Closest Point Query Among The Union Of Convex Polytopes Using Rasterization Hardware

Young J. Kim Kenneth Hoff Ming C. Lin and Dinesh Manocha
Department of Computer Science
University of North Carolina in Chapel Hill
{youngkim,hoff,lin,dm}@cs.unc.edu

Abstract: *We present a novel approach using rasterization hardware to perform the following query: Given a collection of convex polytopes in 3D, find the closest point from some given point inside the polytopes to the surface of the union of the polytopes. The algorithm takes advantage of multi-pass rendering, clipping and depth tests. We also demonstrate its application to penetration depth computation.*

1 Introduction

Interpolation-based polygon rasterization hardware has been increasingly used for geometric applications. These include visibility and shadow computations, CSG rendering, proximity queries, morphing, object reconstruction, etc. A recent survey on different applications is given in [TPK01]. All these algorithms perform computations in a discretized space (i.e. the image-space) and their accuracy is governed by the underlying pixel resolution.

We present a novel approach to perform the *closest point query* using rasterization hardware. The query can be stated as:

> Given m convex polytopes M_i in 3D, find the closest point from some given point $\boldsymbol{O}$ to the surface of M, where $M = \bigcup_i M_i$ and $\boldsymbol{O}$ is contained inside of M.

This query often arises in the classical motion planning problem, where a robot is represented as a point and obstacles are represented as the union of convex polytopes in the configuration space [Lat91]. It also plays a crucial role in the penetration depth (PD) problem [Cam97]. However, the union operation included in the query not only requires very expensive computation [AST97], but its implementation is also quite tedious and difficult to make it robust [Hof89]. As a result, the aforementioned problems either become intractable at interactive rates or need to be approached in a different way.

Our algorithm performs the closest point query up to the image-space resolution used in the computation. The main idea is to visualize the boundary of M from $\boldsymbol{O}$

without computing a surface representation of M explicitly. Afterwards, we compute the closest point, the distance and the direction by reading back the frame buffer. For a typical query with 70 convex polytopes with 30 faces on average, the algorithm takes 0.07 seconds on the NVIDIA GeForce 3 card. As an application example of our closest point query algorithm, we also demonstrate how to apply our closest point query algorithm to the penetration depth (PD) problem in Section 3.

2 Closest Point Query

Let us assume that we are given m convex polytopes M_i's in 3D space, and let us denote the union of M_i's as M. Our goal is, from some given point $\boldsymbol{O}$ in 3D, to compute the closest point on the boundary of M, ∂M, along with its distance and direction from $\boldsymbol{O}$. We use polygon rasterization hardware to perform this query up to image-space resolution.

2.1 Visualizing the Boundary of the Union

In order to visualize ∂M from a point $\boldsymbol{O}$ that is outside M, we simply display all the M_i's using the standard Z-buffer visibility algorithm. The nearest or minimum depth objects at each pixel will correctly construct the ∂M from the outside with only a single pass over each M_i. The resulting algorithm computes the closest point on ∂M from $\boldsymbol{O}$ up to image-space resolution. However, the same approach does not work if $\boldsymbol{O}$ is inside M. We present a new, incremental algorithm that can require m^2 passes, where m is the number of convex polytopes, M_i. The algorithm does not assume any kind of spatial ordering among M_i's.

Given the point $\boldsymbol{O}$ inside M, the normal Z-buffer minimum or maximum depth test may not suffice, since the visible internal boundaries may not even lie on ∂M. The algorithm has to remove the boundaries corresponding to the intersections between M_i's that do not belong to ∂M. If all the M_i's contain $\boldsymbol{O}$, then the Z-buffer maximum depth test will construct ∂M with a single pass over each M_i. However, we only know that at least one of the M_i's contains $\boldsymbol{O}$. So, we use an incremental algorithm that constructs ∂M out from $\boldsymbol{O}$.

Our algorithm for visualizing ∂M from a point inside is essentially a ray-shooting procedure from $\boldsymbol{O}$ to ∂M, and incrementally expands the front of ∂M. For example, in Fig. 1, we expand the current ∂M (thick line) by repeatedly rendering M_0, M_1, M_2. Each time M_0, M_1, M_2 are rendered, as shown in Fig. 1 (b)-(d), it opens up a new window (shown as dotted line) of the update region (thick gray line) on the current ∂M.

The algorithm maintains the current boundary of M, ∂M^k, where k is the current iteration, and incrementally expands it with M_i that intersects ∂M^k. We attempt to add M_i by rendering the front faces of M_i. The front faces that "pierce" the current ∂M^k open up a window through which $\boldsymbol{O}$ can see ∂M. Afterwards, we render the backfaces of M_i into the opened window using the maximum depth test.

In summary, the basic algorithm simply performs the following procedure:

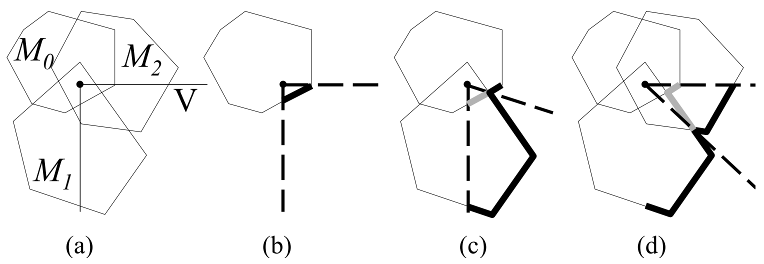

Figure 1: Visualizing the Boundary of the Union From Inside. *In (a), V is the current view-frustum. In (b), M_0 is rendered, and a new ∂M is constructed (thick line). In (c), when M_1 is rendered, it opens up a new window (dotted line), and the update region (thick gray line) on the current ∂M is established. Thus a new ∂M (thick line) is constructed. In (d), we perform the same procedure for M_2.*

ALGORITHM 2.1 - find boundary of M visible from $\boldsymbol{O}$

1. Initialize ∂M^0 to zero.
2. Repeat steps 3-5 m times.
3. Repeat steps 4-5 for each M_i.
4. Render the front faces of M_i, and using stencil operations, open a window where the depth value of the front faces is less than that of the current ∂M^k (pierces the current ∂M).
5. Render the back faces of M_i inside the stenciled window where the depth value of the back faces is greater than the current ∂M. This updates the ∂M^k in the window.

The Algorithm 2.1 correctly finds the portion of ∂M that is visible from $\boldsymbol{O}$ in the following sense. After the kth iteration in step 2 of Algorithm 2.1, ∂M^k includes the subset of ∂M that the ray can reach with less than or equal to $k-1$ *hops* from $\boldsymbol{O}$. Here, the *hop* on some point $\boldsymbol{p}$ on ∂M means the number of M_i's the ray should pass through to reach $\boldsymbol{p}$. For example, ∂M^1 includes the possible contribution to the final ∂M of all M_i's that contain $\boldsymbol{O}$ and have zero hops. Therefore, by induction on k, we correctly find the portion of ∂M that is visible from $\boldsymbol{O}$ after mth iteration.

2.2 Computing the Closest Point

For a given view, we can compute the closest point on the boundary by simply finding the pixel with the minimum distance value. The algorithm reads back the Z-buffer to obtain the depth values for each pixel. However, these depth values have undergone the perspective depth transformation and do not contain the non-linearity that is present in the distance values. The depth transformation is applied only at the vertices and has the following geometric properties: it preserves lines and planes between the transformed vertices and preserves depth relationships with respect to an orthographic view. This is not sufficient for finding the closest point. A pixel with the minimum depth value is not necessarily the closest point in terms of distance from $\boldsymbol{O}$.

The algorithm transforms the pixel depth values into distance values based on their (x, y) coordinate positions on the viewing plane. Each pixel depth value is divided by $\cos\theta$, where θ is the angle between the vector to the (x, y) position on the viewing plane and the center of viewing direction. This depth transformation is performed in software, and this operation typically takes a few milliseconds.

The minimum distance and direction to the closest point are derived from the pixel position containing the minimum transformed depth value. In order to examine views in all directions, we construct six views on the faces of a cube around $\boldsymbol{O}$ and repeat the operation.

2.3 Implementation and Acceleration Techniques

We implement the closest point query operation using OpenGL graphics library. The main code to draw ∂M from the inside is as simple as Program 2.1.

Program 2.1 OpenGL Code to Render ∂M From $\boldsymbol{O}$.

```
void DrawUnionOfConvex(ConvexPolytope *M_i, int Num_Of_M_i)
{
  glClearDepth(0);
  glClearStencil(0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_STENCIL_TEST);
  for (int i=0; i<Num_Of_M_i; i++)
    for (int j=0; j<Num_Of_M_i; j++)
    {
      glDepthMask(0);
      glColorMask(0,0,0,0);
      glDepthFunc(GL_LESS);
      glStencilFunc(GL_ALWAYS,1,1);
      glStencilOp(GL_KEEP,GL_REPLACE,GL_KEEP);
      M_i[j].DrawFrontFaces();

      glDepthMask(1);
      glColorMask(1,1,1,1);
      glDepthFunc(GL_GREATER);
      glStencilFunc(GL_EQUAL,0,1);
      glStencilOp(GL_ZERO,GL_KEEP,GL_KEEP);
      M_i[j].DrawBackFaces();
    }
}
```

This basic implementation can be improved by using two techniques. First of all, if we can compute a subset of the pairs, M_i's, that contain $\boldsymbol{O}$ (i.e. zero hops), we need to render them only once in the step 2 of algorithm 2.1. Once we have identified these l M_i's that contain $\boldsymbol{O}$, we first render them using the maximum depth test and then the remaining $(m - l)$ M_i's, $(m - l)$ times using the incremental algorithm.

Secondly, for each view, when we render the M_i's, we perform view-frustum culling by checking whether the axis aligned bounding box of each M_i lies in the current view. This object-space view frustum culling significantly reduces the number of primitives rendered during each iteration of the algorithm.

The performance of the algorithm depends on both the number of convex polytopes and the complexity (i.e. number of faces) of each polytope. When the above acceleration techniques are employed, the performance can also vary depending on the relative configuration among the polytopes. For a typical query with 45 convex polytopes with 40 faces each, the computation takes about 0.1 seconds. In another example with 70 convex polytopes with 30 faces each, the query takes about 0.07 seconds. The performance was measured on a 1.6 GHz PC with an NVIDIA GeForce 3 graphics card.

2.4 Accuracy of Closest Point Query

Our algorithm always computes an upper estimate to the closest point on the boundary of M. In other words, the algorithm may be conservative and the value of the computed answer may be larger than the value of the global minimum. The rasterization errors and precision of image-space computations governs the tightness of the resulting answer. The main sources of these errors are:

1. The discretization of ray directions to lie on a pixel grid for each view.
2. The fixed precision of the Z-buffer.

Increasing the resolution of the grid decreases the worst-case angular error that is proportional to the distance between adjacent pixels. Moreover, constructing tighter bounds on the minimum and maximum distances in each view (near and far plane distances), logarithmically decreases the Z-buffer precision error [Shr99]. When M_i is represented explicitly (e.g. B-rep), we can find the tight bounds on the near and far plane distances by explicitly computing minimum and maximum distances from $\boldsymbol{O}$ to each M_i, and taking the minimum and maximum of them. More information about the worst case bound of the closest point query can be found in [KLM02].

3 Application to Penetration Depth Computation

In this section, we demonstrate the application of hardware accelerated closest point query on a class of proximity query problems, namely penetration depth computation. Penetration depth (PD) is an important measure to quantify the amount of penetration between two intersected objects, and can be defined by their Minkowski sums. However, due to the fact that a Minkowski sum can have high combinatorial and computational complexities especially for non-convex polyhedra and it also requires robust union implementation, there has been no practical algorithm for the PD problem.

3.1 Penetration Depth and Minkowski Sums

Let P and Q be two intersecting polyhedra. The PD of P and Q, $PD(P, Q)$, is the minimum translational distance that one of the polyhedra must undergo to render them disjoint. A general framework to compute the PD is based on Minkowski sums. The Minkowski sum, $P \oplus Q$, is defined as a set of pairwise sums of vectors from P and Q. In other words, $P \oplus Q = \{\boldsymbol{p} + \boldsymbol{q} | \ \boldsymbol{p} \in P, \boldsymbol{q} \in Q\}$. Furthermore, $P \oplus -Q$ is defined by negating Q; i.e. $P \oplus -Q = \{\boldsymbol{p} - \boldsymbol{q} | \ \boldsymbol{p} \in P, \boldsymbol{q} \in Q\}$.

It is well known that one can reduce the problem of computing the PD between P and Q to a minimum distance query on the surface of their Minkowski sum , $P \oplus -Q$

[Cam97]. More specifically, if two polyhedra P and Q intersect, then the difference vector, $\boldsymbol{O}_Q - \boldsymbol{O}_P$, between the origins of P and Q is inside $P \oplus -Q$. The $PD(P,Q)$ is defined as a minimum distance from $\boldsymbol{O}_{Q-P}$ to the surface of $P \oplus -Q$, see Fig. 2. However, the worst case complexity of computing Minkowski sums for non-convex objects can be as high as $O(n^6)$, where n is the number of features in each object [AGHP$^+$00]. As a result, no practical algorithms are yet known for accurately computing the PD between non-convex objects.

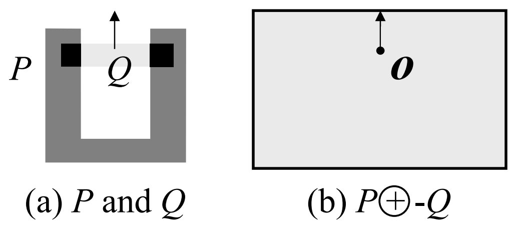

Figure 2: PD Computation Using Minkowski Sum. *(a) shows the situation where two polygons P and Q in 2D are intersected. (b) shows the Minkowski sum $P \oplus -Q$ of the two polygons in (a) (note that $Q = -Q$ in this example). The minimum distance from the origin to the surface of the Minkowski sum corresponds to the PD.*

3.2 Our Approach

Our approach for computing PD is also based on Minkowski sums between two overlapping polyhedra. However, we do not explicitly compute Minkowski sums due to its combinatorial and computational complexity. Instead, we perform the closest point query that computes the closet point from the origin to the implicit boundary of the Minkowski sums.

Minkowski sums has the following *decomposition* property. If $P = P_1 \cup P_2$, then $P \oplus Q = (P_1 \oplus Q) \cup (P_2 \oplus Q)$. Taking advantage of this fact, our basic PD algorithm computes PD as follows.

ALGORITHM 3.1 - PD computation using the closest point query

1. Compute a convex decomposition for each polyhedron
2. Compute the pairwise convex Minkowski sums between all possible pairs of convex pieces in each polyhedron
3. Perform the closest point query (Algorithm 2.1) from the origin to the boundary of the pairwise Minkowski sums.
4. The resulting minimum depth fragment computes an approximation to the PD, up to the image-space resolution used for this computation.

We can use bounding volume hierarchies, model simplification, and culling algorithms to further accelerate the above PD computation pipeline and refine the PD in a hierarchical manner.

In Fig. 3, we illustrate some Benchmark models for PD computation. The whole PD computation takes 0.2 seconds for the touching tori (2000 faces, 67 convex pieces), and 2 seconds for the CAD model (1692 faces, 425 convex pieces). During the PD computation, the closest point query spends 0.04 seconds for the tori, and 0.6 seconds for the CAD model. For more information about the PD algorithm and its performance, we refer the readers to [KLM02].

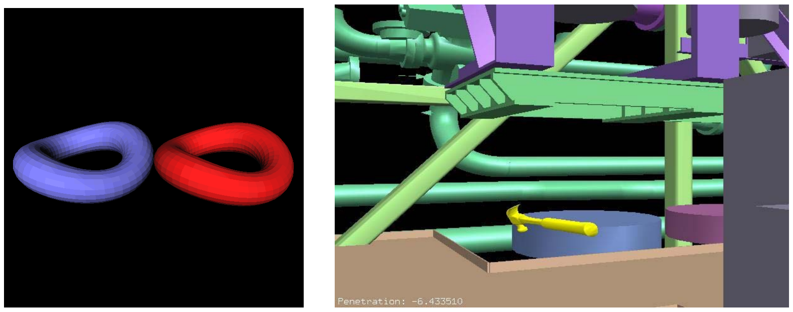

Figure 3: PD Benchmark Models. *Left: touching tori, Right: CAD Model*

4 Acknowledgments

This research was supported in part by ARO Contract DAAG55-98-1-0322, DOE ASCII Grant, NSF NSG-9876914, NSF DMI-9900157, NSF IIS-982167, NSF ACR-0118743, ONR Contracts N00014-01-1-0067 and N00014-01-1-0496, Intel. We thank Miguel A. Otaduy for his help in application to a complex CAD environment.

References

[AGHP$^+$00] P. Agarwal, L. J. Guibas, S. Har-Peled, A. Rabinovitch, and M. Sharir. Penetration depth of two convex polytopes in 3d. *Nordic J. Computing*, 7:227–240, 2000.

[AST97] Boris Aronov, Micha Sharir, and Boaz Tagansky. The union of convex polyhedra in three dimensions. *SIAM J. Comput.*, 26:1670–1688, 1997.

[Cam97] S. Cameron. Enhancing GJK: Computing minimum and penetration distance between convex polyhedra. *Proceedings of International Conference on Robotics and Automation*, pages 3112–3117, 1997.

[Hof89] C. Hoffmann. *Geometric and Solid Modeling*. Morgan-Kaufmann, San Mateo, CA, 1989.

[KLM02] Young J. Kim, Ming C. Lin, and Dinesh Manocha. Fast penetration depth estimation using rasterization hardware anbd hierarchical refinement. In *Fifth International Workshop on Algorithmic Foundations of Robotics*, Dec. 2002.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Shr99] Dave Shreiner, editor. *OpenGL Reference Manual*, pages 238–239. Addison-Wesley, Third edition, 1999.

[TPK01] T. Theoharis, G. Papaiannou, and E. Karabassi. The magic of the z-buffer: A survey. *Proc. of 9th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG*, 2001.