Real-time Bounding Box Area Computation

Dieter Schmalstieg and Robert F. Tobler
Vienna University of Technology

Abstract: The area covered by a 3D 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 2D area of a 3D oriented bounding box, and show how it can be computed equally fast or faster than its approximation with a 2D bounding box enclosing the projected 3D 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 [1]. Doing this exactly would require fist 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 are view-dependent hierarchical radiosity algorithms, 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. However, usually axis-aligned bboxes are used. which can also be a poor fit for the enclosed object.

In contrast, an oriented bounding boxes (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. [2]. For the construction of an OBB, refer to [3].

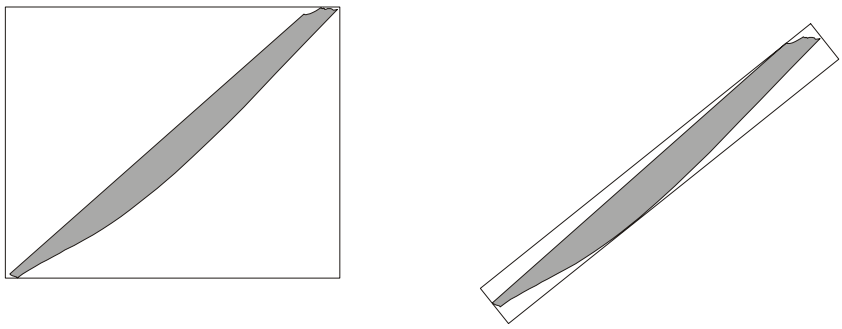

Figure 1: Axis-aligned bounding boxes (left) are often inferior to oriented bounding boxes (right)

To estimate the 2D area of a 3D object when projected to the screen, after perspective projection of the corners of an axis-aligned bbox, the area of the rectangle (2D bbox) enclosing the 3D bbox is used to estimate the area of the object on the screen. This procedure generates two nested approximation which are not necessarily a tight fit, so the error can be rather significant.

We propose to directly project an OBB and compute the area of the enclosing 2D polygon. This procedure yields significantly better approximations. Moreover, we will show that the procedure can be coded to require less operations than the simpler approach with nested bounding boxes. This is especially important as the computation of a bounding box is done many times at runtime and fast computation is essential.

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.

2.1 Viewpoint classification

When a 3D box is projected to the screen either 1, 2, or 3 adjacent faces are visible, depending on the viewpoint (Figure 2):

- **Case 1**: 1 face visible, 2D hull polygon consists of 4 vertices
- **Case 2**: 2 faces visible, 2D hull polygon consists of 6 vertices
- **Case 3**: 3 faces visible, 2D hull polygon consists of 6 vertices

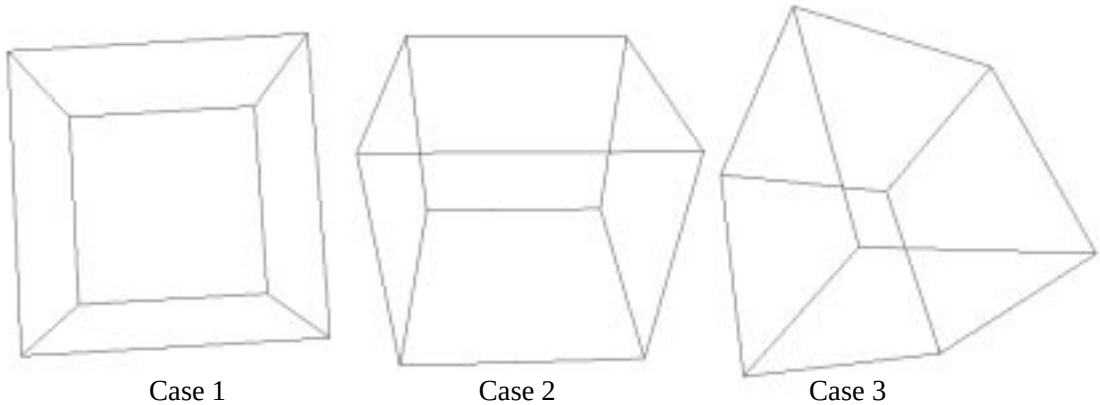

Figure 2: 1, 2, or 3 faces of a box may be visible

Whether a particular placement of the viewpoint relatively to the bbox yields case 1, 2, or 3, can be determined by examining the position of the viewpoint with respect to the 6 planes defined by the 6 faces of the bbox (Figure 3). These 6 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.

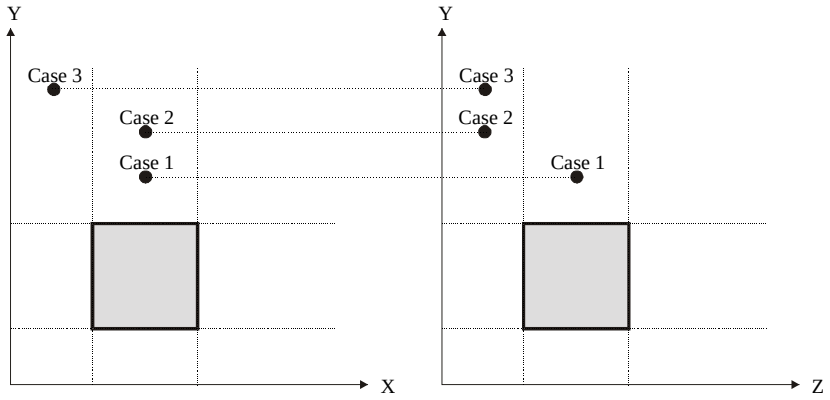

Figure 3: There are 3 cases of the viewpoint's position relative to the box

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

While the position of the viewpoint with respect to each plane can be computed using a three-dimensional dot product, it is simpler 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.

2.2 Area computation

After the classification, the area of the hull polygon must be computed from the bbox vertices given in the hull vertex table. Before the actual area computation, the vertices - either 4 or 6, but not all 8 - must be projected into 2D.

A fast method to compute the area of a planar polygon uses a simple contour integral approach: The edges of the hull polygon are visited in order. For each edge, the signed area between the x-axis and the edge is computed as the area of a rectangle with the height of the midpoint along the edge. For example, the area contribution from A to B is computed as (bx-ax)(ay+by)/2 (compare Figure 4). The sum of the contributions from all edges yields the area of the polygon as the contour integral over the hull polygon.

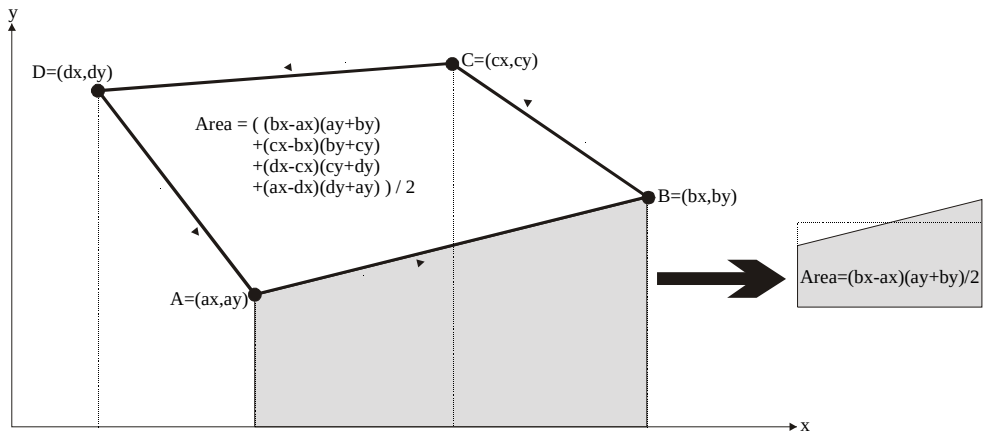

Figure 4: Computing the area of a polygon using a contour integral

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 2D, as well as the number of vertices in the outline (4 or 6, with 0 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.

Case	**Num**						**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	5	4	7	3	bottom, left
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	2	1	5	4	7	3	front, bottom, left

Case	**Num**						**Description**
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

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 8 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 = 0; 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 0.0;  //look up number of vertices
    for(i=0; i<num; i++) dst[i] := projectToScreen(bbox[hullvertex[pos][i]]);
    for(i=0; i<num; i++) sum += (dst[ i ].x - dst[ (i+1) % num ].x)
                              * (dst[ i ].y + dst[ (i+1) % num ].y);
    return sum * 0.5;                                 //return corrected value
}
```

4. Discussion

The proposed implementation gives superior results to a simple “2D bbox of 3D bbox” implementation. However, although it yields better accuracy, it can be implemented to use slightly fewer operations than the simple 2D 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 3D vector is multiplied with a 3x4 matrix, using 12 multiplications and 9 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 6 planes, this step uses at most 6 comparisons (a minimum of 3 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 4 or 6 vertices. The 3D 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 component are transformed using 3 multiply 2 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 1 perspective division and one add operation. The x and y component are then normalized, taking 2 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 as outlined in 2.2 requires 1 add, 1 subtract, and 1 multiply operation, plus one add operation for the running score, except for the first edge. The result must be divided by 2 (1 multiply operation). The total number of operations again depends on whether there are 4 or 6 vertices (and edges).

The total number of operations is 159 for case 2 and 3 (6 vertices), and 115 for case 1 (4 vertices). See Table 3 for comparison, the number of operations requires to compute a simple 2D box area is 163. It can be computed as follows: Projection of 8 vertices (18x8

operations), computation of the 2D bbox of the projected vertices using min-max tests (2x8 comparisons), 2D box computation (2 subtract, 1 add, 1 multiply operation).

	sum	mult	div	add/ sub	cmp
perspective projection	18	10	1	7	
2*4 mult	8	8			
2*3 add	6			6	
normalization factor	2		1	1	
2 * normalize	2	2			
affine transformation	21	12		9	
area segment	4	1		3	
OBB area, 6 vertices	**159**				
viewpoint to OBB space	21	12		9	
classify 6 planes	6				6
project 6 vertices	108	60	6	42	
area comp. 6 segments	24	6		18	
OBB area, 4 vertices	**115**				
viewpoint to OBB space	21	12		9	
classify 6 planes	6				6
project 4 vertices	72	40	4	28	
area comp. 4 segments	16	4		12	
2D bbox of 3D box	**163**				
project 8 vertices	144	80	8	56	
min-max 8 vertices	16				16
2D box area	3	1		2	

Table 3: Comparison of the number of operations required to compute the bbox area

As the number of operations required to compute the exact projected area of a 3D bbox is in the same order or even less expensive than the simple approach using a 2D 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.

5. References

[1] T. A. Funkhouser and C. H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments. Proceedings of SIGGRAPH'93, pages 247-254, 1993.

[2] S.Gottschalk, M. Lin, und D. Manchoa. Obb-tree: A hierarchical structure for rapid interence detection. In Proc. of ACM Siggraph '96, pages 171-180,1996.

[3] X. Wu: A linear-time simple bounding volume algorithm. In Graphics Gems II (David Kirk ed.), Academic Press, pages 301-306, 1992