Generic Convex Collision Detection using Support Mapping

Benjamin Kenwright

Abstract—A collision detection algorithm that is computationally efficient, numerically stable, and straightforward to implement is a valuable tool in any virtual environment. This includes the ability to determine accurate proximity information, such as, penetration depth, contact position, and separating normal. We explore the practical and scalable issues of support mapping for use in detecting contact information for convex shapes. While support mapping is a popular technique used in common algorithms, such as, GJK, EPA, and XenonCollide, we demonstrate how to implement an uncomplicated algorithm and identify pitfalls in three-dimensional space. We explore the scalable nature of the technique for use in massively parallel execution environments and emphasise trade-offs in terms of performance and accuracy to achieve consistent real-time frame-rates through optimisations.

Index Terms—collision detection, Minkowski, support point, contact information, real-time, iterative, features

1 Introduction

A computationally efficient, straightforward and robust generic collision detection algorithm is essential in virtual environments. Collision detection is a diverse area (e.g., broad, narrow, convex, concave, discrete, and continuous). Typically, narrow phase is the most specific and most expensive collision process, since the intersection tests need to be exact while computing detailed contact data. We focus on convex rigid body bodies in discrete systems, particularly the support mapping concept and its application in real-time interactive solutions. The algorithm not only needs to correctly detect collisions but also generate accurate contact information, such as, penetration depth while handling numerical errors.

Due to the immense importance of collision detection algorithms, numerous research has been devoted to the problem in recent decades. This has produced numerous exciting and novel solutions, such as, GJK [1], V-Clip [10], Lin-Canny [9], and I-COLLIDE [3]. Each of these techniques have numerous advantages and disadvantages that evolve around the understanding of fundamental geometric principles. The algorithms can be implemented in three-dimensional space using vector algebra to derive information, such as, face normals, separating planes, support points, and distances. However, implementing an algorithm effectively without incorporating engineering solutions for special cases that is scalable and easy to understand can be challenging and time consuming.

We explain the concept of support mapping and its application in popular collision detection algorithms for generating accurate narrow phase contact information. We present experimental results to demonstrate scalability, practical pitfalls, memory, performance, and hybrid adaptations. We provide a simplified real-world implementation to illustrate the elegant compact nature of the algorithm.

2 Related Work

We show where support mapping sits in terms of other techniques for solving contact and collision problems (Figure 1). Explaining all the collision detection algorithms would be beyond the scope of this article, instead we show key algorithms that have been successful and relate to support mapping approach presented in this article.

A number of algorithms build upon the general support mapping and Minkowski Difference concept as we use in this article, such as, Gilbert-Johnson-Keerthi (GKJ) [5], XenoCollide [14], and Expanding Polytope Algorithm (EPA) [15]. However, we focus on the algorithms ability to solve practical problems through hybrid adaptations (i.e., look-up tables) and through exploitation of technological advancements, such as, the graphical processing unit (GPU). Cube-mapping has been used to speed-up the look-up method [12].

Support mapping is a popular technique that is supported in numerous, open source and commercial physics engines, such as, Bullet (GJK) Physics Engine [4], Open Dynamics Engine (ODE) [13], Havoc (GHK variant of GJK). For a comprehensive evaluation and comparison of physics engines and their differences, see Boeing and Bräunl [2], similarly, Kockara et al. [8] provides an overview of the different collision detection algorithms and differences.

3 Overview

We demonstrate key concepts that are not always made explicit with the algorithm and are crucial in practical high performance narrow phase implementations.

- ✓ closest points between pairs of convex polytopes (proximity data)
- ✓ contact information (e.g., normal, penetration, contact point), which includes handling deep penetrations
- ✓ degenerate convex shape with holes and concave sub-elements (i.e., shape vertices and 'not' the topology and geometric face information)
- ✓ efficient and can scale well to high poly meshes
- ✓ take advantage of coherency between frame updates
- ✓ implemented easily without much difficulty

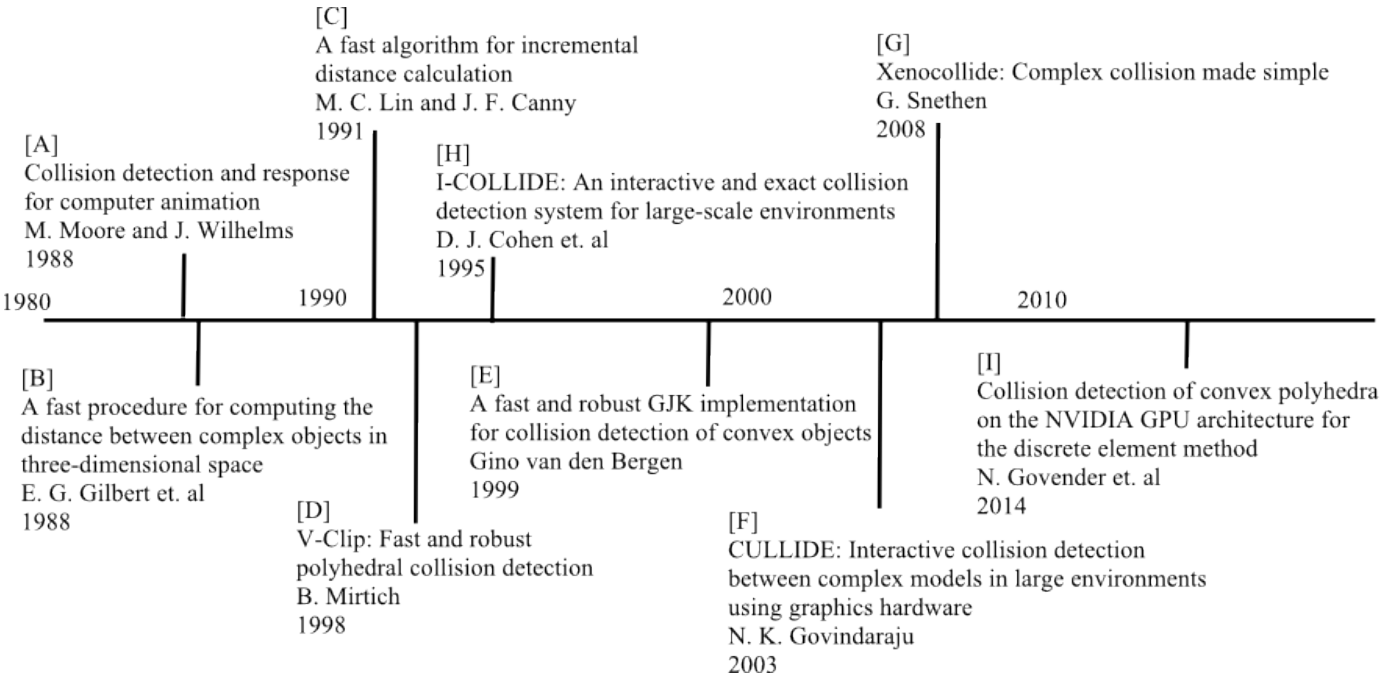

Fig. 1. **Timeline Overview** - Visual illustration of collision detection methods related to support mapping concepts. [A] [11], [B] [5], [C] [9], [D] [10], [E] [1], [F] [7], [G] [14], [H] [3], [I] [6].

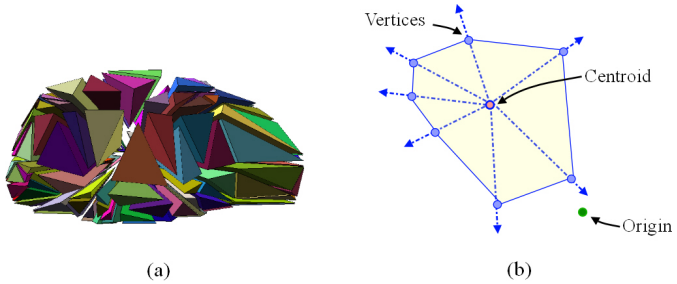

Fig. 2. **Tetrahedrons** - (a) A complex convex mesh can be subdivided into tetrahedron wedges using the centroid. (b) The origin has to lay within one of the open ended tetrahedrons. The calculation of which tetrahedron is at the heart of the algorithm.

- ✓ iterative solution that can provide both accurate and rough approximations (i.e., less iterations for a rough yet acceptable solution)
- ✓ the algorithm is able to exploit massively parallel architectures, such as, the graphical processing unit (GPU)
- ✓ regional clumping of support points for speed ups
- ✓ contact manifold generation (i.e., caching)

4 METHOD

4.1 Fundamental Concepts

Given a convex shape is decomposed of faces, with each face forming a plane. A point is inside the convex shape if the point is on the same side of all the faces (i.e., positive side of the face if all the faces are pointing inwards). Note, when checking all the faces, if we determine a face that fails this test results in a non-intersection, this face should be used as the first check in the following frame to exploit coherency. Due to the fact that objects only move by small amounts during each update. For example, if the shape is a thousand faces, and during the first check it was after 400 checks before determining a face the point is on the wrong side. However, in the subsequent check, if the objects have not moved, the check will be the first iteration and would not need to do the other 999 checks.

4.2 Support Mapping

Support Mapping return the farthest point in some direction. Hence, if we have a collection of points (i.e., the shape vertices), we are able to find the point furthest along a specified direction. The operation can be performed without any complex mathematical operations using the plane equation (i.e., essentially the dot product). Additionally, the direction does not need to be of unit-length, i.e., unit-normal for the calculations, since we are not interested in the distance only the further point.

4.3 Support Direction

When using the object centroid as a reference point, the support direction does not always return a point on the convex hull.

4.4 Minkowski Difference

The collision detection algorithm is from a concept called the Minkowski Sum. The Minkowski Sum is a straightforward concept that we combine with support mapping. For example, the Minkowski Sum shapes of two shapes is simply the addition of all the points in shapeA added to all the points in shapeB, as shown below in Equation 1:

$$A + B = \{a + b \quad | \quad a \in A, b \in B\} \tag{1}$$

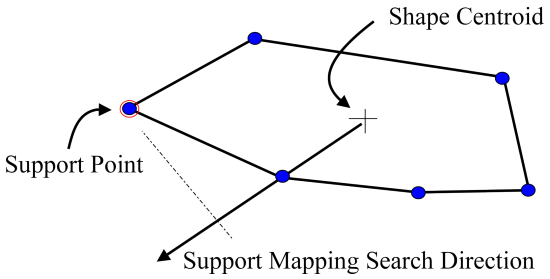

Fig. 3. **Support Mapping** - Two dimensional illustration showing proximity considerations when using the shape centroid to find support points. Importantly, as shown in the figure, the support direction from the centroid to a vertex does not necessarily return the vertex, but the the furthest vertex in the specified direction.

If both shapes are convex, the resulting shape is convex. However, the significance is not in the addition for collision detection, but in the subtraction, as shown below in Equation 2:

$$A - B = \{a - b \quad | \quad a \in A, b \in B\} \tag{2}$$

For clarity, we refer to the subtraction of the Minkowski Sum, as the Minkowski Difference. We use Support Mapping to optimise the generation of the Minkowski difference. Instead of subtracting every point on one shape from every other point on another shape to determine the collection of convex points for the Minkowski difference, we can use the geometric knowledge of convex shapes. The surface points in the resulting convex mesh must be support points (i.e., the furthest point on the resulting Minkowski Difference and opposite furthest distances when taken from both shapes).

4.5 Algorithm

The support mapping algorithm works by continually refining the point on the surface of the Minkowski surface. A basic implementation can be accomplished in two parts. For the first part, we calculate the centroid and work out a tetrahedron using basic geometric principles (i.e., dot and cross product) that contains the origin (see Figure 8). While the second part, iteratively refines the tetrahedron by recursively splitting and narrowing down on a surface plane that is as close to the origin as possible. Once we have identified the final tetrahedron, we can use Barycentric coordinates to work out the ratios and points on the original shape to calculate contact information. The pseudo code for the algorithm is shown below in Algorithm 4.5, while a simplified implementation is given in Listing 2.

The algorithm is elegant and straightforward in its unoptimised form for both two-dimensional and three-dimensional collision tests. The principle evolves around a good understanding of geometric principles and common mathematical operations, such as, the dot and cross product. When implementing and debugging the algorithm it is always worth while including engineering asserts to detect issues, such as, the origin not being within the projected tetrahedrons area (i.e., possible due to cross product normal or plane equation calculation distance being wrong but working for the basic test cases).

Algorithm 1 Support Mapping Algorithm for Calculating Contact Features.

```
1: Calculate Minkowski Centroid v0
2: Calculate open ended tetrahedron (v0, v1, v2, v3)
3: while Tetrahedron does not contain origin do
4:     Recalculate v1, v2, v3
5: end while
6: while Subdivide tetrahedron face until we cannot get any closer do
7:     Recalculate v1, v2, v3
8: end while
9: If the origin is on the inside of the tetrahedron outer face we have a collision
10: Barycentric coordinates for origin on the plane made up of v1, v2, v3 and map it to the original coordinates
```

4.6 Threading

While objects may not be colliding, we are still able to keep track of the closest features to exploit coherency. Hence, we can detect a missed collision after a few iterations, however, if we use a multi-threaded architecture, we can set a flag to identify a miss, while leaving the collision detection algorithm to continue searching for closest features to help with the following frame update (i.e., providing better starting approximation).

4.7 Graphical Processing Unit (GPU)

Initial misconceptions regarding the GPU is the straightforward porting of algorithms to exploit the massively parallel architecture. The GPU unlike the CPU is a single instruction multiple data (SIMD) architecture. What does this mean? It means that code with multiple conditional logic, such as, if statements, and while loops that perform differently for different data would be worse on the GPU. Essentially, GPU devote proportionally more transistors to arithmetic logic units and less to caches and flow control in comparison to CPUs. Typically, the GPU also has a higher memory bandwidth compared to a CPU (i.e., the GPU is ideal for parallel data computations with high arithmetic intensity).

Making the algorithm suitable for the GPU, we need ensure the code runs the same for each instance across the large number of cores. That is, the data will be unique for each collision detection pair, however, the implementation will run the same number of checks/calculations across all the cores. For example, if we have a dozen instances of the algorithm calculating the contact information, and one of the instances requires twenty iterations while the other eleven only require five iterations, all the instances will perform the twenty iterations. However, the algorithm is flexible enough to allow us to iterate over and over without affecting the final solution. Similarly, we can limit the iterations and provide a best guess solution, upon which in the following update we can use the previous frame as a starting approximation to help reduce the iteration search time.

- limit maximum iterations for searching
- feed-forward previous solution as the starting point for the next frame

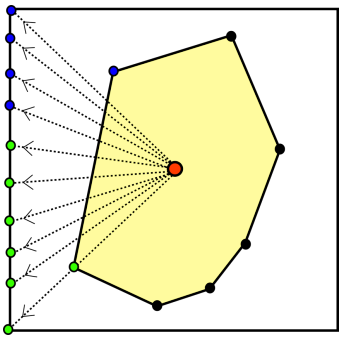

Fig. 4. **Cube-Map Lookup** - A similar analogy to the concept presented by Sathe and Lake [12] who generate a distance cube-map lookup for surface points, however, we modify the idea to storing support points. This avoids iterating over a complex mesh to find a support-point for a given direction. Hence, instead of storing a distance, we store the support point index for that vertex on the shape surface with the greatest distance given the specified direction.

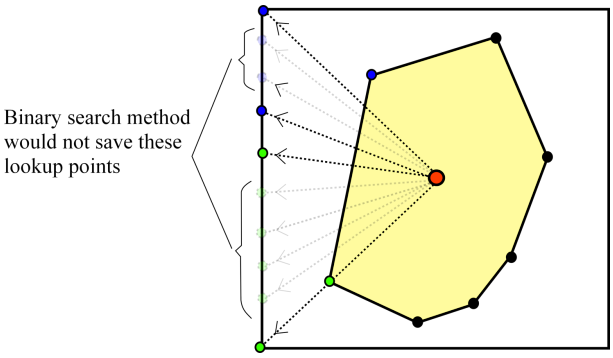

Fig. 5. **Binary Search** - The binary search organisation expands upon the lookup concept shown in Figure 4, however, the points are organised to distribute the resolution across the surface.

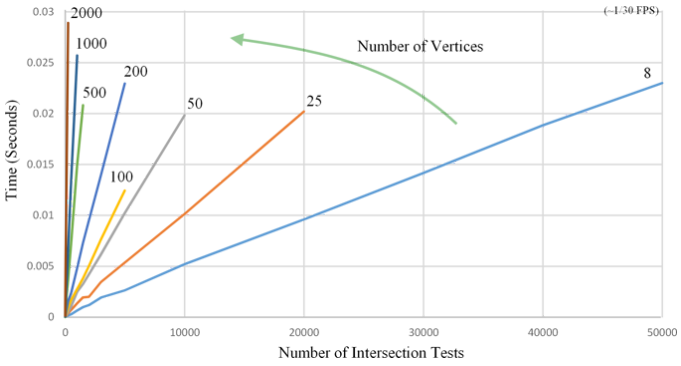

Fig. 6. **Brute Force - Number of Object-Object Collision Tests vs Time for Varying Numbers of Vertices** - Support mapping performance metrics illustrating the linear relationship between vertices and time.

4.8 Surface Projection

We decompose the mesh into regions to accelerate the support mapping search. For example, wrapping the shape with a sphere or cube that is split into equally sized regions and determining which points lay within the region (see Figure 4). When support point searching, we search the pre-computed surface points rather than iterating over all the vertices. After finding the closest direction, we have the support vertex for the mesh. Once approach is to use a 'lookup' method - such as, a cube-map arrangement (Figure 4). We also suggest another concept using a 'binary tree' arrangement. The binary tree arrangement is to help the distribution of points. For example, if we have clusters of points in particular regions, we can add more resolution, rather than linearly distributing over the whole shape area with a cube/sphere map lookup. However, this incurs a marginal cost compared to the instant lookup of a cube-map. Since we need to recursively subdivide the partitions into binary spaces. For instance, if we have 1024 segments, we would need to do 10 tests (i.e., 2^{10}) to reach target support point. This speed comparison is small compared to the brute force approach, however, it is worth mentioning as it can be important for complex meshes with high poly counts (e.g., graphical render meshes).

4.9 Binary Tree Organisation

Organising the shape support mapping using a binary tree requires extra pre-processing. Naively splitting the vertices into quadrants is incorrect (Figure 3). We use the cube-map concept, however, for regions where all the neighbours are identical, we remove the lookup points. For regions with different neighbours we subdivide and search the inner centre. Recursively, subdividing the surface quadrants until we reach a maximum limit or each quadrant contains corners of identical lookup indexes. Hence, the support mapping direction binary tree search allows us to very efficiently locate the correct vertex almost instantly (i.e., compared to iteratively searching over hundreds or thousands of vertices). While the binary tree look provides a more efficient organisation of the lookup data and scales better for non-linearly distributed surface meshes at the cost of more iterations to identify the surface quadrant.

5 Experimental Results

We perform a number of fundamental experiments to provide to the reader essential information about the algorithm, such as, performance, accuracy, and limitations. Preliminary baseline results, using general shapes, large numbers of simple shapes (e.g., cubes), large numbers of complex shapes (e.g., high poly graphical meshes), apply the solution to a dynamic situation (e.g., physics simulator). Secondary results, that use enhancements or modifications, such as, GPU/coherency, and surface projects.

Generating random convex shapes, we can plot an approximation of time verses the number of intersection tests for different shape complexities. As shown in Figure 6, this shows a linear relationship between vertices and intersection tests. Hence, we approximate the time for a specified number of vertices and intersections by Equation 3:

$$t = n\, c \tag{3}$$

where t is time, n is the number of vertices, and c is the number of intersection checks.

5.0.0.1 Level of Detail: This basic demonstration of the support mapping in action for deriving collision and contact information leads to a number of modifications.

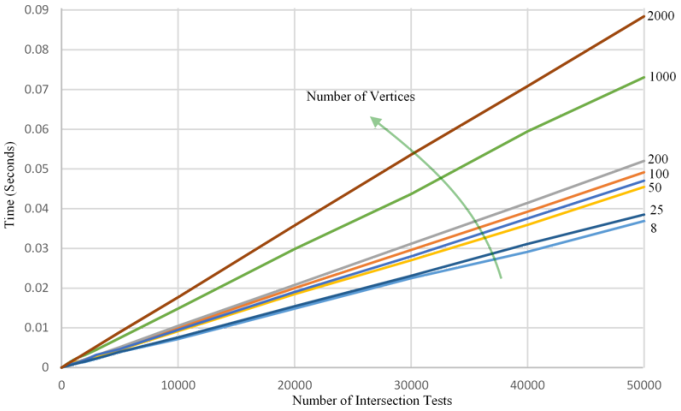

Fig. 7. **Binary Tree Support Mapping Optimisation - Number of Object-Object Collision Tests vs Time for Varying Numbers of Vertices** - Support mapping performance metrics illustrating the linear relationship between vertices and time.

For simple tests (e.g., cubes with 8 vertices), we are able to perform a large numbers of collision tests (e.g., 10,000), while remaining at real-time speeds (30+ fps). Combining a level of detail methodology to the algorithm would enable a basic speed-up enhancement. For example, performing a basic sphere-sphere check, followed by a cube-cube, then a low-poly an finally the higher poly-mesh.

5.0.0.2 Binary Tree: The computational slow-down of the support algorithm shown in Figure 6 for increased number of vertices is due to the 'linear' iterative search algorithm within the support function. For an uncomplicated implementation, the support point is found by iterating over 'all' the vertices. Hence, by quickly identifying individual groups of vertices within the shape, we can achieve a dramatic speed-up. Most importantly, this would break the coupled dependency between vertex counts and computational speed. This would modify the original Equation 3, changing 'n' to a constant based upon the search algorithm. Where a binary tree organisation enables us to exponentially cull large numbers of vertices with a minimal cost to identify groups of vertices for the support mapping tests.

5.0.0.3 Engineering Issues: The algorithm offers a stable solution for calculating reliable contact information even in the face of numerical limitations, such as floating-point round-off, which can cause many geometric algorithm implementations hang, crash, or produce nonsensical output if sufficient measures are not included. The proposed technique is adaptable to speed up calculations that do not always need to be exact, but must satisfy some error bound. The implementation of a generic algorithm, as we have presented here, can be combined with an algorithmic model for pre-defined shapes as done with Xenocollide [14] instead of depending on the level of detail of the collision mesh.

6 Conclusion

We have explained and demonstrated the well known support mapping concept for use in writing a general robust convex collision detection algorithm for narrow phase intersection tests. The algorithm is able to handle degenerate

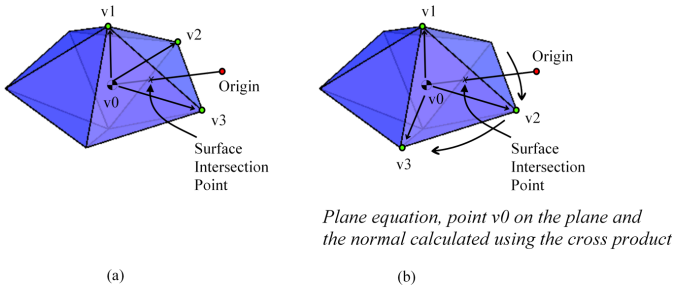

Fig. 8. **Phase One** - Phase one is to find tetrahedron for using the shape centroid that contains the origin (i.e., Minkowski shape).

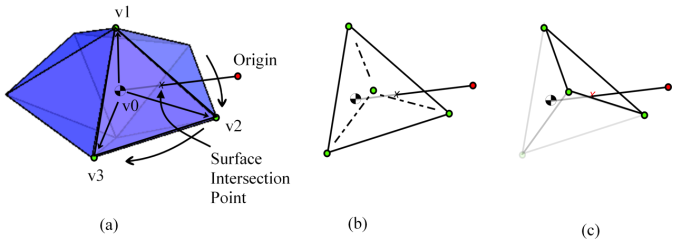

Fig. 9. **Phase Two** - Phase two is to refine the tetrahedron from by recursively subdividing the face (i.e., keep the shape centroid) until we are unable to subdivide any-more and then we have our answer.

shapes (i.e., point cloud shapes or concave meshes) to provide reliable feature information, such as, contact location and penetration depth. The algorithm on its own is simple to understand and implement. As we have shown in this paper, the algorithm can be combined with optimisation techniques, such as, look-up tables and binary trees to make the algorithm a viable solution even for large complex meshes. In addition, the algorithm is suited to massively parallel architectures to enable the calculation of narrow phase contact information for large numbers of objects.

```
struct SupportPoint
{
SupportPoint(const Vector3& pp, // support point
const Vector3& nn, // support normal
const Vector3& aa, // point on shapeA
const Vector3& bb) // point on shapeB
: p(pp), n(nn), a(aa), b(bb) {};
Vector3 p;
Vector3 n;
Vector3 a, b;
};

SupportPoint GetSupportVertex(const Shape* sA,
const Shape* sB, const Vector3& n)
{
Vector3 v0 = sA->GetSupportVertex( n );
Vector3 v1 = sB->GetSupportVertex( -n );
return SupportPoint(v0-v1, n, v0, v1);
}// End GetSupportVertex(..)
```

Listing 1. The minkowski difference (i.e., convex surface), is calculated using the opposite support directions. We need to store the support mapping points to recover proximity information at the end (e.g., contact position, penetration depth, and separating normal).

```
bool Collision(const Shape* shapeA,
const Shape* shapeB,
float&   outPenetrationDepth,
Vector3& outContactPoint,
Vector3& outContactNormal )
{
const Vector3 origin (0,0,0); // our target!
```

```
//
// **** PHASE 1 ****  see Figure 8
//
SupportPoint v0 = SupportPoint( shapeA->GetCentroid() - ↩
      shapeB->GetCentroid(),
Vector3(0,0,0), Vector3(0,0,0), Vector3(0,0,0) );

SupportPoint v1 = GetSupportVertex(shapeA, shapeB, origin-v0.p)↩
      ;
SupportPoint v2 = GetSupportVertex(shapeA, shapeB, Cross(origin↩
      -v0.p, origin-v1.p));

Vector3 dir3 = Cross(v1.p-v0.p, v2.p-v0.p);
if ( Dot( dir3, v0.p - origin ) > 0 ) dir3 = -dir3;
SupportPoint v3 = GetSupportVertex(shapeA, shapeB, dir3);

// v0, v1, v2 and v3 form a tetrahedron - however, we need to do
// a bit of looping around to 'ensure' the tetrahedron
// encloses the origin
while (true)
{
if ( Dot( Cross(v1.p-v0.p, v3.p-v0.p), v0.p-origin ) < 0 )
{
v2   = v3;
v3 = GetSupportVertex(shapeA, shapeB, Cross(v1.p-v0.p, v3.p-v0.↩
      p));
continue;
}

if ( Dot( Cross(v3.p-v0.p,v2.p-v0.p), v0.p-origin ) < 0 )
{
v1   = v3;
v3 = GetSupportVertex(shapeA, shapeB, Cross(v3.p-v0.p, v2.p-v0.↩
      p));
continue;
}
break;
}

//
// ****** PHASE 2 ******   see Figure 9
//
while ( true )
{
// Find support point using the tetrahedron face
SupportPoint v4 = GetSupportVertex(shapeA, shapeB, Cross(v2.p -↩
      v1.p, v3.p - v1.p));

// Is our new point already on the plane of our
// triangle, we're already as close as we can get to the target
float delta = Dot( (v4.p - v3.p), v4.n );
if ( abs(delta) < 0.001f )
{
Vector3 n = Normalize( v4.n );
// Compute distance from origin to the wedge face
outPenetrationDepth = Dot(n, v1.p-origin);
break;
}

// We'll create three baby tetrahedrons and decide
// which one will replace the current tetrahedron
// v1v2 - v2v3 - v3v1 - current tetrahedron edges
// leads to 3 new tetrahedrons:
// v1v2v4
// v2v3v4
// v3v1v4
while ( true )
{
Vector3 na = -Cross( v2.p-v0.p, v4.p-v0.p );
Vector3 nb = -Cross( v4.p-v0.p, v1.p-v0.p );
if ( Dot( na, v0.p-origin ) > 0 &&
Dot( nb, v0.p-origin ) > 0 )
{ // Inside this tetrahedron
v3   = v4;
break;
}

na = -Cross( v3.p-v0.p, v4.p-v0.p );
nb = -Cross( v4.p-v0.p, v2.p-v0.p );
if ( Dot( na, v0.p-origin ) > 0 &&
Dot( nb, v0.p-origin ) > 0 )
{ // Inside this tetrahedron
v1 = v4;
break;
}

na = -Cross( v1.p-v0.p, v4.p-v0.p );
nb = -Cross( v4.p-v0.p, v3.p-v0.p );
if ( Dot( na, v0.p-origin ) > 0 &&
Dot( nb, v0.p-origin ) > 0 )
{ // Inside this tetrahedron
v2 = v4;
break;
}
// Should never get here - as it must be in
// one of the children!
DBG_ASSERT(false);
break;
}

}// End while (true)

// Barycentric coordinates to map from the minkowski
// difference onto the original shape
outContactPoint = MapPointOrigin( v1.p, v2.p, v3.p,
v1.a, v2.a, v3.a );

if ( outPenetrationDepth < 0 ) return false;  // No Hit

return true; // Hit
}// End Collision
```

Listing 2. An example implementation of the support mapping algorithm for finding the closest point between two convex shapes, in addition to the penetration depth, contact normal, and contact point. The uncomplicated nature of the algorithm means we can implement a working model without difficulty that is able to provide robust contact information for programs, such as, physics simulators.

REFERENCES

[1] Gino van den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *Journal of graphics tools*, 4(2):7–25, 1999. 1, 2

[2] Adrian Boeing and Thomas Bräunl. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 281–288. ACM, 2007. 1

[3] Jonathan D Cohen, Ming C Lin, Dinesh Manocha, and Madhav Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–195. ACM, 1995. 1, 2

[4] Erwin Coumans. Bullet physics engine. *Open Source Software: http://bulletphysics. org*, 2010. 1

[5] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *Robotics and Automation, IEEE Journal of*, 4(2):193–203, 1988. 1, 2

[6] Nicolin Govender, Daniel N Wilke, and Schalk Kok. Collision detection of convex polyhedra on the nvidia gpu architecture for the discrete element method. *Applied Mathematics and Computation*, 2014. 2

[7] Naga K Govindaraju, Stephane Redon, Ming C Lin, and Dinesh Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 25–32. Eurographics Association, 2003. 2

[8] Sinan Kockara, Tansel Halic, K Iqbal, Coskun Bayrak, and Richard Rowe. Collision detection: A survey. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 4046–4051. IEEE, 2007. 1

[9] Ming C Lin and John F Canny. A fast algorithm for incremental distance calculation. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1008–1014. IEEE, 1991. 1, 2

[10] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics (TOG)*, 17(3):177–208, 1998. 1, 2

[11] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. *ACM Siggraph Computer Graphics*, 22(4):289–298, 1988. 2

[12] Rahul Sathe and Adam Lake. Rigid body collision detection on the gpu. In *ACM SIGGRAPH 2006 Research posters*, page 49. ACM, 2006. 1, 4

[13] Russell Smith et al. Open dynamics engine. 2005. 1

[14] Gary Snethen. Xenocollide: Complex collision made simple. *Game Programming Gems*, 7:165–178, 2008. 1, 2, 5

[15] Gino Van Den Bergen and Gino Johannes Apolonia van den Bergen. *Collision detection in interactive 3D environments*. Elsevier, 2004. 1