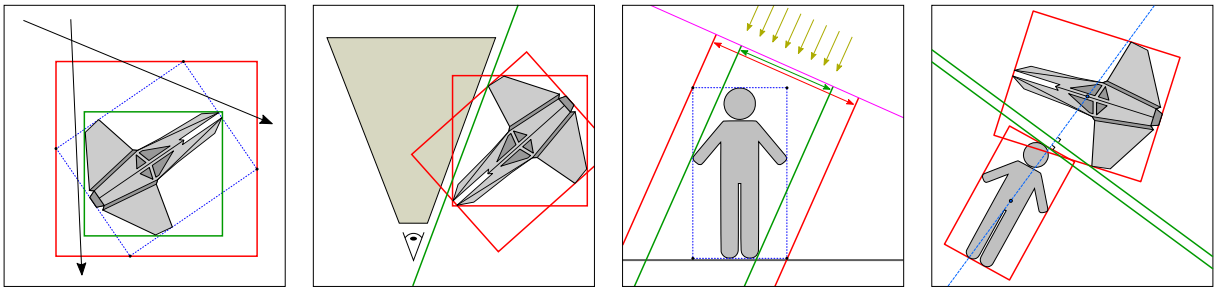
Eurographics Symposium on Rendering - Experimental Ideas & Implementations (2015)
J. Lehtinen and D. Nowrouzezahrai (Editors)

Apex Point Map for Constant-Time Bounding Plane Approximation

Samuli Laine Tero Karras

NVIDIA



(a) Real-time ray tracing (b) View frustum culling (c) Shadow map extents (d) Collision detection

Figure 1: *Many applications benefit from the ability to construct tight bounding planes with arbitrary normals.* ***(a)*** *In real-time ray tracing, a naïve way to construct world-space bounds for a moving object is to wrap the transformed object-space AABB (blue) with a world-space AABB (red). In contrast, our method makes it possible to take the world-space axes to object space and construct tightly fitting bounding planes to yield a better result (green).* ***(b)*** *In view frustum culling, both object-space AABB and tight world-space AABB (red) may indicate false visibility. Using a bounding plane constructed with the same normal as the frustum plane (green) enables better culling.* ***(c)*** *When determining per-object shadow map extents for a directional light, using the object's bounding box results in wasted texels (red). Bounding the object with planes fit according to light direction (green) provides better usage of the available shadow map resolution.* ***(d)*** *For object-vs-object overlap test in collision detection, we can rapidly test for non-overlap by choosing a potential separating axis (blue) and constructing a pair of bounding planes (green) perpendicular to the chosen axis. This can give more accurate results than using object bounding boxes (red).*

Abstract

We introduce apex point map, a simple data structure for constructing conservative bounds for rigid objects. The data structure is distilled from a dense k-DOP, and can be queried in constant time to determine a tight bounding plane with any given normal vector. Both precalculation and lookup can be implemented very efficiently on current GPUs. Applications include, e.g., finding tight world-space bounds for transformed meshes, determining per-object shadow map extents, more accurate view frustum culling, and collision detection.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations

1. Introduction

Bounding representations for objects are immensely useful in computer graphics. They can be used for quick determination of various overlap and intersection tests, because if overlap/intersection does not occur against a conservative bounding volume, it will not occur against the object either. A myriad of different bounding volume representations for different tasks are described in the scientific literature. We cannot hope to touch all of them in this paper, but direct the interested reader to surveys on the topic ([vdB99], [Eri04] ch. 4).

DOI: 10.2312/sre.20151166

A common source of grief is that bounding representations can be wrongly aligned for the task we would like to use them for. For example, in real-time ray tracing, the main acceleration structure is often a bounding volume hierarchy (BVH) of axis-aligned bounding boxes (AABBs). Dynamically moving rigid objects can be handled by transforming the ray into the object's model space when such object is encountered during ray traversal. For this purpose, a separate world-space BVH is reconstructed or refitted every frame, where each leaf stores a transformation matrix for one object and points to the corresponding per-object BVH [WBS03]. However, finding good world-space axis-aligned bounds for the leaves of the world-space hierarchy is problematic. The traditional, naïve way is to take the object's model-space AABB, transform it into world space, and wrap it with a world-space AABB. As illustrated in Figure 1(a), this can result in overly conservative AABB which leads to false hits and unnecessary ray-vs-object traversals. Similar problems arise in many other applications such as view frustum culling [AM99], shadow map fitting [BS07], and collision detection (e.g., [GLM96]), depicted in Figure 1(b–d).

All of these problems are due to available bounding volumes being misaligned for the task at hand. In this paper, we introduce *apex point map*, a data structure that allows rapid construction of bounding planes with arbitrary normal vectors. This altogether removes the need to wrestle with badly aligned bounding representations, as perfectly oriented bounds are always available at low cost.

The bounding planes generated by our method are not exact, but they are conservatively correct and we demonstrate that they generally fit the object tightly. The accuracy can be tuned through the resolution of the apex point map, which also affects the memory consumption and the cost of precalculation. The subsequent bounding plane construction has a constant cost regardless of the resolution.

Background. Constructing a bounding plane for a given polygonal mesh is a matter finding the extremal point along the desired normal vector and positioning the plane so that it passes through this point. The exact solution can be found in linear time by testing each vertex of the mesh in turn, but this becomes prohibitively expensive as the number vertices increases. The execution time can be reduced drastically by using a simpler bounding representation in place of the original object. For accurate results, the bounding representation needs to capture the shape of the object faithfully.

A particularly versatile bounding representation was originally proposed by Kay and Kajiya [KK86] and later came to be called *discrete orientation polytope* (k-DOP) [KHM*98]. The idea is to construct k bounding planes for the original object using a fixed set of normal vectors, typically distributed uniformly in the direction space. The object is then approximated using a convex polytope defined as the intersection of k half-spaces, each corresponding to one of the bounding planes [KZ97].

While the k-DOP representation can be used directly for constructing bounding planes, this approach suffers from two major shortcomings. First, it requires extracting the surface topology of the intersection polytope, which is remarkably difficult to do in a robust fashion. This is studied in the field of linear programming where it is known as the vertex enumeration problem. Second, no efficient method exists for finding the extremal k-DOP vertex along a given vector, necessitating a costly linear scan over the vertices.

2. Apex point map

Our data structure is a derived representation of a dense k-DOP, designed to enable extremely simple and efficient bounding plane lookups. Instead of storing distance values for a discrete set of directions, we divide the direction space into a set of *facets* and store a single *apex point* for each facet to act as a conservative representative of the k-DOP surface geometry. We organize the apex points according to an implicit indexing scheme through careful choice of the k-DOP plane normals, allowing us to find them in constant time.

Our method consists of a precalculation phase (Section 2.1) followed by a lookup phase (Section 2.2). The precalculation is done once per object by first approximating the object using a traditional k-DOP and then distilling the resulting planes into a set of apex points. These operations can be parallelized perfectly and involve only a single scan over mesh vertices, resulting in excellent fit for the current GPUs (Section 4). In the lookup phase, the apex point map is consulted for a given plane normal by first identifying the appropriate facet in direction space and then using the associated apex point to generate a valid bounding plane.

2.1. Precalculation

We choose the plane normal vectors in our initial k-DOP so that they point towards the vertices of a cube with each face tessellated into $n \times n$ squares. Figure 2 illustrates these normal directions (red dots) with resolution $n = 8$. Due to sharing of vertices at cube edges and corners, we have a total of $k = 6n^2 + 2$ planes in the k-DOP. This *direction cube* is most conveniently oriented to be axis-aligned in the model space of the object, and from now on we will assume this is the case.

We subdivide each of the $6 \times n \times n$ squares on the direction cube into two triangular facets, resulting in $12n^2$ facets in total. For each such triangle, we take the three k-DOP planes at its vertices and compute their intersection point. Note that this is always well defined, as the plane normals come from three obviously non-collinear points on the direction cube. This intersection point is the apex point for the facet. Figure 2 shows one facet highlighted in red, and the k-DOP normals whose planes determine its apex point are shown as red arrows.

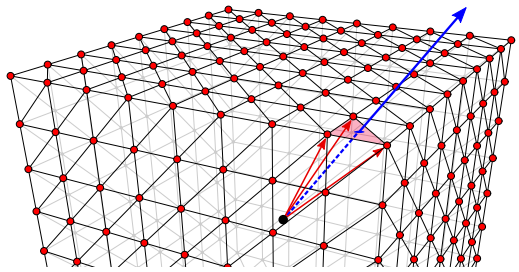


Figure 2: *Points on the tessellated direction cube (red dots) define the normal vectors of the original k-DOP planes. The surface of the cube is tessellated into facets, each of which is assigned an apex point during the precalculation. Highlighted red triangle illustrates a facet covering the directions between three k-DOP plane normals (red arrows). Blue arrow indicates a possible bounding plane normal for which the apex point stored for the highlighted facet would be used.*

The apex points constitute the entire output of the precalculation phase. Because there are $12n^2$ facets and we store one three-dimensional apex point for each, this yields $144n^2$ bytes using 32-bit floating point numbers. With $n = 8$ resolution illustrated in Figure 2, the total memory consumption of one apex point map is therefore 9 KB. We do not need to store the k-DOP normals, as they are implicitly defined.

Note that the direction cube that determines the original k-DOP normals and the facets exists only in the direction space. It is thus not in any sense centered about the object being bounded.

2.2. Bounding plane lookup

Let us now look at how a bounding plane can be constructed based on the apex point map. Given the desired normal $\mathbf{n}$ for the bounding plane, we first determine which facet of the direction cube it points at. For this purpose, $\mathbf{n}$ has to be transformed into the space where the direction cube is defined, i.e., the model space of the object.

For example, when given the vector illustrated as blue arrow in Figure 2, we need to find the index of the highlighted red facet. Due to the regular tessellation, this costs no more than a couple dozen arithmetic instructions.

After the facet index is known, we fetch its apex point from the precalculated data. The offset of the bounding plane is determined simply by requiring that it passes through the apex point. In other words, with the given normal vector $\mathbf{n}$ and fetched apex point $\mathbf{p}$, the final bounding plane is $\mathbf{n} \cdot \mathbf{x} + d = 0$, where $d = -\mathbf{n} \cdot \mathbf{p}$. This completes the bounding plane calculation.

Quite often there is a need to determine two bounding planes with opposite normals, for example when determining a world-space AABB for a transformed object. This can be exploited by storing the apex points for each facet and its opposite facet together, which lets us calculate the facet index only once per two planes. The performance of determining a single bounding plane does not suffer from this optimization.

3. Discussion

To see why our combination of precalculation and lookup produces conservatively correct bounding planes, let us consider the original k-DOP for the object for a moment. The k-DOP is clearly a conservative bounding volume for the object. Because it is an intersection of half-spaces, removing any of the half-spaces can only make the result more conservative.

In particular, picking any three half-spaces from the original k-DOP produces a conservatively correct bounding volume for the object, albeit an infinite one. Assuming these three k-DOP planes have a common point—which is always true in our case—they form an infinite pyramid-shaped bounding volume V with the intersection point $\mathbf{p}$ at its apex. The point stored in the apex point map for a facet is exactly this point, calculated for the subset of the three k-DOP planes whose normals point at the facet's vertices.

Therefore, our lookup phase works correctly as long as the fetched apex point $\mathbf{p}$ is a part of such pyramidal volume V that can be bounded by a plane with normal $\mathbf{n}$. Because $\mathbf{n}$ passes through the direction cube facet for which $\mathbf{p}$ was calculated, we know that $\mathbf{n}$ is surrounded by the normal vectors of the faces of V. It is easy to see that when this is the case, a plane with normal $\mathbf{n}$ can bound V, and the best bound is obtained by positioning the plane so that it passes through $\mathbf{p}$.

Our method shares similarities with the k-DOP realignment method of Zachmann [Zac98], where instead of enumerating the surface vertices of a k-DOP, it is assumed that they are the same as for a "generic" k-DOP where all planes have a constant offset of one. This allows one to assign a vertex (i.e., a plane triplet) of one k-DOP to each plane of another k-DOP with a different orientation, and to reuse this correspondence while processing two k-DOP hierarchies that are rotated with respect to one another. However, no implicit indexing is employed in Zachmann's method, so the calculation of the correspondences is not a constant-time operation per plane.

Compared to convex hull based methods, our precalculation is vastly simpler, and the memory consumption (with given resolution) of the apex point map is constant regardless of the mesh geometry. Comparing to oriented bounding boxes (OBBs), our data structure can be seen as their superset, as it allows constant-time construction of OBBs with arbitrary orientations. Due to our larger memory consumption, we envision apex point maps to be useful mostly on a per-object granularity, but less suitable than AABBs and OBBs for per-node bounds in large hierarchies.

Anisotropic scale optimization. For objects that are reasonably spherical, a regularly tessellated direction cube provides good distribution of k-DOP normals. However, objects that are flatter on one or two axes may suffer from an even distribution of k-DOP normals.

An easy, non-invasive improvement to the basic method is to scale the direction cube anisotropically according to the inverse per-axis scale of the input object. This is conceptually equivalent to pre-scaling the object so that its model-space bounding box becomes a cube. To implement this, we apply a per-component scale factor to the k-DOP normal vectors during precalculation, and to the lookup normals $\mathbf{n}$ during the bounding plane computation.

Extension to skinned meshes. Our method can be extended to support skinned meshes by computing a separate apex point map for each bone (i.e., skinning matrix), taking its weight to be one for each influenced vertex. When computing a bounding plane for the entire mesh, we query each of the apex point maps, transforming $\mathbf{n}$ through the respective bone transformation matrices, and take the most conservative plane offset as the result.

This works because each skinned vertex position is a convex combination of its transformed positions with each individual bone that influences it (with a weight of one). As these extreme positions are included in the individual apex point maps, the conservative bounding plane for these will bound the final vertex position as well. The result is more conservative than with rigid objects, though.

Alternative direction space parameterizations. We have used a regularly tessellated cube as the parameterization for the direction space, but other alternatives are also possible. For example, octahedral maps [ED08] would provide a similarly simple way to compute facet indices during lookup, and would thus be suitable for our purposes. We have not examined the performance and quality implications of alternative parameterizations.

4. Implementation

The most computationally intensive part of our method is generating the k-DOP for a given mesh, which involves calculating a dot product between each mesh vertex and k-DOP plane normal and tracking the maximum value for each normal. Since our plane normals always come in antipodal pairs, we can track both the minimum and maximum and consider only a half of the plane normals to reduce the total number of dot products by 50%. With $n = 8$, we thus need a total of $k/2 = 193$ dot products per vertex.

For maximum efficiency, we implemented the k-DOP generation in CUDA using a two-level parallelization scheme, where the vertices are distributed across warps (i.e., groups of 32 threads), and the normals are distributed across individual threads in a warp. Each warp tracks the minimum and maximum dot products for the full set of plane normals

Figure 3: *Test scenes used in the performance and quality measurements.*

and processes a subset of the mesh vertices. The first thread in a warp tracks the first $m = \lceil k/32 \rceil$ normals, the second one tracks the m subsequent normals, and so on. We treat m as a compile-time constant and employ loop unrolling to process one vertex in $5m$ warp instructions ($3\times$FFMA, $2\times$FMNMX) while keeping all the data in registers.

To read the vertices from global memory efficiently, we dereference 32 consecutive vertices at a time using the `__restrict__` modifier. We then broadcast each vertex in turn across the threads using the `__shfl` intrinsic, calculate the dot products, and proceed to the next 32 vertices. At the end, we perform global reduction over the per-warp results by having each thread write out its minimum and maximum into a global memory using atomic operations. We minimize the reduction overhead by launching a fixed number of resident warps and distributing the work among them in batches of 128 vertices using a global counter.

5. Results

We benchmarked the performance of the precalculation phase on an NVIDIA GTX 980 GPU using six meshes (Figure 3) as test input. The results are summarized in Table 1. The column for resolution $n = 8$, corresponding to illustration in Figure 2 is highlighted. On large vertex counts, we see precomputation performance of approximately 1.5M vertices per millisecond using this resolution, and approximately 360K vertices per millisecond for the smallest one tested. The construction of the apex point maps is therefore a relatively cheap operation if performed at scene load time.

We assessed the quality of the bounds obtained with apex point maps by constructing a world-space AABB for each test mesh in 100 000 random orientations, and comparing the resulting AABB surface area against the true world-space AABB calculated for each orientation. Table 2 shows the average and worst-case AABB surface area increase compared to the optimum. The anisotropic scale optimization was used in these measurements.

scene	#vertices	direction cube resolution n							
		2	4	6	**8**	10	12	14	16
Mustang	14259	.01	.02	.03	**.04**	.05	.06	.07	.09
Fairy	51390	.02	.03	.06	**.07**	.08	.09	.12	.15
Armadillo	172974	.04	.06	.11	**.15**	.20	.24	.31	.41
Buddha	543652	.10	.13	.25	**.39**	.53	.68	.89	1.16
Veyron	967113	.18	.22	.42	**.66**	.91	1.17	1.55	2.02
Hairball	1470000	.26	.33	.62	**.98**	1.35	1.76	2.34	3.03

Table 1: *Precalculation time in milliseconds for various scenes on NVIDIA GTX 980. For the highlighted $n = 8$ direction cube resolution, we can process approximately 1.5M vertices per millisecond for large meshes.*

With the highlighted $n = 8$ cube map resolution, the AABB computed using our method had only 1.3–2.2% larger surface area than the optimal AABB, on average. Even in the worst cases among the tested 100 000 orientations, our AABB had at most 10% slack in this set of meshes. The naïve method of using the transformed model-space AABB corners (cf. the red box in Figure 1(a)) gave much more conservative results in both average and worst cases. From the table, we see that even a small apex point map with resolution of $n = 2$ or $n = 4$ (consuming 0.56 KB and 2.25 KB of memory, respectively) provided a significant improvement over the naïve method. These improvements translate directly to, e.g., fewer ray-vs-object traversals in the real-time ray tracing use case [WBS03].

The anisotropic scale optimization reduced the worst-case AABB area increase in some scenes, but had no effect on the average-case results. Wth resolution $n = 8$, the greatest impact was in Buddha scene, where the worst-case AABB area increase was 15.9% without the optimization, and 7.3% with the optimization. Mustang improved from 8.0% to 4.9%, and other scenes saw no appreciable benefit.

We did not benchmark the performance of bounding plane construction using apex point maps, but we expect that the per-plane memory fetch of apex point constitutes the majority of the cost. For example, when computing a bounding box with an arbitrary orientation, we need to fetch six points, whereas fetching an object-space AABB to implement the naïve method would require fetching two points. We have not identified a use case where such tiny performance differences would become significant.

6. Conclusion

We have presented apex point map, a practical data structure for on-demand construction of arbitrary bounding planes for rigid objects. We see it as a simple tool for use cases where using conventional bounding boxes leads to overly conservative results. Due to the small precalculation and bounding plane lookup costs, the more accurate bounds can provide a net benefit in various applications by enabling, e.g., more aggressive culling of work.

Average AABB area increase in %

scene	naïve	direction cube resolution n							
		2	4	6	**8**	10	12	14	16
Mustang	44.2	10.8	5.0	2.9	**1.8**	1.3	0.9	0.6	0.5
Fairy	95.5	16.9	5.1	2.1	**1.3**	0.8	0.6	0.4	0.3
Armadillo	94.3	16.0	4.8	2.3	**1.3**	0.8	0.6	0.4	0.3
Buddha	57.7	13.4	5.0	2.6	**1.5**	1.0	0.8	0.6	0.5
Veyron	51.9	15.5	6.7	3.6	**2.2**	1.5	1.0	0.7	0.6
Hairball	124.7	19.7	5.0	2.2	**1.3**	0.8	0.6	0.4	0.3

Worst-case AABB area increase in %

scene	naïve	direction cube resolution n							
		2	4	6	**8**	10	12	14	16
Mustang	57.1	18.2	10.7	6.9	**4.9**	3.7	2.9	2.6	2.2
Fairy	170.6	62.3	21.7	10.4	**10.0**	7.4	5.7	5.5	3.9
Armadillo	168.9	47.1	22.6	14.3	**9.6**	5.3	4.5	3.7	3.8
Buddha	102.0	31.5	16.3	10.7	**7.3**	5.3	4.5	4.3	3.7
Veyron	69.2	25.7	14.4	8.5	**6.1**	4.5	3.5	2.8	2.4
Hairball	179.2	33.1	12.8	6.4	**4.4**	3.1	2.4	2.0	1.6

Table 2: *Quality of world-space AABBs over 100 000 random rotations for the mesh.* ***Top:*** *Average AABB area increase (in %) over optimal AABB.* ***Bottom:*** *Worst-case AABB area increase (in %) over optimal AABB. Column "naïve" refers to the technique where transformed object-space AABB is surrounded with a world-space AABB.*

References

[AM99] ASSARSSON U., MÖLLER T.: *Optimized View Frustum Culling Algorithms for AABBs and OBBs*. Tech. rep., 1999.

[BS07] BUNTIN S., STAMMINGER M.: Instanced shadow maps. In *Proc. CGAMES '07* (2007), pp. 135–142.

[ED08] ENGELHARDT T., DACHSBACHER C.: Octahedron environment maps. In *Proc. Vision, Modeling, and Visualization Workshop; VMV '08* (2008).

[Eri04] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: OBBTree: A hierarchical structure for rapid interference detection. In *Proc. SIGGRAPH '96* (1996), pp. 171–180.

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *Visualization and Computer Graphics, IEEE Transactions on 4*, 1 (1998), 21–36.

[KK86] KAY T. L., KAJIYA J. T.: Ray tracing complex scenes. In *ACM SIGGRAPH computer graphics* (1986), vol. 20, ACM, pp. 269–278.

[KZ97] KONEČNÝ P., ZIKAN K.: Lower bound of distance in 3D. In *Proc. of WSCG* (1997), vol. 3, pp. 640–649.

[vdB99] VAN DEN BERGEN G.: *Collision detection in interactive 3D computer animation*. PhD thesis, Eindhoven University of Technology, 1999.

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics; PVG '03* (2003).

[Zac98] ZACHMANN G.: Rapid collision detection by dynamically aligned DOP-trees. In *Proc. IEEE Virtual Reality Annual International Symposium; VRAIS '98* (1998), pp. 90–97.