1

Fast Computation of Tight-Fitting Oriented Bounding Boxes

Thomas Larsson
Linus Kallberg
Malardalen University, Sweden

1.1 Introduction

Bounding shapes, or containers, are frequently used to speed up algorithms in games, computer graphics, and visualization [Ericson 2005]. In particular, the oriented bounding box (OBB) is an excellent convex enclosing shape since it provides good approximations of a wide range of geometric objects [Gottschalk 2000]. Furthermore, the OBB has reasonable transformation and storage costs, and several efficient operations have been presented such as OBB-OBB [Gottschalk et al. 1996], sphere-OBB [Larsson et al. 2007], ellipsoid-OBB [Larsson 2008], and ray-OBB [Ericson 2005] intersection tests. Therefore, OBBs can potentially speed up operations such as collision detection, path planning, frustum culling, occlusion culling, ray tracing, radiosity, photon mapping, and other spatial queries.

To leverage the full power of OBBs, however, fast construction methods are needed. Unfortunately, the exact minimum volume OBB computation algorithm given by O'Rourke [1985] has $O(n^3)$ running time. Therefore, more practical methods have been presented, for example techniques for computing a (1- 8) approximation of the minimum volume box [Barequet and Har-Peled 1999]. Another widely adopted technique is to compute OBBs by using principal component analysis (PCA) [Gottschalk 2000]. The PCA algorithm runs in linear time, but unfortunately may produce quite loose-fitting boxes [Dimitrov et al. 2009]. By initially computing the convex hull, better results are expected since this keeps internal features of the

model from affecting the resulting OBB orientation. However, this makes the method superlinear.

The goal of this chapter is to present an alternative algorithm with a simple implementation that runs in linear time and produces OBBs of high quality. It is immediately applicable to point clouds, polygon meshes, or polygon soups, without any need for an initial convex hull generation. This makes the algorithm fast and generally applicable for many types of models used in computer graphics applications.

1.2 Algorithm

The algorithm is based on processing a small constant number of extremal vertices selected from the input models. The selected points are then used to construct a representative simple shape, which we refer to as the *ditetrahedron*, from which a suitable orientation of the box can be derived efficiently. Hence, our heuristic is called the ditetrahedron OBB algorithm, or DiTO for short. Since the chosen number of selected extremal vertices affects the running time of the algorithm as well as the resulting OBB quality, different instances of the algorithm are called DiTO-*k*, where *k* is the number of selected vertices.

The ditetrahedron consists of two irregular tetrahedra connected along a shared interior side called the base triangle. Thus, it is a polyhedron having six faces, five vertices, and nine edges. In total, counting also the interior base triangle, there are seven triangles. Note that this shape is not to be confused with the triangular dipyramid (or bipyramid), which can be regarded as two pyramids with equal heights and a shared base.

For most input meshes, it is expected that at least one of the seven triangles of the ditetrahedron will be characteristic of the orientation of a tight-fitting OBB. Let us consider two simple example meshes-a randomly rotated cube with 8 vertices and 12 triangles and a randomly rotated star shape with 10 vertices and 16 triangles. For these two shapes, the DiTO algorithm finds the minimum volume OBBs. Ironically, the PCA algorithm computes an excessively large OBB for the canonical cube example, with a volume approximately two to four times larger than the minimum volume, depending on the orientation of the cube mesh. Similarly, it also computes a loose-fitting OBB for the star shape, with a volume approximately 1.1 to 2.2 times larger than the optimum, depending on the given orientation of the mesh. In Figure 1.1, these two models are shown together with their axis-aligned bounding box (AABB),

OBB computed using PCA, and OBB computed using DiTO for a random orientation of the models.

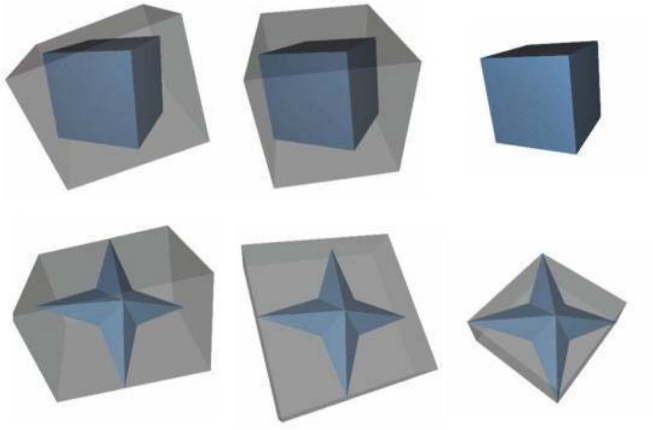

Figure 1.1. Computed boxes for a simple cube mesh (12 triangles) and star mesh (16 triangles). The first column shows the AABB, the second column shows the OBB computed by PCA, and the last column shows the OBB computed by DiTO. The meshes were randomly rotated before the computation.

The OBB is represented here by three orthonormal vectors **u**, **v**, and **w** defining the orientation of the box's axes, the half-extents h_u, k_v, and h_w defining the size of the box, and a midpoint m defining the center of the box. In the following, we explain how the algorithm works, and we present our experimental evaluation.

Selecting the Extremal Points

Call the set containing the n vertices of the input model P. The algorithm starts off by selecting a subset S with k representative extremal points from the vertices P, where k is an even number. This is done by finding the vertices with minimum and maximum projection values along $s = k/2$ predefined directions or normal vectors. The points in S are stored systematically so that any extremal point pair ($\mathbf{a}_j$, $\mathbf{b}_j$) along a direction $\mathbf{n}_j$; can be retrieved, where $\mathbf{a}_j$ and $\mathbf{b}_j$ are the points with minimum and maximum projection values, respectively. Note that all the points in S are guaranteed to be located on the convex hull of the input mesh. Hopefully, this makes them important for the later determination of the OBB axes. Ideally, the used normal vectors should be uniformly distributed in direction space. However, to be able to optimize the projection calculations that are calculated by sim ple dot products, normals with many 0s and 1 s may be preferable, given that they sample the direction space in a reasonable manner.

Clearly, the DiTO-k algorithm relies on the choice of an appropriate normal set N_s, and simply by choosing a different normal set a new instance of DiTO-k is created. In the experiments described later, five normal sets are used, yielding five algorithm instances. The normal sets are listed in Table 1.1. The normals in N_6, used in DiTO-12, are obtained from the vertices of a regular icosahedron with the mirror vertices removed. Similarly, the normals in N_{10}, used in DiTO-20, are taken from the vertices of a regular dodecahedron. The normal set $N_{16} = N_6 \cup N_{10}$ is used in DiTO-32. The normals in N_7 and N_{13}, used in DiTO-14 and DiTO-26, are not uniformly distributed, but they are still usually regarded as good choices for computing k-DOPs [Ericson 2005]. Therefore, they are also expected to work well in this case.

Table 1.1. Efficient normal sets N_6, N_{10}, N_7, and N_{13} used for DiTO-12, DiTO-20, DiTO-14, and DiTO-26, respectively, with the value $a = (\sqrt{5} - 1)/2 \approx 0.61803399$. The normals in N_6 and N_{10} are uniformly distributed.

N_6	N_{10}	N_7	N_{13}
(0, 1, a)	(0, a, 1+ a)	(1, 0, 0)	(1, 0, 0)
(0, 1, -a)	(0, a, -1- a)	(0, 1, 0)	(0, 1, 0)
(1, a, 0)	(a, 1+ a, 0)	(0, 0, 1)	(0, 0, 1)
(1, -a, 0)	(a, -1- a, 0)	(1, 1, 1)	(1, 1, 1)
(a, 0, 1)	(1+a, 0, a)	(1, 1, -1)	(1, 1, -1)
(a, 0, -1)	(1+a, 0, -a)	(1, -1, 1)	(1, -1, 1)
	(1, 1, 1)	(1, -1, -1)	(1, -1, -1)
	(1, 1, -1)		(1, 1, 0)
	(1, -1, 1)		(1, -1, 0)
	(1, -1, -1)		(1, 0, 1)
			(1, 0, -1)
			(0, 1, 1)
			(0, 1,-1)

Finding the Axes of the OBB

To determine the orientation of the OBB, candidate axes are generated, and the best axes found are kept. To measure the quality of an orientation, an OBB covering the subset S is computed. In this way, the quality of an orientation is determined in $O(k)$ time, where $k << n$ for complex models. For very simple models, however, when $n \leq k$, the candidate OBBs are of course computed using the entire point set P. To measure the quality, the surface areas of the boxes are

used, rather than their volumes. This effectively avoids the case of comparing boxes with zero volumes covering completely flat geometry. Furthermore, it makes sense to reduce the surface area as much as possible for applications such as ray casting. Of course, if preferred, a quality criterion based on the volume can be used in the algorithm instead.

The best axes found are initialized to the standard axis-aligned base, and the best surface area is initialized to the surface area of the AABB covering *P*. Then the algorithm proceeds by calculating what we call the *large base triangle*. The first edge of this triangle is given by the point pair with the furthest distance among the s point pairs in *S*, that is, the point pair that satisfies

$$\max \|\boldsymbol{a}_j - \boldsymbol{b}_j\|.$$

Call these points $\mathbf{p}_0$ and $\mathbf{p}_1$. Then a third point $\mathbf{p}_2$ is selected from *S* that lies furthest away from the infinite line through $\mathbf{p}_0$ and $\mathbf{p}_1$. An example of a constructed large base triangle is shown on the left in Figure 1.2.

The base triangle is then used to generate three different candidate orientations, one for each edge of the triangle. Let $\mathbf{n}$ be the normal of the triangle, and $\mathbf{e}_0 = \mathbf{p}_1 - \mathbf{p}_0$ be the first edge. The axes are then chosen as

$$\begin{aligned}\mathbf{u}_0 &= \mathbf{e}_0 \,/\, \|\mathbf{e}_0\|\,,\\ \mathbf{u}_1 &= \mathbf{n} \,/\, \|\mathbf{n}\|\,,\\ \mathbf{u}_2 &= \mathbf{u}_0 \,/\, \mathbf{u}_1.\end{aligned}$$

The axes are chosen similarly for the other two edges of the triangle. For each computed set of axes, an approximation of the size of the resulting OBB is computed by projecting the points in S on the axes, and the best axes found are kept. In Figure 1.3, an example of the three considered OBBs for the base triangle is shown.

Next, the algorithm proceeds by constructing the ditetrahedron, which consists of two connected tetrahedra sharing the large base triangle. For this, two additional points $\mathbf{q}_0$ and $\mathbf{q}_1$, are computed by searching *S* for the points furthest above and below the plane of the base triangle. An example of a ditetrahedron constructed in this way is shown on the right in Figure 1.2. This effectively generates six new triangles, three top triangles located above the plane of the base triangle and three bottom triangles located below the base triangle. For each one of these triangles, candidate OBBs are generated in the same way as already described above for the large base triangle, and the best axes found are kept.

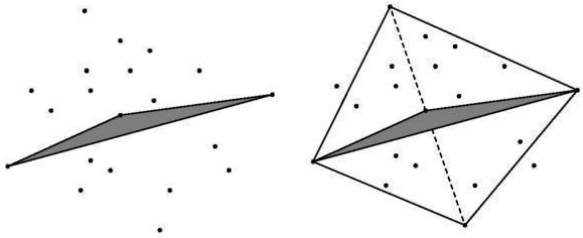

Figure 1.2. Illustration of how the large base triangle spanning extremal points (left) is extended to tetrahedra in two directions by finding the most distant extremal points below and above the triangle surface (right).

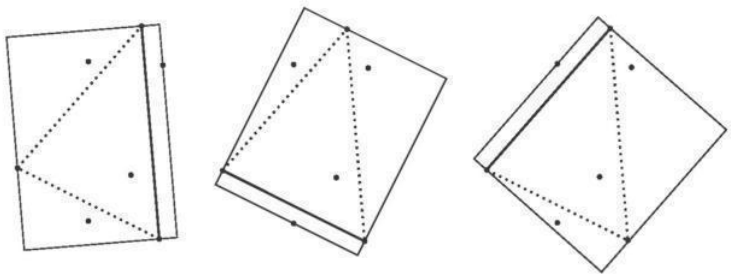

Figure 1.3. The three different candidate orientations generated from the normal and edges of the large base triangle. In each case, the box is generated from the edge drawn with a solid line.

After this, all that remains is to define the final OBB appropriately. A final pass through all n vertices in P determines the true size of the OBB, that is, the smallest projection values s_u, s_v, and s_w, as well as the largest projection values l_u, l_v, and l_w, of P along the determined axes **u**, **v**, and **w**. The final OBB parameters besides the best axes found are then given by

$$h_u = \frac{l_u - s_u}{2},$$
$$h_v = \frac{l_v - s_v}{2},$$
$$h_w = \frac{l_w - s_w}{2},$$
$$\mathbf{m} = \frac{l_u - s_u}{2}\mathrm{u} + \frac{l_v - s_v}{2}\mathrm{v} + \frac{l_w - s_w}{2}\mathrm{w}.$$

The parameters h_u, h_v, and h_w are the half-extents, and **m** is the midpoint of the box. Note that **m** needs to be computed in the standard base, rather than as the midpoint in its own base.

A final check is also made to make sure that the OBB is still smaller than the initially computed AABB; otherwise, the OBB is aligned with the AABB instead. This may happen in some cases, since the final iteration over all n points in P usually grows the OBB slightly compared to the best-found candidate OBB, whose size only depends on the subset S.

This completes our basic presentation of the DiTO algorithm. Example source code in C/C++ for the DiTO-14 algorithm is available, which shows how to efficiently implement the algorithm and how the low-level functions work.

Handling Detrimental Cases

There are at least three cases of detrimental input that the algorithm needs to detect and handle appropriately. The first case is when the computation of the first long edge in the base triangle results in a degenerate edge; that is, the two points are located at the same spot. In this case, the OBB is aligned with the already computed AABB, and the algorithm is aborted.

The second case arises when the computation of the third point in the base triangle results in a degenerate triangle; that is, the point is collinear with the endpoints of the first long edge. In this case, one of the OBB axes is aligned with the already-found first long edge in the base triangle, and the other two axes are chosen arbitrarily to form an orthogonal base. The dimensions of the OBB are then computed, and the algorithm is terminated.

The third detrimental case is when the construction of a tetrahedron (either the upper or the lower) fails because the computed fourth point lies in the plane of the already-found base triangle. When this happens, the arising triangles of the degenerate tetrahedron are simply ignored by the algorithm; that is, they are not used in the search for better OBB axes.

1.3 Evaluation

To evaluate the DiTO algorithm, we compared it to three other methods referred to here as AABB, PCA, and brute force (BF). The AABB method simply computes an axis-aligned bounding box, which is then used as an OBB. While this method is expected to be extremely fast, it also produces OBBs of poor quality in general.

The PCA method was first used to compute OBBs by Gottschalk et al. [1996]. It works by

first creating a representation of the input model's shape in the form of a covariance matrix. High-quality OBB axes are then assumed to be given by the eigenvectors of this matrix. As an implementation of the PCA method, we used code from Gottschalk et al.'s RAPID source package. This code works on the triangles of the model and so has linear complexity in the input size.

The naive BF method systematically tests 90 × 90 × 90 different orientations by incrementing Euler angles one degree at a time using a triple-nested loop. This method is of course extremely slow, but in general it is expected to create OBBs of high quality which is useful for comparison to the other algorithms. To avoid having the performance of BF break down completely, only 26 extremal points are used in the iterations. This subset is selected initially in the same way as in the DiTO algorithm.

All the algorithms were implemented in C/C++. The source code was compiled using Microsoft Visual Studio 2008 Professional Edition and run singlethreaded using a laptop with an Intel Core2 Duo T9600 2.80 GHz processor and 4 GB RAM. The input data sets were triangle meshes with varying shapes and varying geometric complexity. The vertex and triangle counts of these meshes are summarized in Table 1.2. Screenshots of the triangle meshes are shown in Figure 1.4.

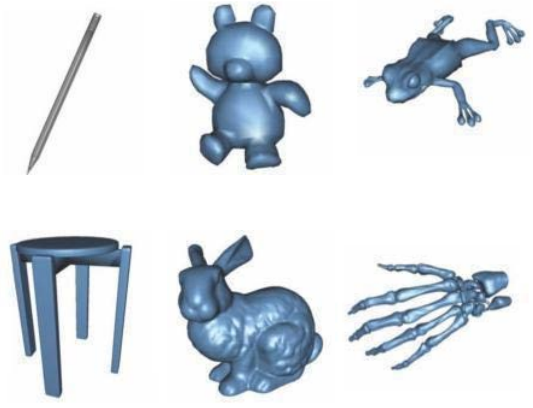

Figure 1.4. Visualizations of the triangle meshes used for evaluation of the algorithms.

Table 1.2. The number of vertices and triangles for the polygon meshes used for evaluation of the algorithms.

Model	Vertices	Triangles
Pencil	1234	2448
Teddy	1598	3192
Frog	4010	7964
Chair	7260	14,372
Bunny	32,875	65,536
Hand	327,323	654,666

To gather statistics about the quality of the computed boxes, each algorithm computes an OBB for 100 randomly generated rotations of the input meshes. We then report the average surface area A_{avg}, the minimum and maximum surface areas A_{min} and A_{max}, as well as the average execution time t_{avg} in milliseconds. The results are given in Table 1.3.

Table 1.3. The average, minimum, and maximum area as well as the average execution time in ms over 100 random orientations of the input meshes.

Pencil

Method	A_{avg}	A_{min}	A_{max}	t_{avg}
AABB	1.4155	0.2725	2.1331	0.02
PCA	0.2359	0.2286	0.2414	0.51
BF	0.2302	0.2031	0.2696	1009
DiTO-12	0.2316	0.1995	0.2692	0.11
DiTO-14	0.2344	0.1995	0.2707	0.09
DiTO-20	0.2306	0.1995	0.2708	0.14
DiTO-26	0.2331	0.1995	0.2707	0.15
DiTO-32	0.2229	0.1995	0.2744	0.20

Chair

Method	A_{avg}	A_{min}	A_{max}	t_{avg}
AABB	7.1318	4.6044	8.6380	0.07
PCA	4.7149	4.7139	4.7179	1.87
BF	3.6931	3.6106	4.6579	1047
DiTO-12	3.8261	3.6119	4.1786	0.35
DiTO-14	3.8094	3.6129	4.2141	0.25
DiTO-20	3.8213	3.6164	3.9648	0.37
DiTO-26	3.8782	3.6232	4.0355	0.35
DiTO-32	3.8741	3.6227	3.9294	0.49

Teddy

Method	A_{avg}	A_{min}	A_{max}	t_{avg}
AABB	3.9655	3.5438	4.3102	0.02
PCA	4.0546	4.0546	4.0546	0.60
BF	3.3893	3.3250	3.5945	1043
DiTO-12	3.7711	3.5438	4.0198	0.14
DiTO-14	3.7203	3.5438	3.9577	0.12
DiTO-20	3.7040	3.5438	3.8554	0.19
DiTO-26	3.7193	3.5438	3.8807	0.16
DiTO-32	3.7099	3.5438	3.8330	0.22

Bunny

Method	A_{avg}	A_{min}	A_{max}	t_{avg}
AABB	5.7259	4.7230	64833	0.19
PCA	5.2541	5.2540	5.2541	8.76
BF	4.6934	4.5324	4.9091	1041
DiTO-12	4.9403	4.5635	57922	1.13
DiTO-14	4.9172	4.5810	5.6695	0.98
DiTO-20	4.8510	4.5837	5.5334	1.55
DiTO-26	4.7590	4.5810	5.3967	1.42
DiTO-32	4.7277	4.6552	5.1037	2.04

Frog

Method	A_{avg}	A_{min}	A_{max}	t_{avg}
AABB	4.6888	3.0713	5.7148	0.07
PCA	2.6782	2.6782	2.6782	1.14
BF	2.7642	2.6582	3.5491	1037
DiTO-12	2.7882	2.6652	3.0052	0.28
DiTO-14	2.7754	2.6563	2.9933	0.24
DiTO-20	2.7542	2.6602	2.9635	0.40
DiTO-26	2.7929	2.6579	3.0009	0.36
DiTO-32	2.7685	2.6538	2.9823	0.44

Hand

Method	A_{avg}	A_{min}	A_{max}	t_{avg}
AABB	2.8848	2.4002	3.2693	1.98
PCA	2.5066	2.5062	2.5069	86.6
BF	2.3071	2.2684	2.4531	1067
DiTO-12	2.3722	2.2946	2.5499	11.8
DiTO-14	2.3741	2.2914	2.5476	10.0
DiTO-20	2.3494	2.2805	2.4978	15.5
DiTO-26	2.3499	2.2825	2.5483	14.5
DiTO-32	2.3372	2.2963	2.4281	20.6

The BF algorithm is very slow, but it computes high-quality OBBs on average. The running times lie around one second for each mesh since the triplenested loop acts like a huge hidden constant factor. The quality of the boxes var ies slightly due to the testing of somewhat unevenly distributed orientations arising from the incremental stepping of the Euler angles. The quality of boxes computed by the other algorithms, however, can be measured quite well by comparing them to the sizes of the boxes computed by the BF method.

The DiTO algorithm is very competitive. For example, it runs significantly faster than the PCA algorithm, although both methods are fast linear algorithms. The big performance difference is mainly due to the fact that the PCA method needs to iterate over the polygon data instead of iterating over the list of unique vertices. For connected triangle meshes, the number of triangles is roughly twice the number of vertices, and each triangle has three vertices. Therefore, the total size of the vertex data that the PCA method processes is roughly six times

as large as the corresponding data for the DiTO algorithm.

The DiTO algorithm also produces oriented boxes of relatively high quality. For all meshes except Frog, the DiTO algorithm computes OBBs with smaller surface areas than the PCA method does. For some of the models, the difference is significant, and for the Teddy model, the PCA method computes boxes that are actually looser fitting than the naive AABB method does. The DiTO algorithm, however, is in general more sensitive than the PCA method to the orientation of the input meshes as can be seen in the minimum and maximum area columns.

Among the included DiTO instances, there seems to be a small quality improvement for increasing k values for some of the models. DiTO-32 seems to compute the best boxes in general. The quality difference of the computed boxes, however, is quite small in most cases. Therefore, since DiTO-14 is approximately twice as fast as DiTO-32, it is probably the preferable choice when speed is a prioritized factor.

Bounding Volume Hierarchy Quality

We also tried the different OBB fitting algorithms for OBB hierarchy construction to evaluate them on different levels of geometric subdivision. On models with smooth curvature, the geometry enclosed on the lower levels of the hierarchy tend to get an increasing flatness with each level, and we wanted to compare the algorithms on this type of input. It turns out that our algorithm handles this better than the other algorithms we tested. This is visualized using the Bunny model in Figure 1.5.

For each one of the six test models, we applied the OBB fitting algorithms during the hierarchy construction. These test runs have been done only once for each model and OBB algorithm, with the model kept in its original local coordinate system, due to the longer construction times for whole hierarchies. Table 1.4 shows the total surface areas of the resulting hierarchies.

Although the RAPID source package also includes a hierarchy builder, we extracted the functions involved in fitting the OBBs and plugged them into our own hierarchy builder since it uses a more elaborate strategy for partitioning primitives that generally creates better hierarchy structures, albeit at a higher computational cost. The table does not show execution times because the time for hierarchy construction is influenced too much by factors other than the time to fit the bounding volumes, such as the strategy used to find clusters in the geometry to build a good tree structure. However, the construction times were about the same with all the algorithms. Note also that the BF algorithm is not included since it gives unreasonably long

construction times.

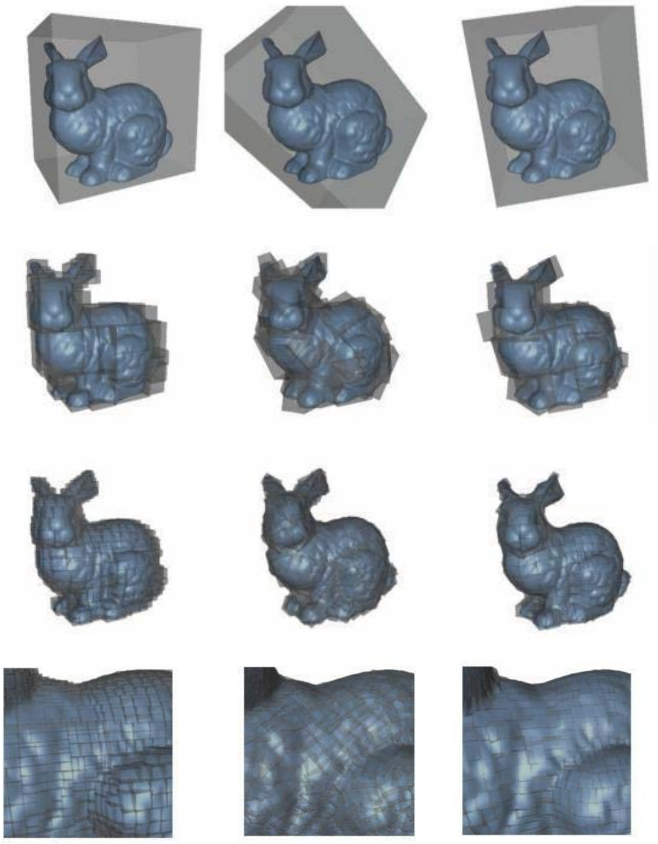

Figure 1.5. Levels 0, 6, 9, and 12 of OBB hierarchies built using AABBs (leftmost column), PCA (middle column), and DiTO-20 (rightmost column). As can be seen in the magnified pictures in the bottom row, PCA and DiTO both produce OBBs properly aligned with the curvature of the model, but the boxes produced by PCA have poor mutual orientations with much overlap between neighboring boxes.

The algorithm used for building the tree structure is a top-down algorithm, where the set of primitives is recursively partitioned into two subsets until there is only one primitive left, which is then stored in a leaf node. Before partitioning the primitives in each step, the selected OBB fitting procedure is called to create an OBB to store in the node. This means that the procedure is called once for every node of the tree. To partition the primitives under a node, we use a strategy that tries to minimize the tree's total surface area, similar to that used by Wald et al. [2007] for building AABB hierarchies.

As Table 1.4 shows, the DiTO algorithms create better trees than the other two algorithms do. An implication of the increasing flatness on the lower hierarchy levels is that the first base triangle more accurately captures the spatial extents of the geometry, and that the two additional tetrahedra get small heights. It is therefore likely that the chosen OBB most often is found from the base triangle and not from the triangles in the tetrahedra. The PCA fitter frequently produces poor OBBs, and in three cases (Chair, Bunny, and Hand), produces even worse OBBs than the AABB method does. It is also never better than any of the DiTO versions.

Table 1.4. Total surface areas of OBB hierarchies built using the different OBB fitting algorithms.

Model	AABB	PCA	DiTO-12	DiTO-14	DiTO-20	DiTO-26	DiTO-32
Pencil	9.64676	4.05974	3.03241	3.03952	3.03098	3.0111	3.00679
Teddy	90.6025	86.2482	71.9596	74.071	74.3019	73.1202	74.7392
Frog	56.1077	52.2178	42.7492	41. 9447	41.6065	41.2272	42.0487
Chair	81.3232	92.9097	64.5399	66.9878	64.2992	65.5366	64.8454
Bunny	170.01	171.901	125.176	122.108	119.035	119.791	119.172
Hand	112.625	117.241	50.8327	49.8038	50.0446	48.8985	49.7918

Interesting to note is that there is a weak correspondence between the number of extremal directions used in DiTO and the tree quality. This can be partly explained by the fact that the directions included in a smaller set are not always included in a larger set, which, for example, is the case in DiTO-14 versus DiTO-12. This means that for some models, fewer directions happen to give better results than a larger number of directions. Another part of the explanation is that the tested OBBs are only fitted to the set of extracted extremal points *S*, which means that good-quality OBBs might be missed because worse OBBs get better surface areas on the selected extremal points. All this suggests that execution time can be improved by using fewer extremal directions (see Table 1.3), while not much OBB quality can be gained by using more.

Note that the AABB hierarchies sometimes have somewhat undeservedly good figures because the models were kept in their local coordinate systems during the construction. This gives the AABB an advantage in, for example, Pencil and Chair, where much of the geometry is axis-aligned.

1.4 Optimization Using SIMD Instructions

By using data-parallelism at the instruction level, the execution speed of the DiTO algorithm can be improved substantially. As an initial case study, a new version of DiTO-14 has been written that utilizes Intel's Streaming SIMD Extensions (SSE). For this version of the algorithm, the vertices are pre-stored as groups of four vertices, which can be packed componentwise into three full SSE registers, one for each coordinate component. For example, the first four vertices

$$\begin{aligned}\mathbf{p}_0 &= (x_0, y_0, z_0),\\ \mathbf{p}_1 &= (x_1, y_1, z_1),\\ \mathbf{p}_2 &= (x_2, y_2, z_2),\\ \mathbf{p}_3 &= (x_3, y_3, z_3).\end{aligned}$$

are stored in the first vertex group as three arrays

$$\begin{aligned}\boldsymbol{x}_0 &= (x_0, x_1, x_2, x_3),\\ \boldsymbol{y}_0 &= (y_0, y_1, y_2, y_3),\\ \boldsymbol{z}_0 &= (z_0, z_1, z_2, z_3).\end{aligned}$$

Usually, this kind of data structure is referred to as structure of arrays (SoA).

There are two main passes in the algorithm, and they are the two loops over all input vertices. These two passes can be implemented easily using SSE. First, consider the last loop over all input vertices that determines the final dimensions of the OBB by computing minimum and maximum projection values on the given axes. This operation is ideal for an SSE implementation. By using the mentioned SoA representation, four dot products are computed simultaneously. Furthermore, the branches used in the scalar version to keep track of the most extremal projection values are eliminated by using the far more efficient minps and maxps SSE instructions.

Similarly, the first loop over all points of the model can also benefit a lot from using SSE since extremal projections along different axes are also computed. However, in this case the actual extremal points along the given directions are wanted as outputs in addition to the maximum and minimum projection values. Therefore, the solution is slightly more involved, but the loop can still be converted to quite efficient SSE code using standard techniques for branch elimination.

As shown in Table 1.5, our initial SIMD implementation of DiTO-14 is significantly faster than the corresponding scalar version. Speed-ups greater than four times are achieved for the most complex models, Bunny and Hand. To be as efficient as possible for models with fewer input points, the remaining parts of the algorithm have to be converted to SSE as well. This is particularly true when building a bounding volume hierarchy, since most of the boxes in the hierarchy only enclose small subsets of vertices.

Table 1.5. The execution time t for the scalar version of DiTO-14 versus t_v for the vectorized SSE version. All timings are in ms, and the speed-up factors s are listed in the last column.

Model	t	t_v	s
Pencil	0.09	0.035	2.6
Teddy	0.12	0.04	3.0
Frog	0.24	0.08	3.0
Chair	0.25	0.08	3.1
Bunny	0.98	0.21	4.7
Hand	10.0	1.99	5.0

1.5 Discussion and Future Work

The assumption that the constructed ditetrahedron is a characterizing shape for a tight-fitting OBB seems valid. According to our experiments, the proposed algorithm is faster and gives better OBBs than do algorithms based on the PCA method. Also, when building hierarchies of OBBs, our method gives more accurate approximations of the geometry on the lower hierarchy levels, where the geometry tends to become flatter with each level. In addition, our method is more general since it requires no knowledge of the polygon data.

We have not found any reliable data indicating which instance of the DiTO algorithm is best. The tested variations produce quite similar results in terms of surface area, although in some cases, there seems to be a small quality advantage with increasing sampling directions. For construction of OBB trees, however, it may be unnecessarily slow to use many sampling directions since n is less than k for a large number of nodes at the lower hierarchy levels.

Although the presented heuristics work fine for fast OBB computation, it would still be interesting to try to improve the algorithm further. Perhaps the constructed ditetrahedron can

be utilized more intelligently when searching for good OBB axes. As it is now, we have only considered the triangles of this shape one at a time. Furthermore, the construction of some simple shape other than the ditetrahedron may be found to be more advantageous for determining the OBB axes.

Finally, note that DiTO can be adapted to compute oriented bounding rectangles in two dimensions. The conversion is straightforward. In this case, the large base triangle simplifies to a base line segment, and the ditetrahedron simplifies to a *ditriangle* (i.e., two triangles connected by the base line segment). There are better algorithms available in two dimensions such as the rotating calipers method, which runs in $O(n)$ time, but these methods require the convex hull of the vertices to be present [Toussaint 1983].

References

[Barequet and Har-Peled 1999] Gill Barequet and Sariel Har-Peled. "Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions." SODA '99: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 98-91.

[Dimitrov et al. 2009] Darko Dimitrov, Christian Knauer, Klaus Kriegel, and Gunter Rote. "Bounds on the Quality of the PCA Bounding Boxes." *Computational Geometry: Theory and Applications* 42 (2009), pp. 772-789.

[Ericson 2005] Christer Ericson. *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann, 2005.

[Gottschalk et al. 1996] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection." *Proceedings of SIGGRAPH 1996*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, pp. 171-180.

[Gottschalk 2000] Stefan Gottschalk. "Collision Queries using Oriented Bounding Boxes." PhD dissertation, University of North Carolina at Chapel Hill, 2000.

[Larsson et al. 2007] Thomas Larsson, Tomas Akenine-Möller, and Eric Lengyel. "On Faster Sphere-Box Overlap Testing." *Journal of Graphics Tools* 12:1 (2007), pp. 3-8.

[Larsson 2008] Thomas Larsson. "An Efficient Ellipsoid-OBB Intersection Test." *Journal of Graphics Tools* 13:1 (2008), pp. 31-43.

[O'Rourke 1985] Joseph O'Rourke. "Finding Minimal Enclosing Boxes." *International Journal of Computer and Information Sciences* 14:3 (June 1985), pp. 183-199.

[Toussaint 1983] Godfried Toussaint. "Solving Geometric Problems with the Rotating Calipers." *Proceedings of IEEE Mediterranean Electrotechnical Conference* 1983, pp. 1-4.

[Wald et al. 2007] Ingo Wald, Solomon Boulos, and Peter Shirley. "Ray Tracing Reformable Scenes Using Dynamic Bounding Volume Hierarchies." ACM *Transactions on Graphics* 26:1 (2007).