CInDeR
Collision and Interference Detection in Real-time using Graphics Hardware

Dave Knott[a,b] Dinesh K. Pai[a,c]

[a] Department of Computer Science, University of British Columbia
[b] Radical Entertainment
[c] Department of Computer Science, Rutgers University

Abstract

Collision detection is a vital task in almost all forms of computer animation and physical simulation. It is also one of the most computationally expensive, and therefore a frequent impediment to efficient implementation of real-time graphics applications. We describe how graphics hardware can be used as a geometric co-processor to carry out the bulk of the computation involved with collision detection. Hardware frame buffer operations are used to implement a ray-casting algorithm which detects static interference between solid polyhedral objects. The algorithm is linear in both the number of objects and number of polygons in the models. It also requires no preprocessing or special data structures.

Key words: Collision detection, Graphics hardware, Imagespace computations, Geometric modeling

1 Introduction

A vital task in almost all forms of computer animation or physical simulation is the act of *collision detection*. When polyhedral objects are being animated, it is critical to determine if and when they are about to, or have already, come into contact with each other. Example problem domains where collision detection is almost ubiquitous include rigid and deformable body simulation, computer games, virtual reality, surgical simulation, robotics, path planning, and computer-aided design and manufacturing (CAD/CAM). Collision detection is also one of the most computationally demanding tasks in each of these domains and therefore one of the most common bottlenecks in the simulation pipeline.

In recent years, the computational power of graphics hardware has made enormous leaps, not only in speed but also in functionality. A typical graphics processor now contains more logical transistors than the CPU of the computer that it resides in.

It is reasonable, then, to ask whether the computation involved with collision detection can be offloaded from the computer's primary processor and memory, and be performed instead on the graphics hardware. To do so frees up CPU power for other tasks and enables the application to make use of a computational resource that might otherwise be underutilized.

This paper proposes an image-space method for detecting interference between solid polyhedral bodies. The algorithm makes use of virtual ray casting to determine which portions of the edges of the polyhedrons in question lie within volumes enclosed by other polyhedrons.

The technique exhibits a number of features which, to our knowledge, no other interference detection algorithm has successfully combined:

- Convex and non-convex geometry with hollow regions can be handled.
- Large numbers of objects can be handled.
- Intersection tests are performed on the geometry itself, not on an approximation to the surface.
- No special data structures are required.
- No preprocessing of models is required.
- The algorithm's expected asymptotic running time is linear in both the number of objects being tested and the number of polygons comprising the objects.
- Processing is done with the aid of commodity-level graphics hardware.

The image-space nature of our technique means that the algorithm is an approximate one. The precision of interference computations is constrained by the resolution of the frame buffer region that we render into. The algorithm also cannot detect interference involving objects whose projection lies outside of that region.

2 Previous and Related Work

Collision and interference detection is a widely researched topic. For a comprehensive overview of the subject in general, we refer the reader to the surveys by Lin and Gottschalk [15] and by Jiménez *et al.* [12]. There has also been a good deal of recent research into using graphics hardware for geometric computation [16].

contact e-mail: {knott|pai}@cs.ubc.ca

There have been a number of previous attempts at using graphics hardware to aid in interference detection.

Perhaps the best known example is the work of Rossignac *et al.* [21]. They use the depth and stencil buffer capable hardware to aid in the inspection of cross-sections of computer-modelled mechanical assemblies. Clipping planes are moved through volumes occupied by the solid assemblies and rays are cast toward points on the planes. A point is known to be within the solid if a ray passes through an odd number of polygonal faces before reaching the point.

Shinya and Forgue [26] reported some early results of using a hardware depth buffer to support interference detection. They start with the assumption that all objects are convex. For each pixel, a list of the maximum and minimum depth values of each object is stored. These lists are then sorted. If any object's z_{max} and z_{min} values are not adjacent in the sorted lists, then two objects are interfering. The hardware is used to calculate the z_{min} and z_{max} depth maps of each object. The main drawback of this approach is the huge overhead of repeatedly copying the depth buffer and then sorting pixel depth values. Storing many depth maps also requires huge amounts of memory.

Myszkowski *et al.* [18] describe using the depth and stencil buffers in conjunction to detect inference. Of all the previously reported results, their work most closely resembles our own. As with our algorithm, they use the stencil buffer to store a running count of how many solid objects a ray enters and leaves before reaching a surface point of another object of interest. As with Shinya and Forgue, their method is applicable only to objects that are convex in the direction of the rays being cast. Also, their algorithm does not work for more than two objects.

This work was expanded on by Baciu and Wong [2, 3]. Their primary contribution was to extend the techniques developed by Myszkowski *et al.* to compute the area of the region of overlap between two interfering solids.

Vassilev *et al.* [28] have used an image-space depth and colour buffer technique for detecting collisions in cloth animation for computer-generated characters.

Interference detection is a subset of a more general class of computation which is sometimes called proximity queries. Hoff *et al.* [11] have demonstrated the use of graphics hardware to generate proximity information for two-dimensional objects. They perform image-space computations for collision detection, separation distance, penetration depth, and contact points and normals. Their method is applicable only for individual pairs of objects, and makes use of a distance field computation that was originally used in the context of generating Voronoi diagrams [9]. The technique has also recently been extended to proximity queries in three dimensions [10].

Graphics hardware has also recently been used by Kim *et al.* for the computation of penetration depth between pairs of 3D polyhedral models in the context of rigid-body simulation [14]. They make use of the depth buffer to aid in the computation of Minkowski sums in order to find the minimum translational vector needed to separate two interfering bodies.

2.1 Relationship to Shadow Algorithms

The initial inspiration for our algorithm came from the one of the most common shadowing techniques in real-time rendering: the *shadow volume* algorithm [6]. Using the shadow volume technique, a polygonal mesh is created that represents the volume of space that lies in the shadow cast by an object. Determining whether or not a point lies in shadow involves casting a ray from the viewer toward the point. The point is in shadow if the ray enters more shadow volumes than it exits. The test can be performed in hardware by using the stencil buffer to count the difference in the number of front-facing and back-facing polygons lying between a point and the viewer [8].

3 Background

In the context of the following discussion, a polyhedral solid is deemed to be a closed manifold, enclosing a finite volume. Unless otherwise noted, the polyhedron may be non-convex and may contain hollow regions.

3.1 Interference

Before describing our interference detection algorithm, we will give a brief description of what exactly we mean by interference.

We let $p \in P$ denote that point p is contained within solid polyhedral object P. Also let ∂P be the set of edges of P. We denote intersection of two polyhedral objects by $A \cap B$, and define it as follows: $\exists a \in A, b \in B | a = b$. When two objects intersect each other, we say that they are in *interference*.

Our interference detection algorithm is predicated on the following property: Two polyhedral objects are interfering with each other if and only if an edge of one object intersects the volume occupied by the other. This is expressed by the following theorem [5]:

Theorem 1 $A \cap B \neq \emptyset$ *iff* $\{\exists a \in \partial A, b \in B | a = b\}$ *or* $\{\exists a \in A, b \in \partial B | a = b\}$

We also note that this property is invariant under affine and projective transformations [27], meaning that an intersection test based on it may be applied at any point in the graphics pipeline.

3.2 Counting Boundary Crossings

A central feature of our algorithm is the concept of locating a point relative to a solid by casting a semi-infinite

ray from the point. The number of polygons of the solid's boundary that the ray passes through are counted. The two-dimensional version of this technique was first introduced in 1962 by Shimrat [25] and corrected by Hacker [7]. It is commonly used to solve the point-in-polygon problem, and was first used in the context of interference detection between solid models by Boyse [4].

It is a theorem of computational geometry that a semi-infinite ray originating within a closed solid will intersect the boundary of the solid an odd number of times [19]. This is a three-dimensional variation on the *Jordan Curve Theorem*, first formulated in two dimensions by Camille Jordan in 1893 [13], which states that any simple closed curve divides the plane into two regions.

In addition, for a directed ray, we can specify whether or not an intersection of the ray with a solid corresponds to the ray *entering* or *leaving* the solid's volume. We can make use of the fact that there may not be two consecutive instances of either an "enter" or a "leave". This means that a semi-infinite ray cast from the interior of a solid will "leave" the solid one more time than it "enters" the solid (Figure 1).

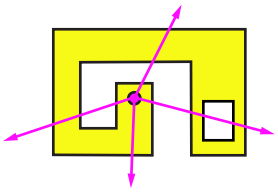


Figure 1: Casting rays from a point in a polygon

In our algorithm, we do not cast rays from objects towards infinity, but rather cast them from infinity toward objects. However, the same principles of counting boundary crossings still apply.

4 The Algorithm

Interference detection is performed by point-sampling the scene and looking for object edges that are interior to other polyhedrons. This is done using hardware ray-casting. Rays are cast through the pixels of the viewport toward objects of interest. Rays that strike those objects' edges are of particular interest. Figure 2 shows this in two dimensions. Note that the edge points appear disjoint. In a one-dimensional slice of the viewport, edge points will only be connected if the edge is colinear with the slice.

When a ray strikes an edge, then we count the difference in the number of back-facing and front-facing polygons lying between the edge point and the ray's origin at the viewport. If the difference is not equal to zero, then we know that the edge point lies within the volume of space occupied by another object.

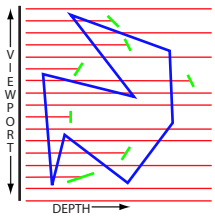


Figure 2: Rays cast at the edges of an object of interest. Polygons of another object enclose some edge points.

4.1 The Rendering Passes

The primary interference detection algorithm consists of the equivalent of three rendering passes. This is shown in pseudocode in Algorithm 1.

Algorithm 1 Detect Interference

```
 1: for all pixels do {clear depth and stencil buffers}
 2:     Z = 0, stencil = 0
 3: end for
 4: depth test = none
 5: Enable depth update
 6: stencil function = none
 7: Disable colour update
 8: for all objects do {draw the edge depths}
 9:     Draw edges {Pass #1}
10: end for
11: Disable depth update
12: depth test = '<'
13: for all objects do
14:     cull mode = back-face
15:     stencil function = increment
16:     Draw polygons {Pass #2: add front-facing polys}
17:     cull mode = front-face
18:     stencil function = decrement
19:     Draw polygons {Pass #3: subtract back-facing polys}
20: end for
21: for all pixels do {check for interference}
22:     if stencil > 0 then
23:         RETURN( interference=true )
24:     end if
25: end for
26: RETURN( interference=false )
```

We also illustrate the process with a sequence of explanatory images showing a two-dimensional version of two objects in interference. These figures show how we test whether the edge points of one object lie within the volume enclosed by the polygons of other object.

In the first rendering pass (lines 4-10), we render all of the edges that we wish to check for interference, and initialize the depth buffer with their depth values (Figure 3). This ensures that all rays cast through pixels will be targeted at polygon edges. The first pass is the only one in which the depth buffer is altered. All subsequent rendering passes perform depth tests relative to these values. What this means is that all rays cast through pixels will

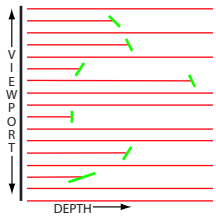


Figure 3: Initialize the depth buffer

either intersect an edge or go to infinity[1].

In the second and third rendering passes, we do not alter the depth buffer or the colour buffer. We do use depth testing, and reject all pixels that fail the depth test.

In the second rendering pass (lines 14-16), we draw only those polygons whose normals face toward the ray's origin (Figure 4). That is, we reject all polygons for which the dot product of the normal with the ray direction is positive. In the graphics hardware this corresponds to a back-face cull. We increment the stencil buffer for each pixel that passes the depth test.

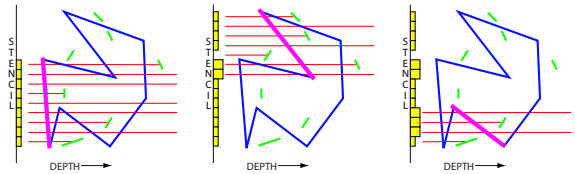


Figure 4: Increment the stencil buffer when rendering "front-facing" polygons.

In the third rendering pass (lines 17-19), we draw only those polygons whose normals face away from the ray's origin (Figure 5). That is, we reject all polygons for which the dot product of the normal with the ray direction is negative. In the graphics hardware this corresponds to a front-face cull. We decrement the stencil buffer for each pixel that passes the depth test.

Passes two and three have the combined effect of using the stencil buffer to count the difference between the number of front-facing and back-facing polygons that lie between the ray's origin (the viewport) and the edge point in question.

After the third pass, the stencil buffer value at each pixel gives the results of the collision detection:

- A positive stencil buffer value at a pixel represents a ray cast toward an interfering edge point. The magnitude of the value indicates how many objects the edge point is interfering with.

[1] Actually the far clipping plane, which is infinity for our purposes

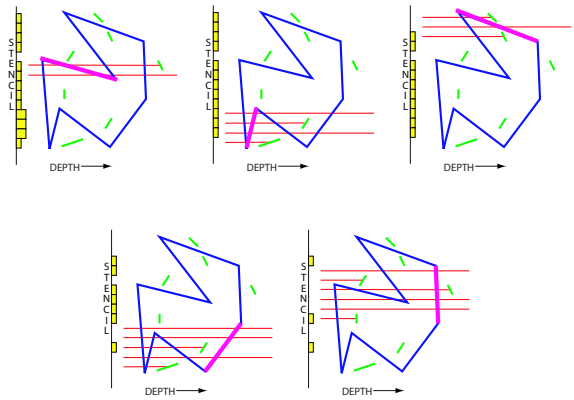


Figure 5: Decrement the stencil buffer when rendering "back-facing" polygons.

- A stencil buffer value of zero indicates that the ray represented by the pixel either (a) did not intersect an edge point, or (b) intersected an edge point that was not interfering with another object.
- A negative stencil buffer value means that the ray intersected more back-facing than front-facing polygons. This indicates that at least one object is not a closed manifold.

To check for interference, the values in the stencil buffer must be scanned. This requires the stencil values to be passed from the graphics hardware to the CPU. Note that at no point do we modify the colour buffer.

5 Object Identification

In our algorithm, an interference point corresponds to an object edge that penetrates the volume contained by another object. For each interfering edge point we can therefore make a distinction between the *penetrating* object and the *penetrated* object. Identifying the penetrating object is easy, but identifying the penetrated object is more difficult.

We identify objects by assigning them bit-wise unique integer values that we call identifiers. These identifiers are written into the colour buffer at locations where interference has been detected. The penetrated and penetrating objects write their identifiers into separate channels of the colour buffer. With a typical frame buffer, this affords us between 12 and 16 bits per identifier, allowing us to distinguish up to 65,000 objects. For most applications, this is enough identifiers to uniquely identify every polygon and every edge.

Finding the identity of interfering objects then amounts to retrieving the colour buffer and parsing through it. Any pixel with non-zero colour will uniquely identify a pair of interfering objects.

5.1 Identifying the Penetrating Object

To identify the penetrating object, we add an extra rendering pass that redraws object edges at only those pixels where the stencil buffer is positive. This writes the object's identifier at those locations where its edges were found to be in interference.

5.2 Identifying the Penetrated Object

Before identifying the penetrated object, we reset all positive stencil buffer values to zero, in order to remove the effects of identifying the penetrating object. This can be accomplished in the same rendering pass that we use to write the penetrating object's identity.

The basic idea is to repeat the counting process for each individual object. If, after repeating the counting for a single object, the stencil buffer was incremented (i.e. stencil=1), then we know that the object has an edge penetrating it. We write the object's identity into the frame buffer by redrawing the object's polygons and updating the buffer only where the stencil equals 1. In the same pass we also reset the stencil value to 0 whenever it equals 1, in order to restore the stencil buffer's original state.

6 Other Issues

6.1 Avoiding Frame Buffer Reads

We have found that, in practise, one of the most computationally expensive parts of the algorithm is the act of retrieving the frame buffer to main memory and parsing through it one pixel at a time. For instance, it can be shown that, on a standard PC, transferring a 256 by 256 pixel area of the colour buffer will take at least 1 msec [17]. This is slightly mollified by the Unified Memory Architecture (UMA) of systems such as the Microsoft XBox [1], but UMA is not common, and in any case we prefer to keep as much computation as possible local to the graphics processor.

We observe that, regardless of whether we are identifying the penetrating or penetrated object, we are rerendering either edges or polygons and writing pixels that must pass both a stencil test and a depth test. Testing for interference without reading the frame buffer can therefore be accomplished if the graphics hardware provides information about whether or not rendering a primitive resulted in some pixel passing both of these tests.

Fortuitously, this operation is supported in commodity-level hardware via hardware-based occlusion queries. Hardware occlusion queries were first introduced by Hewlett-Packard in their *Visualize fx* graphics hardware [23] and are also available in NVidia's latest graphics accelerators. Furthermore, this functionality is exposed in OpenGL through the *HP_occlusion_test* and *NV_occlusion_query* extensions. These extensions require almost no extra CPU or GPU overhead and do not require an extra rendering pass.

What is more, if objects are identified using occlusion queries, then the colour buffer does not need to be touched, since it is sufficient only to know that some pixel passed both the depth and stencil tests. However, it is important to note that when multiple pairs of objects are interfering, occlusion queries cannot be used to determine exactly which objects constitute each pair. We must still read the colour buffer to do this.

Early Non-Interference Detection

Even if we decide that the colour buffer needs to be read in order to resolve interference pairs, we can still use occlusion queries to perform *non-interference detection*. The colour buffer needs to be retrieved and parsed only if we know that interference is occurring. We can therefore use occlusion queries to perform a boolean test for interference (at virtually no cost) and then retrieve the colour buffer for inspection only if necessary.

6.2 Interference Localization

For many applications, it does not suffice to know only that two objects are interfering with each other. Rigid body simulation, for instance, requires knowledge of the surface points at which objects are interpenetrating, in order to correctly apply forces or impulses to separate contacting bodies.

Using a variation of the object identification scheme, it is possible to determine which edges are involved in the interference. During interference detection, we render each edge with an unique colour, which is used to identify the edge when we scan through the frame buffer. We therefore have a list of edges that intersect the volumes of other objects. This list obviously includes those edges that penetrate the surfaces of other objects, which makes the search for surface contact points much easier.

We also observe that every pixel in the frame buffer has an associated depth value. The pixel location and depth combine to give the screen-space location of an interfering point located by the ray cast through the pixel. The screen-space position can be subjected to a reverse transformation to find the world-space (or object-space) location of the associated interference point.

Additionally, our algorithm can easily be combined with other hardware-based algorithms that are specialized for localization. A good example is the proximity query techniques of Hoff *et al.* [10, 11]. In such a situation, our method can be used for coarse-grained detection, in order minimize the number of objects or polygons that need to be processed by the fine-grained proximity queries.

6.3 Offset Edges

Graphics hardware does not rasterize lines in exactly the same way that it rasterizes polygons. In particular, an

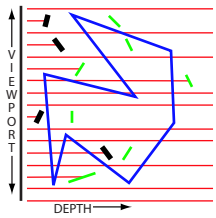


Figure 6: An undetectable interference. Interfering edges of one object are blocked by th e edges of another object.

edge of a polygon, when rasterized as a line, is not guaranteed to have exactly the same depth values as the corresponding "edge" of the rasterized polygon. Therefore, it is possible for the rasterized edge points of a polyhedron to lie slightly inside or outside its polygonal boundary. If an edge lies inside the boundary, then the algorithm will (incorrectly) determine that the edge is in interference with its own polyhedron. We therefore offset all edges by a small amount in the direction of the normals of their vertices, ensuring that such a situation cannot occur.

6.4 Multiple Objects and Non-Convex Geometry

Our ray casting algorithm, unlike previous efforts in hardware-assisted interference detection, can handle both non-convex geometry and large sets of potentially interfering objects.

The reason for this stems from the previous observation that if two objects are interfering with each other, then an edge of one of them must intersect the volume of the other. The only way for an intersection to miss detection by the algorithm is if no ray can see an interfering edge point. Recall that the depth buffer stores the depth values of the closest edges to the eyepoint. An obscured interference would require every interfering edge point to be occluded by another, closer edge point (Figure 6).

The most common cause of obscured interferences is a locally dense cluster of edges in the projection of the scene onto the viewport. Dense clusters of edges indicate that either the objects have very large edge counts, or the projection of the objects is taking up a small viewport area. This can be solved by increasing viewport resolution, but doing so is not very practical beyond a small amount. A better solution would be to perform some precomputation, such as bounding boxes, that minimizes the world-space area that rays are cast into.

6.5 Interference Detection Precision

The resolution of the viewport through which rays are being cast is of paramount importance. It directly affects the precision of the interference detection.

Suppose that we are using an orthographic projection. Let x_v, y_v be the image-space dimensions of the viewport, and x_w, y_w, z_w be the world-space dimensions of the view frustum. World-space precision of interference detection in the plane parallel to the viewport is then limited to $\frac{x_w}{x_v}$ by $\frac{y_w}{y_v}$.

Precision in the dimension perpendicular to the viewport is better, since this is limited by the higher precision of the depth buffer. For instance, a typical depth buffer has 24 bits of precision, while viewport size has the equivalent of between 7 and 10 bits of resolution.

It is therefore very important to make the view frustum as tight as possible around the objects being tested, in order to maximize the projection of the objects onto the viewport. This can be accomplished by doing a fast bounding volume calculation, for example.

If we use a perspective projection, then interference detection is more precise for objects that are closer to the viewpoint. For many applications this is actually a benefit, since detection will be more precise for interferences that are likely to be more noticable to a viewer [20].

7 Results

7.1 Implementation

We implemented the interference detection algorithm in the C++ programming language, using OpenGL [24] as the real-time rendering API. OpenGL is standardized across multiple computing architectures and stencil buffers are required by the standard, making it ideal for our purposes.

Our timing tests were performed on a computer with dual 1.8GHz Pentium IV CPUs and a graphics card that used the NVidia GeForce4 chip set.

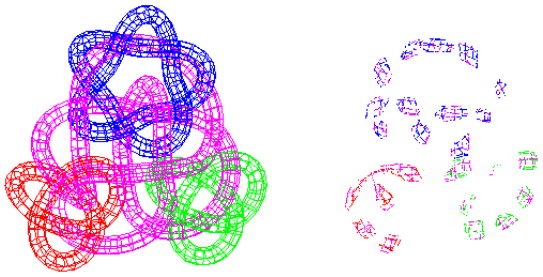
7.2 Examples

We have tested the algorithm with a wide variety of polyhedral objects.

An example is given in Figure 7, which shows a scene involving several highly non-convex objects [22] that are entangled and interfering with each other. Figure 7(a) shows the edges of the objects being tested. Figure 7(b) shows an enhanced version of the interferences reported in the stencil buffer.

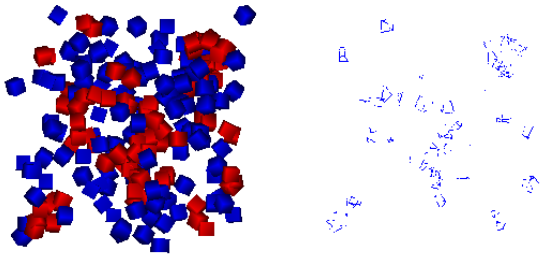
Another example is given in Figure 8, which shows a large number of objects, many of which in interference. Interfering objects are highlighted in red.

For low polygon count objects such as boxes, we were able to simultaneously detect interference between several hundreds of objects. We also tested the algorithm with objects of high polygon count. With small numbers of objects, we were able to use models with polygon counts of over five thousand before mesh density became too high for the algorithm to function correctly.



(a) Scene (b) Stencil Buffer

Figure 7: Non-convex objects in interference



(a) Scene (b) Stencil Buffer

Figure 8: Multiple objects in interference

7.3 Complexity Analysis

For collision detection involving multiple objects, the running time is usually a function of how many objects are involved. The naïve algorithm for N objects involves $\mathcal{O}(N^2)$ pair-wise tests. Our algorithm draws each object a constant number of times and is therefore $\mathcal{O}(N)$ in the number of objects involved.

For collision detection involving two polygonal objects, the running time is usually a function of the number of polygons. If the two objects are constructed of P_i and P_j polygons, respectively, then the naïve algorithm involves $\mathcal{O}(P_iP_j)$ polygon-polygon intersection tests. Our algorithm renders each edge or polygon a constant number of times and is therefore $\mathcal{O}(P)$ in the number of polygons, where P is the total number of polygons of all objects being tested.

7.4 Timings

We start the timing with the first command issued to the graphics card. Timing is concluded when either pixel parsing is finished or the last occlusion query result becomes available.

Figure 9 shows interference detection time as a function of the number of objects being tested. The objects were all boxes constructed as strips of twelve triangles.

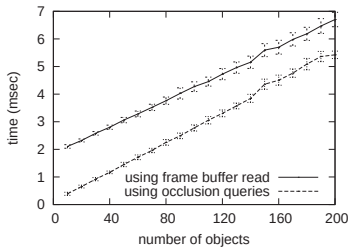


Figure 9: Timing data as a function of object count

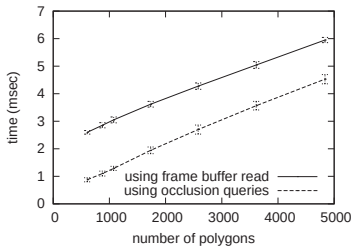


Figure 10: Timing data as a function of polygon count

The relationship is clearly linear.

Figure 10 shows interference detection time as a function of the number of polygons in the objects. In this example, we used objects with the same basic shape, but constructed at several levels of detail. The relationship here is also linear.

Timing values were taken as the mean over 100 trials. Vertical bars at data points show the standard deviation. The spatial configuration of the objects was randomized for each trial.

Note in particular that reading the frame buffer rather than using occlusion queries to identify objects can be fairly costly. In this example we rendered to a 256 by 256 pixel off-screen rendering surface.

8 Conclusions and Future Work

We have presented an algorithm for performing interference detection between solid polyhedral bodies in real-time with the aid of graphics hardware. The algorithm is linear in both the number of objects and the number of polygons comprising those objects. Non-convex geometry can be handled and no specialized data structures or preprocessing is required.

The precision of our hardware-assisted ray-casting is currently constrained by the dimensions of the viewport that we are rendering the objects into. Overcoming the limitations imposed by viewport resolution is a natural area for future work.

Our method could be extended to provide better interference localization. For applications such as rigid

body simulation, this would entail identifying the surface points at which edges intersect objects.

We believe there is some merit in combining our algorithm with level-of-detail techniques. This would allow us to perform fast rejection tests on coarse approximations to the polygonal models, and use the full model only when higher accuracy is needed.

Ray-casting is commonly used for a variety of other applications that require visibility information. Alternate uses for our algorithm is an avenue of future research.

Finally, we note that although our technique does not require the use of programmable graphics hardware, we expect that it could lead to improvements in this work. Vertex programs could be used to automatically generate the offsets required to avoid detecting self-interference. Similarly, using fragment shaders would allow us to render more complicated interference information.

Acknowledgements

We thank Dave Forsey for many interesting early discussions, and Rod Davison for his input on practical uses for this research.

Our research group is supported by grants from IRIS and NSERC. The first author was also supported by a Science Council of British Columbia GREAT Scholarship.

References

[1] M. Abrash. Inside Xbox graphics. *Dr, Dobb's Journal*, pages 21–26, Aug. 2000.

[2] G. Baciu and W. S.-K. Wong. Rendering in object interference detection on conventional graphics workstations. In *Pacific Graphics '97*, Oct. 1997.

[3] G. Baciu, W. S.-K. Wong, and H. Sun. RECODE: An image-based collision detection algorithm. In *Pacific Graphics '98*, Oct. 1998.

[4] J. W. Boyse. Interference detection among solids and surfaces. *Communications of the ACM*, 2(1):3–9, Jan. 1979.

[5] J. F. Canny. *The Complexity of Robot Motion Planning*. PhD thesis, Massachusetts Institute of Technology, 1987.

[6] F. C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, volume 11, pages 242–248, July 1977.

[7] R. Hacker. Certification of Algorithm 112: Position of point relative to polygon. *Communications of the ACM*, 5(12):606, Dec. 1962.

[8] T. Heidmann. Real shadows real time. *IRIS Universe*, (18):28–31, 1991.

[9] K. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, Aug. 1999.

[10] K. E. Hoff III, A. Zaferakis, M. Lin, and D. Manocha. Fast 3D geometric proximity queries between rigid & deformable models using graphics hardware acceleration. Technical Report TR02-004, Dept. of Computer Science, University of North Carolina at Chapel Hill, 2002.

[11] K. E. Hoff III, A. Zaferakis, M. C. Lin, and D. Manocha. Fast and simple 2D geometric proximity queries using graphics hardware. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 145–148, Mar. 2001.

[12] P. Jiménez, F. Thomas, and C. Torras. 3D collision detection: a survey. *Computers & Graphics*, 25(2):269–285, Apr. 2001.

[13] C. Jordan. *Cours d'analyse de l'Ecole Polytechnique*, volume 1. 2nd edition, 1893.

[14] Y. J. Kim, M. A. Otaduy, M. C. Lin, and D. Manocha. Fast penetration depth computation for physically-based animation. In *2002 ACM SIGGRAPH Symposium on Computer Animation*, pages 23–31,187, July 2002.

[15] M. C. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA Conference on Mathematics of Surfaces*, pages 37–56, Sept. 1998.

[16] M. C. Lin and D. Manocha, editors. *Interactive Geometric Computations Using Graphics Hardware: SIGGRAPH 2002 Course Notes #31*, July 2002.

[17] C. Maughan. Texture masking for faster lens flare. In M. A. DeLoura, editor, *Game Programming Gems 2*, pages 474–480. Charles River Media, Inc, 2001.

[18] K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995.

[19] J. O'Rourke. *Computational Geometry In C*. Cambridge University Press, 2nd edition, 1998.

[20] C. O'Sullivan and J. Dingliana. Collisions and perception. *ACM Trans. on Graphics*, 20(3):151–168, July 2001.

[21] J. Rossignac, A. Megahed, and B.-O. Schneider. Interactive inspection of solids: Cross-sections and interferences. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 353–360, July 1992.

[22] R. G. Scharein. *Interactive Topological Drawing*. PhD thesis, University of British Columbia, 1998.

[23] N. D. Scott, D. M. Olsen, and E. W. Gannett. An overview of the VISUALIZE fx graphics accelerator hardware. *The Hewlett-Packard Journal*, pages 28–24, May 1998.

[24] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 1.4). 2002.

[25] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Comm. of the ACM*, 5(8):434, Aug. 1962.

[26] M. Shinya and M.-C. Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):131–134, 1991.

[27] F. Thomas and C. Torras. A projectively invariant intersection test for polyhedra. *The Visual Computer*, 18(7):405–414, 2002.

[28] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics 2001)*, 20(3):260–267, 2001.