              //return -1 if inside
    for(i=0; i<num; i++) dst[i]:= projectToScreen(bbox[hullvertex[pos][i]]);
    sum = (dst[num-1].x - dst[0].x) * (dst[num-1].y + dst[0].y);
    for (i=0; i<num-1; i++)
        sum += (dst[i].x - dst[i+1].x) * (dst[i].y + dst[i+1].y);
    return sum * 0.5; //return corrected value
}
```

4. Discussion

The proposed implementation gives superior results to a simple “two-dimensional bbox of three-dimensional bbox” implementation. However, although it yields better accuracy, it can be implemented to use slightly fewer operations than the simple two-dimensional bbox variant. Our algorithm is composed of the following steps:

1. Transformation of the viewpoint into local bbox coordinates: Note that this step is not included in the sample implementation. Given the transformation matrix from world coordinates to local bbox coordinates, this is a simple affine transformation a three-dimensional

Schmalstieg and Tobler: Fast Projected Area Computation for 3D Bounding Boxes

Case	Num	Vertex Indices						Description
0	0							inside
1	4	0	4	7	3			left
2	4	1	2	6	5			right
3	0							-
4	4	0	1	5	4			bottom
5	6	0	1	2	6	5	4	bottom, right
6	6	0	1	2	6	5	4	bottom, right
7	0							-
8	4	2	3	7	6			top
9	6	4	7	6	2	3	0	top, left
10	6	2	3	7	6	5	1	top, right
11	0							-
12	0							-
13	0							-
14	0							-
15	0							-
16	4	0	3	2	1			front
17	6	0	4	7	3	2	1	front, left
18	6	0	3	2	6	5	1	front, right
19	0							-
20	6	0	3	2	1	5	4	front, bottom
21	6	1	5	4	7	3	2	front, bottom, left
22	6	0	3	2	6	5	4	front, bottom, right
23	0							-
24	6	0	3	7	6	2	1	front, top
25	6	0	4	7	6	2	1	front, top, left
26	6	0	3	7	6	5	1	front, top, right
27	0							-
28	0							-
29	0							-
30	0							-
31	0							-
32	4	4	5	6	7			back
33	6	4	5	6	7	3	0	back, left
34	6	1	2	6	7	4	5	back, right
35	0							-
36	6	0	1	5	6	7	4	back, bottom
37	6	0	1	5	6	7	3	back, bottom, left
38	6	0	1	2	6	7	4	back, bottom, right
39	0							-
40	6	2	3	7	4	5	6	back, top
41	6	0	4	5	6	2	3	back, top, left
42	6	1	2	3	7	4	5	back, top, right
≥ 43	0							-

Table 2. The hull vertex table stores precomputed information about the projected bbox.

vector is multiplied with a 3×4 matrix, using 12 multiplications and nine additions.

2. Computation of the index into the hull vertex table: To perform this step, the viewpoint's coordinates are compared to the defining planes of the bounding box. As there are six planes, this step uses at most six comparisons (a minimum of three comparisons is necessary if a cascading conditional is used for the implementation).
3. Perspective projection of the hull vertices: This step has variable costs depending on whether the hull consists of four or six vertices. The three-dimensional vertices that form the hull polygon must be projected into screen space. This is a perspective projection, but the fact that we are only interested in the x and y components (for area computation) allows a few optimizations. The x and y components are transformed using three multiply and two add operations per component. However, a perspective projection requires normalization after the matrix multiplication to yield homogeneous coordinates. A normalization factor must be computed, which takes one perspective division and one add operation. The x and y components are then normalized, taking two multiply operations. This analysis yields a total of 18 operations for an optimized perspective projection.
4. Area computation using a contour integral: Each signed area segment associated with one edge requires one add, one subtract, and one multiply operation, plus one add operation for the running score, except for the first edge. The result must be divided by two (one multiply operation). The total number of operations again depends on whether there are four or six vertices (and edges).

The total number of operations is 159 for Case 2 and 3 (six vertices), and 115 for case 1 (four vertices). The number of operations required to compute a simple two-dimensional box area is 163. It can be computed as follows: Projection of eight vertices (18×8 operations), computation of the two-dimensional bbox of the projected vertices using min-max tests (2×8 comparisons), two-dimensional box computation (two subtract, one add, one multiply operation).

As the number of operations required to compute the exact projected area of a three-dimensional bbox is of the same order or even less expensive than the simple approach using a two-dimensional bbox, it is recommended to use this procedure for real-time bounding box area computation.

Acknowledgments. Special thanks to Erik Pojar for his help with the sample implementation. This work was sponsored by the Austrian Science Foundation (FWF) under contract no. P-11392-MAT.

Schmalstieg and Tobler: Fast Projected Area Computation for 3D Bounding Boxes

References

- [Foley et al. 90] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics—Principles and Practice*. 2nd edition, Reading, MA: Addison Wesley, 1990.
- [Funkhouser, Sequin 93] T. A. Funkhouser and C. H. Sequin. “Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments.” In *Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series*, edited by James T. Kajiya, pp. 247–254, New York: ACM Press, 1993.
- [Gottschalk et al. 96] S. Gottschalk, M. Lin, and D. Manchoa. “OBBTree: A Hierarchical Structure for Rapid Interference Detection.” In *Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series*, edited by Holly Rushmeier, pp. 171–180, Reading, MA: Addison-Wesley, 1996.
- [Wu 92] X. Wu: “A Linear-time Simple Bounding Volume Algorithm.” In *Graphics Gems II*, edited by David Kirk, pp. 301–306, Boston: Academic Press, 1992.

Web Information:

The C source code in this paper, including the hull vertex table data, is available online at <http://www.acm.org/jgt/papers/SchmalstiegTobler99>

Dieter Schmalstieg, Vienna University of Technology, Karlsplatz 13/186/2, A-1040 Vienna, Austria (dieter@cg.tuvien.ac.at)

Robert F. Tobler, Vienna University of Technology, Karlsplatz 13/186/2, A-1040 Vienna, Austria (rft@cg.tuvien.ac.at)

Received February 19, 1999; accepted August 27, 1999

Current Contact Information:

Dieter Schmalstieg, Graz University of Technology, Inffeldgasse 16, A8010 Graz, Austria (schmalstieg@icg.tu-graz.ac.at)

Robert F. Tobler, VRVis Research Center, Donau-City-Strasse 1/3 OG, A1030 Vienna, Austria (tobler@vrvis.at)