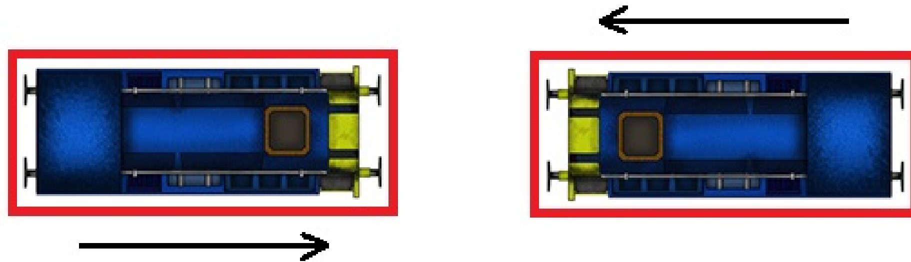
Physics Tutorial 4: Collision Detection



Summary

Collision detection is explicitly introduced, and explored in the context of physics simulation. We discuss the two-tiered approach which is typically taken in collision detection for games, and implement the basic set of collision detection routines which achieve this. We proceed to explore more advanced approaches to collision detection between complex objects.

New Concepts

Collision detection concepts, broad phase and narrow phase, convex and concave shapes, generic collision detection algorithms, Separating Axis Theorem.

Introduction

The physics engine which we are developing consists of three major sections:

- Simulate the motion of bodies in the world.
- Determine if it is possible for two objects to have collided or intersected (Broad Phase)
- Detect when two bodies have collided or intersected and extract information about those collisions (Narrow Phase).
- Compute a resolution to any intersections.

In this tutorial we consider the second section. We have discussed and implemented the algorithms for moving objects around the three-dimensional simulated world in a believable manner, and we are aware of the concept of constraints and interfaces, but we have yet to explore how we address the issue explicitly. Put simply, 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 first step toward adding this physical presence is to detect when two objects have collided. And the first step towards *detecting* that is determining whether or not it's possible for two objects to have collided. This is normally handled through computationally inexpensive checks, referred to as the Broad Phase of collision detection.

These inexpensive checks, however, were the original building blocks of all real-time physics simulation. For this reason, we considered these algorithms in some detail in a previous tutorial. In this tutorial, we begin by exploring the concept of broad phase and narrow phase separation. Advanced methods of collision detection, based on the intersection of convex objects, are then explored in detail.

Organising Collision Detection – Broad Phase and Narrow Phase

The goal of the collision detection section of the physics engine is to identify which objects in our simulation have intersected, or interfaced, so that we can push them apart. The sledgehammer approach to this would be test every simulated object in the world against every other simulated object. This is a N^2 problem and obviously it quickly becomes impractical for even a relatively small game world, particularly if we're using complex collision checks.

The collision detection routines will be much more efficient if we can structure the code in a way which ensures that the more complex algorithms are only applied to pairs of objects which are likely to have intersected. As already highlighted, a physics engine will typically divide the collision detection mechanism into two phases:

- **Broad Phase.** Identify which pairs of objects need to be tested for intersection.
- **Narrow Phase.** Carry out collision tests for each pair of objects identified in step 1.

We will discuss each of these phases, before moving on to the specifics of the detection algorithms that can be employed as an element of both phases. It is important to stress the point that checks notionally considered 'broad phase' can still be applied to all entities in a region to dismiss potential collisions. It is perhaps better to think of the process as hierarchical, balancing the cost of verifying a collision as possible against the cost of checking the collision with a more costly algorithm.

The sphere-sphere algorithm, for example, can be as appropriate in broad phase as narrow phase - as in all things, it is relative. If your narrow phase collision detection relates to many vertices of a highly complex object, then virtualising a sphere around an object in the broad phase to determine to a finer degree of accuracy whether or not you need to make that comparison is a worthwhile endeavour. In this case, a true sphere-sphere check to refine the results of one of the algorithms described below is a very useful saving, even if it comes as an intermediate step between some form of spatial partitioning and a convex hull intersection check.

Broad Phase

Imagine that the collision detection loop starts with a long list of paired objects, consisting of all N^2 possible pairings in the simulated world. The purpose of the broad phase is to provide a very quick culling procedure in order to "throw out" as many of these pairings as possible, before moving on to the more complex algorithms during narrow phase. It is analogous to the frustum culling and other techniques that were introduced during the Graphics course to reduce the number of graphical objects submitted for rendering.

Frustum culling may initially appear to be a good option for use in broad phase, especially as the work has already been done for the rendering loop. However there are some pretty big disadvantages to this approach, as we would not be testing for collision between any objects out of view – for example, if the player turns around there could well be a big mess of intersecting objects lurking behind him.

A simple but effective approach is to carry out quick bounding box tests between all object pairs (or sphere or capsule checks, depending on situation). This method culls any object pairs which have no chance of their extremities being in contact. The algorithm and implementation of this approach are discussed in detail in a later section of this tutorial.

While this is a computationally cheap test per object pair, it still must be carried out for every possible pairing (i.e. it is still N^2). A far more efficient routine would involve grouping nearby objects together is some way, so that entire groups of object pairs can be culled from the list at once, before applying cheap checks to those smaller groups.

BSP Trees and World Partition

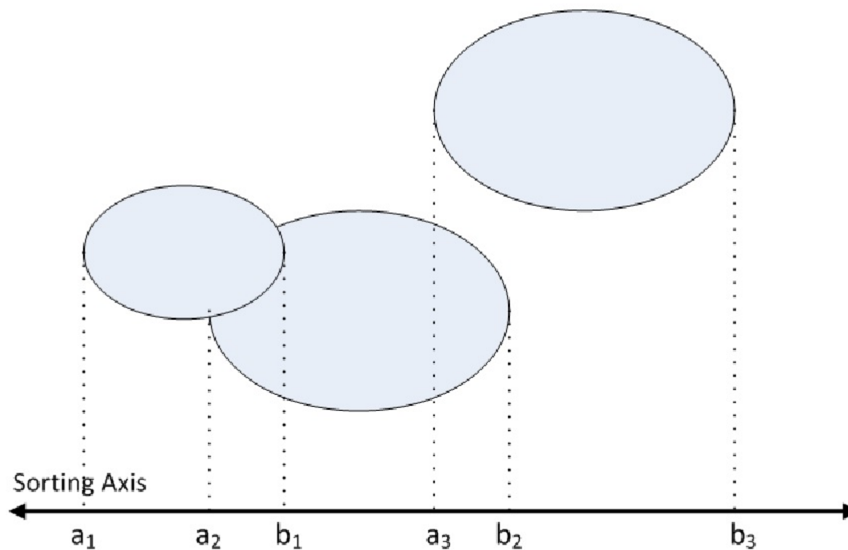
There are various ways in which the objects can be grouped to facilitate this faster culling during broad phase. In general, the techniques involve dividing the game world into a number of sections, and identifying which section each object belongs to. The assumption is that any object can only

be colliding with another object that is in the same section. The bounding box test then just needs to be carried out between objects in the same section. The problem then becomes a series of much smaller N^2 tests, where N is now the number of objects in each group rather than the entire world. It should be noted that any particular object can be in more than one world section at the same time; this is perfectly acceptable and the object just needs to be included in the loop for each section that it encroaches on.

The main technique for achieving this goal is Binary Search Partitioning (BSP) which recursively sub-divides the world into convex sections. The *Octree* and *Quadtree* are commonly used examples of BSP trees. The octree approach splits the world into progressively smaller cubes (i.e. it divides each cube into eight equally sized smaller cubes – hence the name octree). If a cube contains more than a threshold number of objects, then it is split down into eight further cubes, and so on recursively. The quadtree, as the name suggests, carries out the same process but in two dimensions only, so each square is split into four smaller squares.

Sort and Sweep

A further step in reducing the number of more expensive object pair tests is to implement a *Sort and Sweep* algorithm. The bounding volume of each object is projected onto a single axis. If the bounding extents of a particular pair of objects do not overlap then they can not possibly have collided, so that pair is discarded from the list of possible intersecting pairs.



This is achieved by constructing a list of all bounding markers (i.e. two per object, leading to a list of $2n$ items) along a particular axis. The list is then sorted and traversed. Whenever the start of a bounding volume is found (i.e. an a value in the diagram), that object is added to the active list, when the corresponding end of the bounding value is found (i.e. a b value in the diagram) it is removed from the list. When an object is added to the active list, any other objects on the list are potential collision candidates, so an object pair is created for the new object and each of the currently active objects.

In the example diagram shown, the sort and sweep algorithm creates a possible collision pair for the first and second object, and another for the second and third object, but not for the first and third object. So, when the more computationally expensive narrow phase routines cycle through the possible collisions, only two of the pairs are considered. Obviously expanding this method to a larger area with many more objects can greatly increase the number of potentially intersecting pairs which are culled during broad phase.

Narrow Phase

The list of N^2 object pairs that must be checked for intersection should have been greatly reduced by the broad phase algorithms, leaving a list of object pairs which we believe have a higher chance

of having collided. The narrow phase algorithms test these individual pairs in more detail, so as to identify which objects have actually intersected and need to be moved apart.

The narrow phase can operate at various different levels of detail, depending on the circumstances, and lower levels of detail often blur into the broad phase. In many cases a test of simplified collision shapes will be ample, but in some cases the algorithms need to get down to the polygonal level. Previous material focused on describing and implementing the basic algorithms which can be used, either for a simple narrow phase test, or during the broad phase. This tutorial describes the more complex algorithms used during narrow phase for polygonal level collision tests.

Collision Data

Collision Shape Data

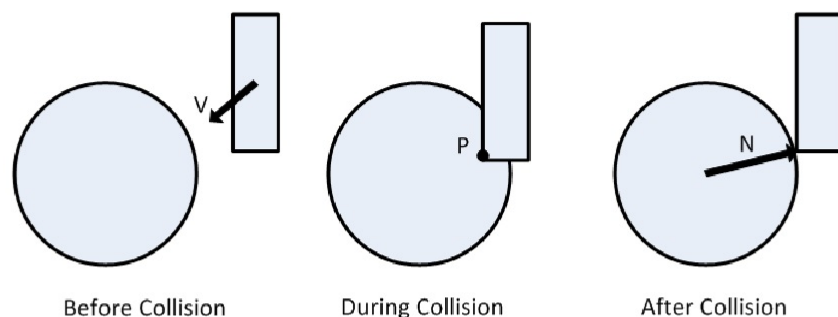
In the first tutorial in this series we discussed the necessity for differentiating between the graphical model which is displayed on screen (the vertex lists, texture data, etc.), and the physical model which is simulated by the physics engine (the size, shape and mass). The prime reason for taking this approach is the efficiency benefits that it brings to the collision detection algorithms.
Checking for collisions with every single polygon of a complex graphical model is a highly computationally intensive task. Even if this were possible at the kind of real time frame-rate that we strive for, the player is unlikely to notice such extreme attention to detail. Instead, each game object is represented by a simplified shape, or a collection of simplified shapes, for the purpose of physical simulation.

For example, a telegraph pole may be simulated as a tall thin cuboid, while a hand grenade may be simulated as a sphere. The more complex the shape that is used to simulate an object, the more complex the collision algorithms need to be, so care must be taken to use the appropriate simulation shapes for each object in various circumstances during the game. These simulation shapes will not always be simple primitives (spheres, cubes, etc.), but they will almost always be simpler than the graphical representation of the object.

As a result of this, we need to store, somewhere, the 'physical' representation of our game object. This representation might well be hierarchical. Consider the example of a complex, destructible aeroplane. Due to its complexity, broad phase includes a (cheaper) capsule-based check to see if a narrow phase check is warranted. As such, the aeroplane needs both a broad phase physical representation (the capsule) and a narrow phase representation (e.g., a collection of convex objects which more closely represent the physical reality), accessed by different stages of the physics update loop.

Collision Response Data

The aim of the collision detection algorithms is not only to flag when an intersection of two bodies has occurred, but to provide information on how to resolve that intersection. Basically we need to know where to move the bodies so that there is no longer an intersection.



The diagram shows a rectangular block approaching a static sphere with velocity V. At the end of the motion update phase of the simulation, the collision detection algorithms detect an intersection

which must be resolved (i.e. the corner of the block has penetrated the sphere). The information which the collision detection algorithms must provide, in order to resolve this intersection, is:

- The contact point P where the intersection has been detected.
- The contact normal N which represents the direction vector along which the intersecting object needs to move to resolve the collision.
- The penetration depth, i.e. the minimum distance along the normal that the intersecting object must move.

In the example diagram, the collision response routine has moved the block out along the normal of the sphere so that it is no longer intersecting. The manner in which this is achieved is addressed in the next tutorial; for now we need to concentrate on how the physics engine recognised that an intersection had occurred, and how it calculated the information required by the collision response algorithms.

We can define a `struct` called CollisionData to act as a container for this information:

```
1 struct CollisionData
2 {
3   float penetration;
4   Vector3 normal;
5   Vector3 pointOnPlane;
6 };
```

CollisionData Struct

We note that while this information is suitable to resolve relatively simple collisions, more complex collision resolution requires more information. This will be explored in more detail in the tutorial on Manifolds.

Separating Axis Theorem

There are many different algorithms which can be used to detect whether three-dimensional bodies are intersecting. As ever, the more accurate routines are more computationally expensive, so care should be taken when choosing which algorithm to use in which circumstances. In our last tutorial, we discussed several simple collision detection algorithms which could easily be used to enhance the broad phase of our physics engine, to cull impossible interfaces.

Once we have pared down the number of possible interfaces to a collection of complex objects, however, or we have reached a point where the process of paring becomes more costly than simply undertaking the checks explicitly, we need to employ an algorithm which can detect interfaces between those complex, potentially arbitrary, shapes as the basis of our narrow phase.

Introduction to SAT

Fundamentally, collision detection solutions exist to detect interfaces between objects (often, but not always, with a view to resolving those interfaces). The complexity of the task is inherently connected not only to the complexity of the objects in question, but the complexity of the environment as a whole.

It would be impossible to explore each related algorithm in detail, and the most efficient of these algorithms are optimised to a point where they reveal very little as learning tools. To that end, and to help you get a better understanding of just what you do to your processor scheduling in your representation of complex objects, we will be looking into the Separating Axis Theorem (SAT), one specific example of a generic collision detection algorithm.

SAT has the benefit of providing both a useful learning environment on which you can build and improve, while also allowing you to represent complicated entities in your engine. You are encouraged to explore other, more sophisticated collision detection algorithms if the domain interests you, and implement them as an advanced element of your coursework for this module.

Convex or Concave?

The first thing to note about an implementation of Separating Axis Theorem is that the algorithm will only work with convex shapes. You can identify whether a shape is convex of concave by how many times a line intersects that shape. A convex shape will only ever have two points of intersection.

Consider Figure 1. You can be seen that irrespective of angle, a line passing through the left-hand polygon will never intersect more than twice. On the other hand, a concave shape will have at least one instance - one angle, if you will - where an intersecting line will have more than two points of contact.

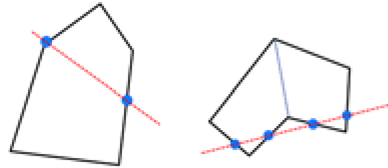


Figure 1: Convex and Concave Objects

While a SAT-based algorithm for collision detection cannot alone account for collisions between concave objects, any concave object can be split into two or more convex objects. Looking again at Figure 1, you can see how this is possible for the right-hand object, which can be partitioned into two quads (along the grey line).

This process can actually be automated, but the implementation of an algorithm which decomposes concave hulls into multiple convex hulls can be fairly daunting. For the purposes of this module, it is assumed that any concave objects you wish to represent (such as those discussed in the coursework specification) will be decomposed manually into constituent convex hulls.

Separating Axis Theorem

The Separating Axis Theorem states that if two objects are NOT colliding, then a line (or plane in 3D) can be drawn between them (see Figure 2). This is in fact the same proof we used implicitly for the sphere-sphere collisions in the previous tutorial.

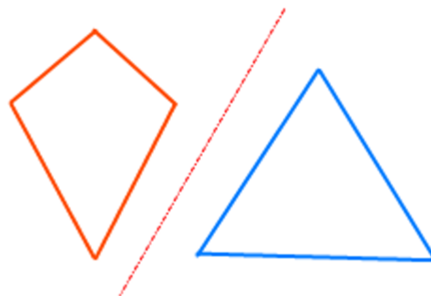


Figure 2: Line Between Two Polygons

The corollary to this is that **if we can find a single case where we can draw a line/plane between the two shapes without intersecting either of them, then we can prove that the two shapes are not colliding**.

This can be achieved by projecting all the points of each shape along the axis being tested. This will give us a single value describing the distance of that point along that given axis. It should be noted that this is the same projection used in the previous sphere-plane test, and that the plane equation will play a significant role in our implementation of SAT.

It is this projection value that can be used to check whether the two objects overlap. As can be seen in Figure 3, if we project the two shapes along the separating axis (shown by extending the green normal infinitely in either direction) then we end up with two lines (orange and blue) that can easily be tested for overlap.

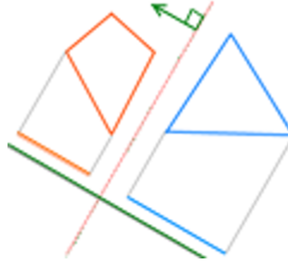


Figure 3: Projection of Two Non-Intersection Objects

The actual code to find the minimum and maximum projections along a vector are already provided in SphereCollisionShape and CuboidCollisionShape.

In the cuboid example, which is the generic example of a convex polygon in this tutorial, all vertices of the cuboid are projected along the axis and the closest and furthest distances are returned as the min/max points. In the sphere example, which is the example of a shape with potentially infinite points, the min/max points are calculated directly from what we know about the shape, in this case it is just $\pm radius \times axis$.

To project a point along an axis we use vector projection, defined as:

$$proj_a b = \frac{a \cdot b}{|b|} \frac{b}{|b|} \tag{1}$$

As b (our axis) is normalised, this can be simplified to $(a \cdot \hat{b})\hat{b}$ which gives us the physical location of the vertex along the given axis. For the purpose of identifying distance however, we use the single value produced by $a \cdot \hat{b}$ as the 'distance' on vector a along our axis $\hat{b}$, or:

$$distance = a \cdot \hat{b} \tag{2}$$

Potential Separating Axes

Now we have seen how we can prove two objects are NOT colliding in a given axis. The problem, and the main issue of the SAT algorithm, will be identifying all possible axes that will need to be evaluated. We can't simply just consider every axis - after all, there are an infinite number of them!

Thankfully, the fact we are dealing with polygonal shapes, rather than pure geometrical shapes, helps us here. In fact, the easiest way to determine which axes to consider is, first, to assume the shapes are not curved, and are in fact made up of a series of lines/faces. Just like (functionally) all objects in a game environment.

This will mean that the number of possible collision points is limited to each flat face. In this case, we can use the normals of the faces of each object as a possible collision axis.

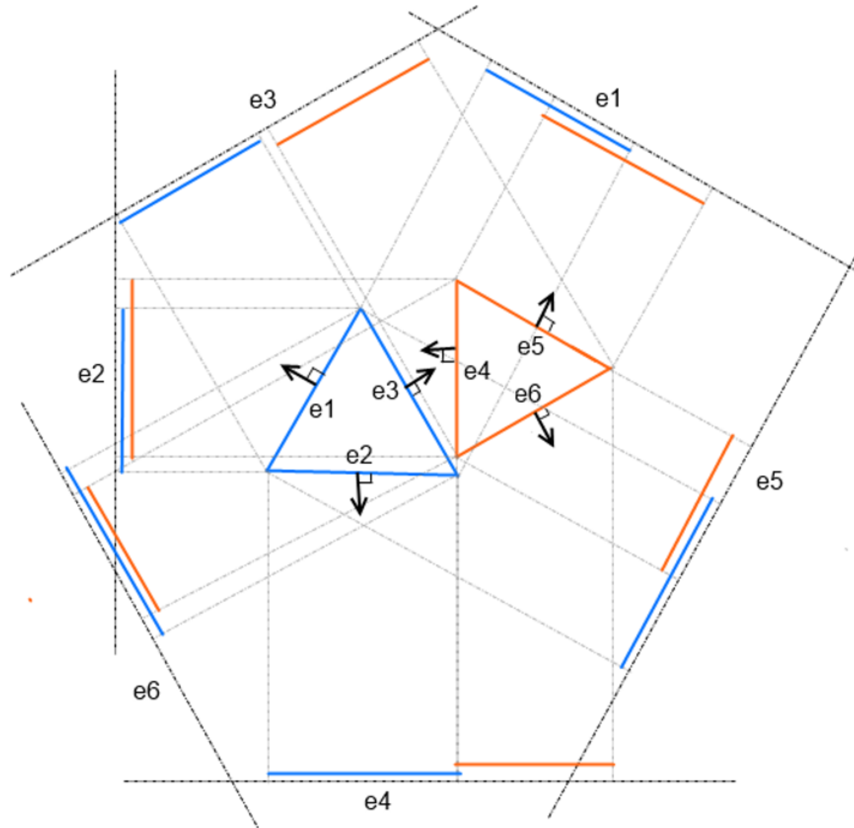


Figure 4: Possible Collision Faces Between Two Triangles

Figure 4 shows all possible collision axes between two 2D triangles. Each test (e1-e6) has been formed from the normal of the corresponding edge. As can be seen on all but the third test (e3) all tests return that the two triangles are in fact colliding, while we can see from the diagram that they are not.

Every normal could potentially be a separating axis and, therefore, all normals must be evaluated. One optimisation we can make to the separating axis theorem is that as soon as a single separating axis is identified, the algorithm can exit early in knowledge that the two objects are **not** colliding. This can be seen in the e3 test above where, programmatically the algorithm could exit early without testing e4-e6.

In the worst case scenario, the total number of axes to be checked is equal to the sum of the faces of both objects (in the case of two triangles, six).

It is also worth noting that parallel axes do not need to be checked multiple times. In an un-rotated square example, this is equivalent to saying that checking the +Y normal produced from the top face is the same as the Y normal produced by the bottom face. This is because we are only concerned with the distance to that plane from each point and not on which side of the plane each point lies (see the sphere-plane example in the previous tutorial).

SAT: Key Features

To summarise the key features of an implementation of separating axis theorem,

- If there exists an axis in which the two objects do **not** overlap, then we can prove that they do not collide.
- If no axis exists in which they do not overlap, then we can safely assume that they **do** collide.

- The number of possible axes is the same as the number of faces on the objects combined (as illustrated in Figure 4).
- Parallel axes can be ignored.

It should be clear at this point why we might choose, for an object with a complex physical representation, to employ checks such as sphere-sphere before employing the (potentially) more computationally expensive polygonal checks of narrow phase collision detection.

Extending SAT into the Third Dimension

Problem: Edge-Edge Collisions

In two-dimensional environments, edges can be considered the same as faces. In 3D, however, this is an incorrect assumption. This requires us to compensate for edge-edge collisions if we're to maintain the correctness of our collision detection algorithm. An example of a 3D case where purely using the normal of each face does not provide the correct result is shown in Figure 5.

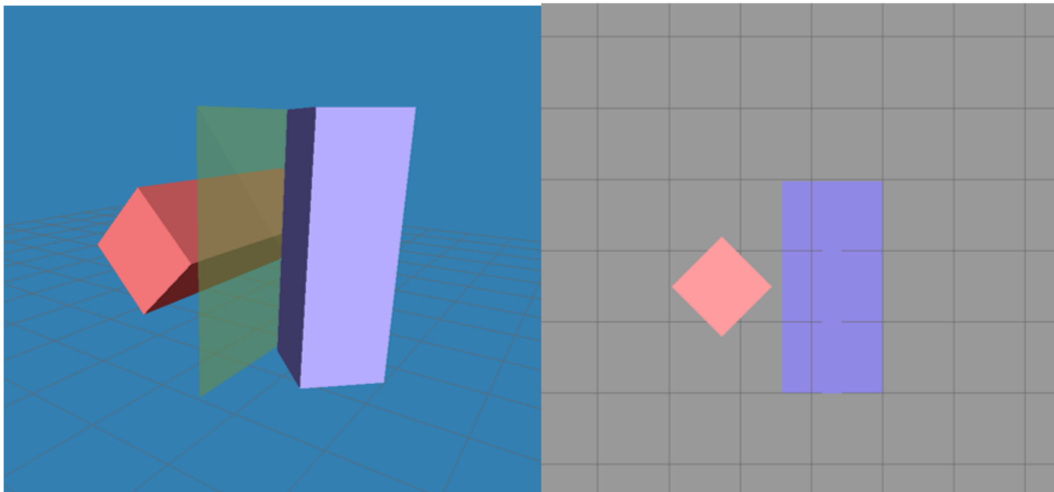


Figure 5: Scenario - Two Cuboids

The two cuboids (Red and Blue) are not colliding, as we know that a plane exists in which they do not collide. Working through our current version of SAT, however, we will see that all possible separating axes return true, meaning the algorithm returns a false positive, detecting a collision where none has occurred. This is illustrated, progressively, in Figure 6.

In order to resolve this problem, we will need to generate additional axes, and check a larger number of possible planes between the two 3D objects.

The simplest way to address the issue, and the way we cover in these tutorials, is to take every edge of both objects and use a cross product of each permutation to produce additional axes. This list will cover all possible edge-edge cases, including the green plane shown in Figure 1.

You should recall from the graphics tutorials that the cross product between two non-parallel vectors will result in a vector that is orthogonal to both of the previous vectors. In simplest example, the x, y, and z axes, the cross product of x and y will be z, $y \times z = x$, and so on.

By using the cross product of an edge on object 1 and an edge on object 2, we will produce a possible axis that is orthogonal to both of those edges and a possible plane in which the two objects

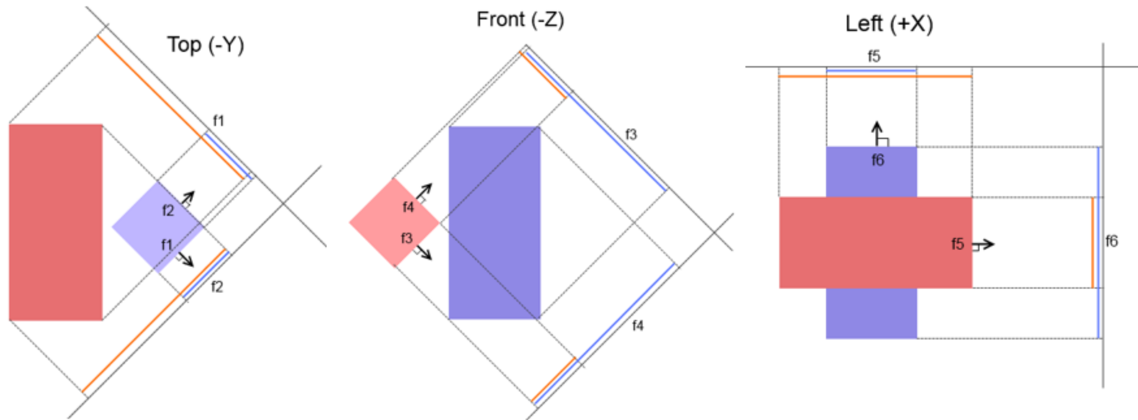


Figure 6: Separating Axis Checks for the Scenario

do not overlap.

Implementing this fix will ensure that we have a robust polyhedron-polyhedron collision detection algorithm which will work for all convex hulls.

One thing of note which has changed in the above code is the additional check for previously found axes. As we now have to check all possible permutations between edges, the number of possible collision axes has vastly increased; being able to remove duplicate axes will save us a lot of time (processor cycles) when we have to iterate through each axis later on.

Problem: Spheres and Curves

One very important aspect of collision detection which we have ignored since Tutorial 5 is how our system will curved surfaces such as spheres. A curve has infinite edges and infinite normals, so being able to handle all possible permutations is no longer viable. Instead we will have to come up with another solution to handle these cases and produce a list of possible collision axes.

A limitation of SAT assists us in this: because we know that all our shapes are convex, there will only ever be one point of contact. This means that all we need to do is find and check a single axis to know whether the curved shape is colliding. The complexity, then, is in determining what that axis is.

Let us consider the simplest case: sphere-sphere. All we have to check is whether the distance between the two centre points is less than their combined radii to prove they are colliding, as illustrated in Tutorial 5. Going back to SAT, the only axis we need to check in this case is the direction between those centre points, as illustrated in Figure 7.

In the sphere-sphere case, we are actually checking the closest point of object 1 against the closest point of object 2. It's just convenient that, with spheres specifically, that direction vector results in the same vector as the one produced by the two centre points.

In the sphere-polygon case this is not always true. Consider the left-hand diagram in Figure 8. The possible collision axis produced by the two centre points will give us a false positive. Therefore we must calculate the closest point of object 2 to the centre point of object 1. This is accomplished by iterating through each face in turn and deducing this closest point.

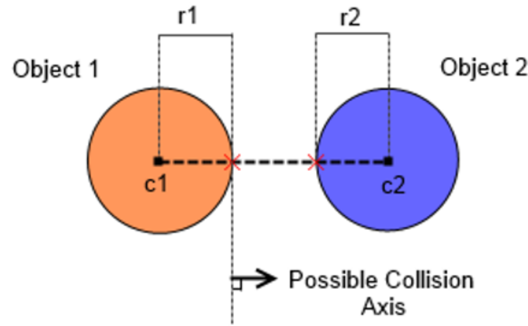


Figure 7: The Sphere-Sphere Case

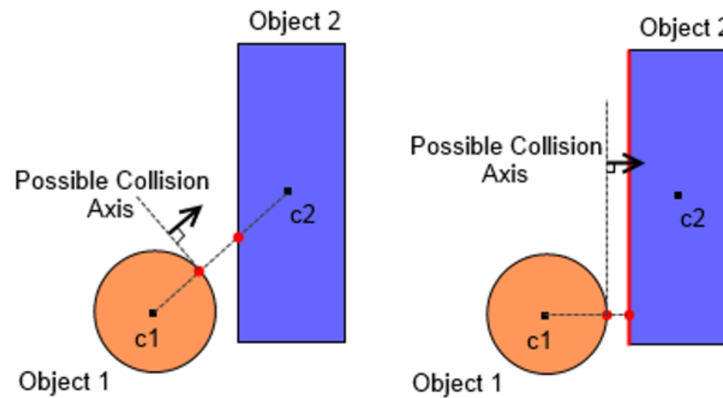


Figure 8: The Sphere-Polyhedron Case

Implementation

Review the tasks for Day 3 on your Practical Tasks handout. This afternoon is a good opportunity to power through any outstanding tasks from previous days, as well as exploring how to make good use of collision detection algorithms without necessarily relating them to collision response.

Tutorial Summary

In this tutorial we have introduced the concept of collision detection. This is the first step in the collisions module of the physics engine – once a collision is detected, the second step is to respond to that by moving the intersecting objects apart. The collision detection routines are two-tiered – broad phase and narrow phase. This tutorial has dealt with the quick collision detection routines which are typically employed during broad phase, or in simple gameplay/AI programming, before introducing the more computationally expensive routines typically used during narrow phase.

```
1 void Narrowphase()
2 {
3    for (all possible collisions)
4       if (collisiondetectionSAT::AreColliding(pair))
5
6       // YAY
7 }
```

General (and inside PhysicsEngine::Narrowphase)

```cpp
void SphereCollisionShape::GetCollisionAxes(
    const PhysicsNode* otherObject,
    std::vector<Vector3>& out_axes) const
{
    // There are infinite possible axes on a sphere so we MUST
    // handle it seperately. Luckily we can just get the closest point
    // on the opposite object to our centre and use that.

    Vector3 dir = (otherObject->GetPosition()
        - Parent()->GetPosition()).Normalise();

    Vector3 p1 = Parent()->GetPosition();
    Vector3 p2 = otherObject->GetCollisionShape()->GetClosestPoint(p1);

    out_axes.push_back((p1 - p2).Normalise());
}

Vector3 SphereCollisionShape::GetClosestPoint(
    const Vector3& point) const
{
    Vector3 diff = (point - Parent()->GetPosition()).Normalise();
    return Parent()->GetPosition() + diff * m_Radius;
}

void SphereCollisionShape::GetMinMaxVertexOnAxis(
    const Vector3& axis,
    Vector3& out_min,
    Vector3& out_max) const
{
    out_min = Parent()->GetPosition() - axis * m_Radius;
    out_max = Parent()->GetPosition() + axis * m_Radius;
}
```

SphereCollisionShape.cpp

```cpp
bool CollisionDetectionSAT::AreColliding(CollisionData* out_coldata)
{
    if (!cshapeA || !cshapeB)
        return false;

    areColliding = false;
    possibleColAxes.clear();

    //<----- DEFAULT AXES ------->

    // GetCollisionAxes takes in the /other/ object as a parameter here

    std::vector<Vector3> axes1, axes2;

    cshapeA->GetCollisionAxes(pnodeB, axes1);
    for (const Vector3& axis : axes1)
        AddPossibleCollisionAxis(axis);

    cshapeB->GetCollisionAxes(pnodeA, axes2);
    for (const Vector3& axis : axes2)
        AddPossibleCollisionAxis(axis);

    //<----- EDGE-EDGE CASES ----->
```

```
24
25    // Handles the case where two edges meet and the final collision
26    // direction is mid way between two default collision axes
27    // provided above. (This is only needed when dealing with 3D
28    // collision shapes)
29
30    // As mentioned in the tutorial, this should be the edge vector's
31    // not normals we test against, however for a cuboid example this
32    // is the same as testing the normals as each normal /will/ match
33    // a given edge elsewhere on the object.
34
35    // For more complicated geometry, this will have to be changed to
36    // actual edge vectors instead of normals.
37
38    for (const Vector3& norm1 : axes1)
39    {
40       for (const Vector3& norm2 : axes2)
41       {
42
43          AddPossibleCollisionAxis(
44          Vector3::Cross(norm1, norm2).Normalise());
45       }
46    }
47
48    // Seperating axis theorem says that if a single axis can be found
49    // where the two objects are not colliding, then they cannot be
50    // colliding. So we have to check each possible axis until we
51    // either return false, or return the best axis (one with the
52    // least penetration) found.
53
54    CollisionData cur_colData;
55
56    bestColData._penetration = -FLT_MAX;
57    for (const Vector3& axis : possibleColAxes)
58    {
59       //If the collision axis does NOT intersect then return
60       // immediately as we know that atleast in one direction/axis
61       // the two objects do not intersect
62
63       if (!CheckCollisionAxis(axis, cur_colData))
64          return false;
65
66       if (cur_colData._penetration >= bestColData._penetration)
67       {
68          bestColData = cur_colData;
69       }
70    }
71
72    if (out_coldata) *out_coldata = bestColData;
73
74    areColliding = true;
75    return true;
76 }
```

CollisionDetectionSAT.cpp

```
77  bool CollisionDetectionSAT::CheckCollisionAxis(
78      const Vector3& axis, CollisionData& out_coldata)
79  {
80      //Overlap Test
81      // Points go:
82      //            +-------------+
83      //      +-----|-----+   2   |
84      //      |  1  |     |       |
85      //      |     +-----|-------+
86      //      +-----------+
87      //      A ------C---B ----- D
88      //
89      // IF A < C AND B > C (Overlap in order object 1 -> object 2)
90      // IF C < A AND D > A (Overlap in order object 2 -> object 1)
91  
92      Vector3 min1, min2, max1, max2;
93  
94      // Get the min/max vertices along the axis from shape1 and shape2
95  
96      cshapeA->GetMinMaxVertexOnAxis(axis, min1, max1);
97      cshapeB->GetMinMaxVertexOnAxis(axis, min2, max2);
98  
99      float A = Vector3::Dot(axis, min1);
100     float B = Vector3::Dot(axis, max1);
101     float C = Vector3::Dot(axis, min2);
102     float D = Vector3::Dot(axis, max2);
103 
104     //Overlap Test (Order: Object 1 -> Object 2)
105     if (A <= C && B >= C)
106     {
107         out_coldata._normal = axis;
108         out_coldata._penetration = C - B;
109         //Smallest overlap distance is between B->C
110         //Compute closest point on edge of the object
111         out_coldata._pointOnPlane =
112             max1 + out_coldata._normal * out_coldata._penetration;
113 
114         return true;
115     }
116 
117     //Overlap Test (Order: Object 2 -> Object 1)
118     if (C <= A && D >= A)
119     {
120         out_coldata._normal = -axis;
121         // Invert axis here so we can do all our resolution phase as
122         // Object 1 -> Object 2
123         out_coldata._penetration = A - D;
124         // Smallest overlap distance is between D->A
125         //Compute closest point on edge of the object
126         out_coldata._pointOnPlane =
127             min1 + out_coldata._normal * out_coldata._penetration;
128 
129         return true;
130     }
131     return false;
132 }
```

CollisionDetectionSAT.cpp