Workshop Series: Collision Detection

Benjamin Kenwright[1]*

Abstract
This practical focuses on collision detection algorithms and contact data. The student needs to implement a set of real-time interactive collision detection algorithms (e.g., sphere-sphere, sphere-plane, and ray-triangle).

Keywords
Collision Detection, Rigid Body Objects, Particles, Constraints, Classical Mechanics

[1] *Workshop Series (www.xbdev.net) - Benjamin Kenwright*

Contents

Introduction

Collision Detection The topic of this practical is the implementation of a real-time collision detection simulation (i.e., interaction of objects in a virtual environments). The user should be able to interact with the simulation while it is running (e.g., through the mouse or keyboard) to control the objects - such as, moving objects around so that they interact and visually display collision/contact information. The collision detection algorithms should be implemented using component programming principles (i.e., flexible set of library functions that can be integrated into a physics-based simulation).

Tasks
- Visually display collision and contact information
- User input (e.g., mouse or keyboard) to control and move the objects around
- Develop a variety of different collision algorithms (e.g., sphere-sphere, plane-sphere, ray-triangle ..)

o Sphere-Sphere o Sphere-Plane o Ray-Plane o Separating Axis

1. Overview

Principles and Concepts A robust and computationally efficient collision detection framework is indispensable in virtual environments. Without a collision detection framework objects or even the user wouldn't be able to interact with objects or the environments (i.e., objects would pass through one another like ghosts). In this practical, we discussed the concept of splitting the collision detection algorithms into a high level broad-phase and the lower level narrow-phase. We examine the algorithms typically used during narrow-phase, since they are indispensable for a physics-base system, and introduce the broad-phase techniques in later practicals as computational speed becomes an issue. For narrow-phase collision systems, we turn our attention to a series of algorithms for complex collision tests, which can be employed for detailed shapes at a polygonal level. As ever, the more complex and accurate the algorithm, the more computationally expensive it is, so care needs to be taken that these routines are only applied in cases where they are of benefit. The broad-phase methods which we discuss later, will be employed later to cull almost all potential collisions, so the remaining list of possible intersections is sufficiently small to justify these more detailed tests.

Collisions & Contacts In this practical, we discuss and implement various collision detection algorithms. As we are able to move objects around the three-dimensional simulated world in a believable manner; however, if two of our simulated objects meet, they just pass through one another like ghosts - i.e. although they move through free space like believable physical entities, they have no physical presence so they do not bounce off one another, or come to rest one alongside the other. The main features of a collision detection framework:
- Detect if a collision has occurred (i.e., true/false)
- Identify precise contact information (e.g., penetration

depth, contact normal, and contact points)

Essentials The first step toward adding this physical presence is to detect when two objects have collided. For each frame of the simulation, we calculate the position of our simulated objects - if the new position of two of our objects causes an intersection of those objects, then we need to push them back apart. Later practicals will deal with how we actually push them apart (i.e., collision resolution); for now we will concentrate on identifying when such an intersection has occurred, and how we calculate the data required to believably move two colliding bodies apart.

2. Collision Detection Algorithms

Principles and Concepts This section examines a series of increasingly complex collision detection algorithms for use during narrow-phase. There are no hard and fast rules for when each test should be employed; the factors of efficiency, and player immersion (i.e. how important the accuracy of a particular collision check is to appear realistic to the player) must be accounted for. For clarity and understanding the diagrams are presented in two dimensional form; however, the principles work for both 2D and 3D.

It's straightforward to detect a collision. For example, we can 'detect' when a sphere has collided with an infinite plane. However, the collision response requires additional information (i.e., more than a true or false) that includes the position of the contact point on the plane, penetration depth, and contact normal. In simulating a virtual environment, we clearly need to be able to represent finite planes, and test for collisions against them. A infinite plane, such as a wall or roof, can be represented for the purposes of the physics engine as two or more triangles. Consequently, we need an algorithm to test whether our collision point on the infinite plane (which the finite plane is part of) is within any of the triangles which make up the finite plane.

There are well-known algorithms for testing whether a point is inside a triangle - in fact, these algorithms are applicable to any two-dimensional convex shape. We will first consider an algorithm for testing against a generic convex shape, and then look at the more complex case of concave shapes. Remember that a convex shape has no interior angles greater than 180 degrees, whereas a concave shape has at least one interior angle greater than 180 degrees. Examples of convex shapes are a triangle, a rectangle, a circle, a slice of pizza; examples of a concave shape are a star, a crescent moon, a pizza with a slice missing.

3. Convex Shape Intersection

We require a test for whether a point is within a convex shape. The diagram shows a shape consisting of five connected points, with point A outside the shape, and point B inside the shape.

There are some intuitive tests based on drawing a line from the test point to each of the vertices, and checking the size of angles between those lines. If all of those angles are less than π radians then the point must be inside a convex polygon. Alternatively if the sum of all those angles is equal to 2π radians then the point is inside the convex polygon. However, if we were to proceed with this method, we do not have the angle data readily to hand so it must be calculated - this would involve calculating the arc-cosine of the dot product, which is a very expensive thing to do computationally. It is either very slow to calculate (compared to standard operations such as addition and multiply), or it would require a large look-up table which would be both heavy on memory and imprecise. Consequently we require a more clever method.

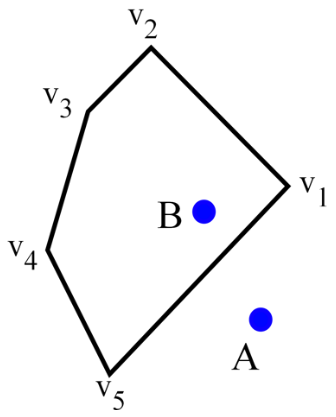

Figure 1. Point Inside - Determining if a point is inside or outside a convex shape.

The "clever" methods are based on the idea that a point inside the polygon is consistently on the inside of each side of the polygon. They take each edge of the polygon in turn and calculate which side of that edge the test point is on. As soon as the point is found to be on the outside of an edge then it must be outside the polygon as a whole and the test is failed. If the test point is found to be on the inside of all edges, then it must be inside the polygon and an intersection has been successfully identified. Note that this is only true for convex polygons.

3.1 Cross Product Method

The first method we will look at involves using the cross product to test whether the point is on the same side of a polygon edge as another vertex of the polygon.

Consider the diagram, which shows the various vectors used in testing whether points A and B are on the inside or outside of the edge connecting vertex V_1 to vertex V_5. The base vector from which we will perform the cross product tests is that connecting V_5 to V_1; this vector is $(V_5 - V_1)$. We are interested in the cross product of this vector and the vector connecting the test point to V_1. In the case of test point A that is $(A - V_1)$, and for B that is $(B - V_1)$.

The vector resulting from this cross product will either point out of the paper, or into the paper (as it must be orthonormal to the two vectors). It should be clear that the cross

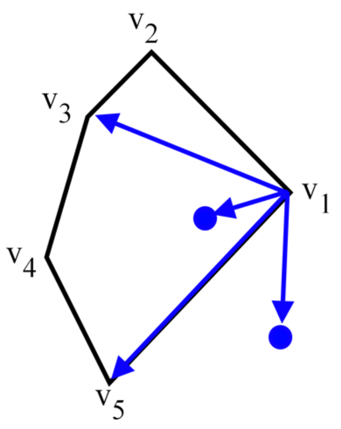

Figure 2. **Cross Product** - Determining if a point is inside or outside a convex shape using the cross product method.

product $(V_5 - V_1) \times (B - V_1)$ will be in the opposite direction to the cross product $(V_5 - V_1) \times (A - V_1)$. So we can distinguish between a point outside the edge, and a point inside the edge, based on the sign of this cross product. However we need to know which sign (positive or negative) results from the point on the inside.

This can be achieved by also testing where one of the other vertices in the polygon resides with respect to the edge. In the diagram we also consider vertex V_3. We know that any vertex of a convex polygon must be on the same side of an edge as the whole polygon (i.e. all possible test points that lie within the polygon). So the cross product $(V_5 - V_1) \times (V_3 - V_1)$ will have the same sign as the test point within the polygon.

This test must be repeated for each edge of the polygon, choosing a random other vertex as the basis for whether the cross product must be negative or positive. As soon as an edge test fails, the point must be outside of the polygon as a whole and the rest of the tests can be abandoned. A big advantage of this method is that the ordering of the vertices does not matter, since the third vertex is used as a control.

3.2 Plane Equation Method

This algorithm exploits the popular plane equation by generating a normal for each side of the polygon, and determining where the test point lies along those normals. If the point is in the negative direction along the normal to one of the sides, then it must be situated outside the polygon. In other words, if the point lies in the positive direction of all edge-normals then the point must be inside the polygon and an intersection has occurred. In order for this test to be meaningful, the ordering of the vertices of the polygon must be correct, so that a negative value of the normal is on the outside of the polygon.

In the example shown, it can be seen that both points A and B lie in the positive direction of the normal for the line between V_2 and V_3. However, point A is in the negative direction of the normal of the line between V_5 and V_1, whereas point B is in the positive direction. Consequently point A fails the test for intersection, whereas point B passes.

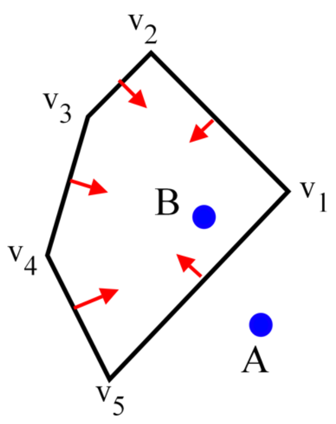

Figure 3. **Plane Equation Point Inside (Convex Shapes)** - Determining if a point is inside or outside a convex shape using the plane equation method.

The C++ code for implementing this algorithm is:

```
bool PointInConvexPolygon ( const Vector3 & TestPosition ,
                  Vector3 * convexShapePoints ,
                  int numPointsL ) const
{
  // Check if our test point is inside our convex shape
  for ( int i =0; i< numPoints ; ++i)
  {
    const int i0 = i;
    const int i1 = (i +1)% numPoints ;

    const Vector3 & p0 = convexShapePoints [ i0 ];
    const Vector3 & p1 = convexShapePoints [ i1 ];

    // We need two things for each edge , a point on the edge ,
    // and the normal
    const Vector3 n = Cross ( Vector3 (0 ,0 ,1) , Normalize ( p0 -p1 ) );

    // Use the plane equation to calculate d, and determine if our
    // point is on the positive or negative side of the plane ( edge )
    const float d = Dot ( n, p0 );

    // Calculate which side our test point is on
    // +ve for inside , -ve for outside , zero on plane
    const float s = d - Dot ( n, TestPosition );

    if ( s < 0.0 f )
    {
      // failed , so skip rest of the tests
      return false ;
    }
  }
  return true ;
}
```

This is a computationally efficient method for testing whether a point is within a convex shape. Of course, not all shapes in a game environment will be convex, so an approach is required for these more complex shapes. One option is to break down a concave shape into a number of convex shapes - if the test point is within any one of those component shape then it must be within the overall shape. The natural progression of such an approach is to break any shape down to its component triangles (which may well match the triangles used in building the object graphically, so the vertex data may already be available). A triangle is the most simple convex shape, so the test is applicable.

4. Concave Shape Intersection Part 1

There are more generic algorithms which can be used to test whether a point lies within any shape, whether convex or concave. Such algorithms are necessarily more computationally expensive, and so should be utilised with care.

The algorithm which we will implement involves imagining a line starting at the test point and ending a long way from the polygonal shape. If that line crosses any edge of the polygon an odd number of times, then the point must be inside the polygon; if it crosses any edge an even number of times then the point must lie outside the polygon. The diagram shows a number of test points in or near a concave shape, as well as the lines leading from the test point. The lines shown are horizontal along the x-axis, but any line will surface as long as its furthest point is sufficiently far from the test shape to guarantee that it is outside the shape.

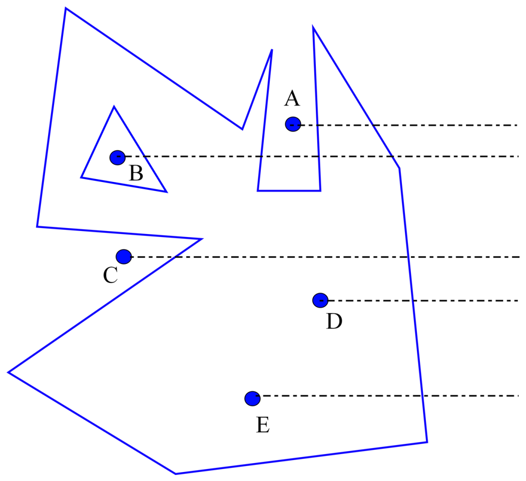

Figure 4. Concave Shape - Determining if a point is inside or outside a concave shape.

It can be seen that the lines leading from points *A* and *C* cross the polygon edges an odd number of times, whereas the lines leading from points *B*, *D* and *E* cross the polygon edges an even number of times. Further note that this method works for shapes which have holes; point *D* is in a hole in the shape and therefore not intersecting with it, and the test shows that the line from point *D* crosses the edges twice resulting in it being correctly identified as outside the shape.

In order to implement this algorithm, we will need a method of testing whether two lines intersect.

5. Line-Line Intersection

The equation describing the line a from point P_1 to point P_2 can be written as

$$P_a = P_1 + U_a(P_2 - P_1) \tag{1}$$

where P_a is any point along the line, and U_a is a factor of how far along the line the point resides. At the start of the line U_a is zero, so the point on the line is equal to P_1; at the end of the line U_a is one, so the point on the line is at P_2. Similarly the point half way along the line can be found by setting U_a to 0.5 and so on. It should therefore be clear that any point on the line will correspond to a U_a value between 0 and 1 - any value beyond these limits will result in a point not on the line between the two end points. This is the fact which we will exploit in our algorithm for testing for line-line intersections.

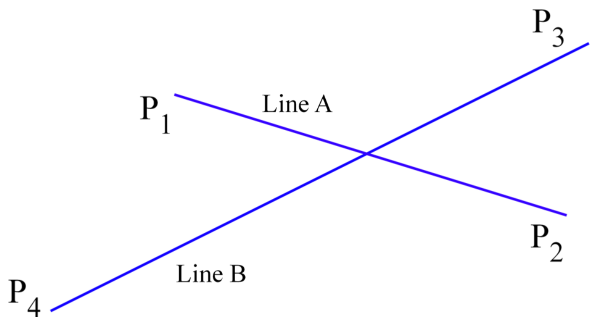

Figure 5. Line-Line Intersection - Determining if two 3D lines intersect.

$$\begin{aligned} P_a &= P_1 + U_a(P_2 - P_1) \\ P_b &= P_3 + U_b(P_4 - P_3) \end{aligned} \tag{2}$$

and we are testing for a point where the two lines cross, i.e. a point at which P_a and P_b are equal. So we are searching for the values of U_a and U_b for which the following equation is true:

$$P_1 + U_a(P_2 - P_1) = P_3 + U_b(P_4 - P_3) \tag{3}$$

Hence, we solve for the unknowns U_a and U_b.

For a two dimensional case, we expand this equation to a separate equation for the *x* and *y* values:

$$\begin{aligned} x_1 + U_a(x_2 - x_1) &= x_3 + U_b(x_4 - x_3) \\ y_1 + U_a(y_2 - y_1) &= y_3 + U_b(y_4 - y_3) \end{aligned} \tag{4}$$

and solving for the unknowns U_a and U_b.

Remember that these values U_a and U_b give the distance along each line at which the intersection occurs, measured as a factor of the total length of the line. Consequently, if the value of U_a is between 0 and 1, then the intersection occurs on line a between P_0 and P_1, and similarly for U_b.

Any two infinitely long lines will intersect at some point, unless they are parallel. For two parallel lines the value of the denominator in the expressions for U_a and U_b is zero. Note

that the denominator is the same for both expressions, and also recall that dividing by zero will result in a run-time error, so this test must be built into the algorithm.
Hence, the two lines intersect if all three of these conditions are met:

- $0 \leq U_a \leq 1$
- $0 \leq U_b \leq 1$

and the point of intersection can be found by plugging the values of U_a and Ub back into the original line equations for P_a and P_b.

In C++ the line-line intersection test is coded as:

Listing 1. Line-Line Intersection

```
class Line_c
{
 public :
 Line_c ( const Vector3 & p0 , const Vector3 & p1)
 { m_p0 =p0; m_p1 =p1; }
 Vector3 m_p0 ;
 Vector3 m_p1 ;
};

bool LineLineIntersection ( const Line_c & l0 ,
              const Line_c & l1 ,
              float * t0 = NULL , float * t1 = NULL )
{
 const Vector3 & p0 = l0. m_p0 ;
 const Vector3 & p1 = l0. m_p1 ;
 const Vector3 & p2 = l1. m_p0 ;
 const Vector3 & p3 = l1. m_p1 ;

 const float div = (p3.y−p2.y)*( p1.x−p0.x)
      − (p3.x−p2.x)*( p1.y−p0.y);

 // Nearly parallel lines
 if ( abs (div ) < 0.000001 f )
 {
  return false ;
 }

 const float ta = ( (p3.x−p2.x)*( p0.y−p2.y)
      − (p3.y−p2.y)*( p0.x−p2.x) ) / div ;
 if (ta <0 || ta >1.0 f)
 {
  return false ;
 }

 const float tb = ( (p1.x−p0.x)*( p0.y−p2.y)
      − (p1.y−p0.y)*( p0.x−p2.x) ) / div ;
 if (tb <0 || tb >1.0 f)
 {
  return false ;
 }

 if (t0) (* t0 )= ta;
 if (t1) (* t1 )= tb;

 return true ;
}
```

6. Concave Shape Intersection Part 2

So now that we have our algorithm for testing whether two finite lines intersect, we can easily implement the test for whether a point lies within any polygon, whether concave or convex. You will recall that the test involves taking a line from the test point to a point guaranteed to be outside the polygon, and counting the number of times it intersects with the polygon's sides. As we have vertex details for the polygon, we know the positions of the start and end points of the polygon edges, so the test in C is written as:

Listing 2. Concave Polygon Point Inside Test

```
bool InsideConcaveShape( const Vector3 * shapePoints ,
       const int numPoints ,
       const Vector3 & testPoint )
{
 int intersectionCount = 0;
 // Count how many times we cross the line
 for ( int i =0; i< numPoints ; ++i)
 {
  const int i0 = i;
  const int i1 = (i +1)% numPoints ;

  const Vector3 & p0 = shapePoints [ i0 ];
  const Vector3 & p1 = shapePoints [ i1 ];

  bool intersect = LineLineIntersection ( Line_c (p0 , p1),
  Line_c ( testPoint , testPoint + Vector3 (1000 ,1000 ,0)) );

  if ( intersect )
  {
   intersectionCount ++;
  }
 }

 // Even number of intersections means false
 if ( intersectionCount % 2 == 0 )
 {
  return false ;
 }
 // We are inside the shape return true ;
}
```

7. Implementation

The aim of this practical session is to expand your collision detection library and detect intersections between a variety of diverse geometric shapes. We construct a set of algorithms which carry out the required tests for polygonal collisions. To demonstrate that the collision tests are working, we will add functionality to the practical which tests for when a collision has occurred and displays the contact information.

8. Summary

In this practical, we introduce the concept of narrow and broad collision detection. We explain how to detect accurate collision information for complex polygonal models. The collision detection routines are two-tiered - narrow-phase and broad-phase. This practical focuses on the more computationally expensive collision detection routines which are typically employed during the narrow-phase. The narrow-phase method is indispensable and provides precise and reliable collision and contact information to the physics-based simulator.

9. Exercises

This practical only gives a brief taste of collision detection principles. As an exercise for the student to help enhance their understanding:

Intermediate

- Implement a collision detection/contact information framework (e.g,. sphere-sphere, sphere-ray, sphere-plane) and visually display the contact information
- Implement a collision detection method for the mouse cursor to select objects in the scene (i.e., 2D near and far ray in 3D space)

Acknowledgements

We would like to thank all the students for taking time out of their busy schedules to provide valuable and constructive feedback to make this practical more concise, informative, and correct. However, we would be pleased to hear your views on the following:

- Is the practical clear to follow?
- Are the examples and tasks achievable?
- Do you understand the objects?
- Did we missed anything?
- Any surprises?

The practicals provide a basic introduction for getting started with physics-based animation effects. So if you can provide any advice, tips, or hints during from your own exploration of simulation development, that you think would be indispensable for a student's learning and understanding, please don't hesitate to contact us so that we can make amendments and incorporate them into future practicals.

Recommended Reading

Real-Time Collision Detection , Christer Ericson, Publisher: CRC Press, ISBN: 978-1558607323

Code Complete: A Practical Handbook of Software Construction, Steve McConnell, ISBN: 978-0735619678

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

Game Inverse Kinematics: A Practical Introduction (2nd Edition) Kenwright. ISBN: 979-8670628204

Kinematics and Dynamics Paperback. Kenwright. ISBN: 978-1539595496

Game Collision Detection: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1511964104

Game C++ Programming: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1516838165

Computational Game Dynamics: Principles and Practice (Paperback). Kenwright. ISBN: 978-1501018398

Game Physics: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1471033971

Game Animation Techniques: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1523210688