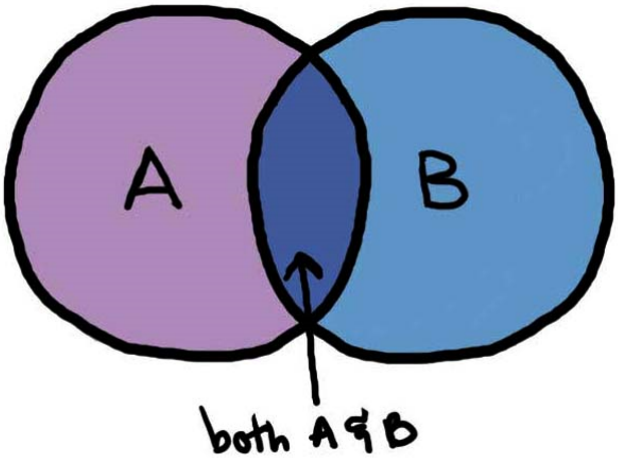
Physics Tutorial 5: Collision Manifolds



Summary

In this tutorial we be expanding our collision detection procedure in order to accurately generate a collision manifold. We will be covering what collision a manifold entails, along with a discussion of the clipping method which will become the main method of computing the collision manifold in this tutorial series.

New Concepts

Contact points, collision manifold, clipping method, Sutherland-Hodgman clipping

Introduction

At this point we have identified when two objects have collided and retrieved the collision/contact normal N as well as the penetration distance p. However, one more piece of information is required before our physics engine can move to the last stage in its update loop and actually resolve our collisions.

Specifically, we need to identify the contact points. Previously, we considered the contact *point* as it applies in collision detection (and collision response, discussed in a future tutorial), but a simple approach is inappropriate to resolving sophisticated collisions.

In this tutorial we will introduce collision manifold. We will define its purpose, and explain how it can be computed using the Clipping Method. By the end of this tutorial, we will have all information required to perform collision response updates, which will be the subject of the final two tutorials in the physics portion of this module.

What is the Contact Point?

At the moment we have the direction of the collision (normal) and the penetration distance. However if this was all that was used to resolve a collision between two objects then no rotation would ever occur in our physics engine. We recall that another piece of collision data exists: the contact point.

A contact point describes a point at which two objects touch. This can be used to resolve collisions in the form of a distance constraint, to constrain the two objects from overlapping in the following time step.

It should be obvious, however, that even a contact point will not necessarily convey all the information required to generate meaningful rotations in response to a collision. Consider the difference between a penny rolling across a surface, and a tyre. If we're going to try and accurately respond to detected collisions, we'll need to gather more data regarding just how our objects are interfacing - this is where the collision manifold is helpful.

What is a Collision Manifold?

A collision manifold is a collection of contact points that form all of the necessary constraints that allow the object to properly resolve all penetrations. It can be seen as the summation of the surface area between two colliding objects. As shown in Figure 1, this could form either a single point, a line or a 2D polygon.

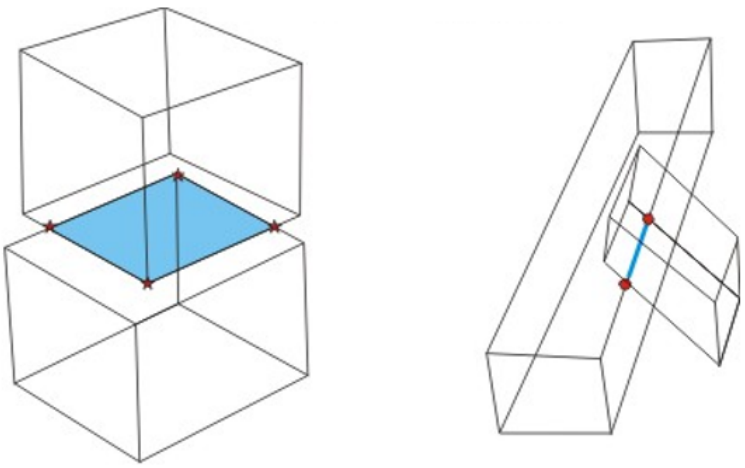


Figure 1: The Contact Manifold

In a discrete physics system, this poses a problem as collisions are only detected after the two objects are already overlapping. This results not in a 2D surface area where the two objects are touching, but rather a 3D volume by which they have already interpenetrated.

To overcome this, we infer the contact manifold as though the two objects were only touching. This permits us to handle the collision resolution as though this were a real event (as interpenetration doesn't occur in real-world occurrences of the collisions we're modelling).

The Clipping Method

To compute the manifold we will be using the clipping method, in which we will be progressively clipping a face of one object with the perimeter of a second object. This results in a 2D collision manifold which can then be used in our resolution calculations.

The best way to show how this algorithm works is through an example. Consider the scenario shown in Figure 2.

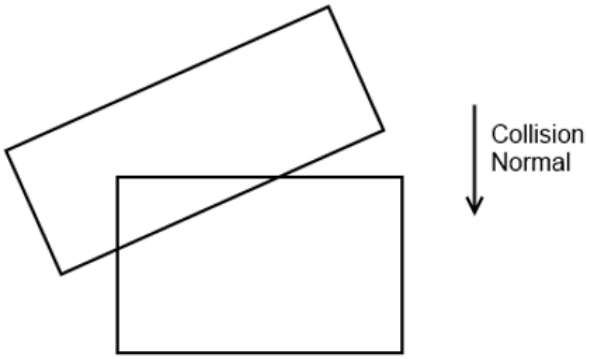


Figure 2: Collision Scenario

In this hypothetical case, two boxes have collided. At this point in our execution, we have just finished executing our SAT routines and know both the collision normal N and penetration depth p.

There are several steps to determining the manifold through the clipping method, and we will address each in turn, beginning with the process by which we identify significant faces (those involved in the collision).

Identifying the Significant Faces

The first step is to identify the significant faces that are intersecting. This is accomplished by selecting the vertex furthest along the collision normal. In Figure 3, these vertices are highlighted with red circles.

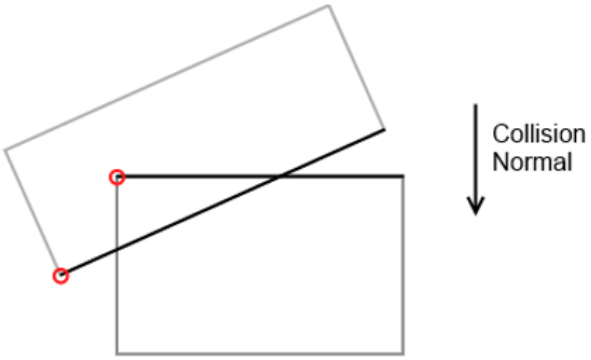


Figure 3: Furthest Vertices Along the Collision Normal

We next select a face on each object which satisfies the following criteria:

- The face includes the selected vertex
- The face's normal is the closest to parallel with the collision normal of all faces which contain the selected vertex

Doing this for both objects gives us the two most significant faces for contact generation.

N.B.: The normal is inverted when selecting the vertex of the second object.

Calculating the Incident and Reference Faces

The reference face will become the point of reference when clipping occurs in subsequent stages of the check. The incident face will, in turn, become a set of vertices that will be clipped.

To do this we compute which of the two significant faces have a normal that is closest to parallel with that of the collision normal. Consider Figure 4. In this case the normal of the face indicated by a blue line is closest to parallel and, as such, that face becomes the reference face.

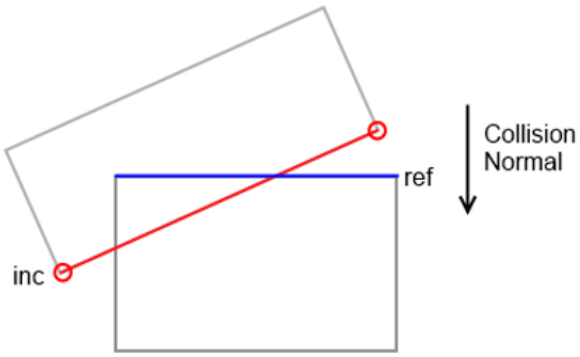


Figure 4: The Contact Manifold

The other face then becomes the incident face which we will clip to generate the contact points. In our example it is comprised of two vertices.

Adjacent Face Clipping

We now clip the incident with all the adjacent faces of the reference. This is done by taking the adjacent faces normal and any vertex that it contains to produce a plane equation. The algorithm we use to compute the clipping is known as Sutherland-Hodgman Clipping. This can easily be adapted to suit a 3D scenario, making it appropriate for use in our physics engine.

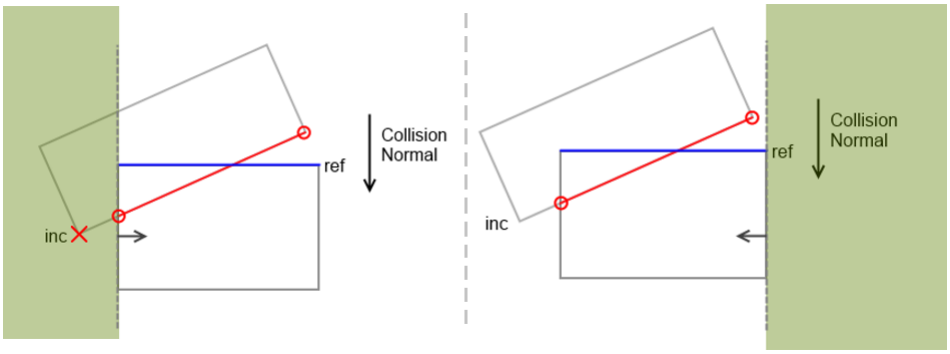


Figure 5: Adjacent Face Clipping

The first clipping plane we will illustrate in this example is the left hand face, shown on the left hand side of Figure 5. As one of the vertices of the incident face lies within the clipping region, it will be replaced with a vertex that lies on the edge of the clipping plane. The second clipping plane is that of the right hand face, shown on the right hand side of Figure 5. In this case, it should be noted that none of the points of the incident face lie in the clipping region, so no changes will be made.

Final Clipping

The final clip plane is that of the reference face itself. However instead of clipping the incident face as in the previous stage, we now just remove all points that lie inside the clipping region. In this shown in Figure 6, this leaves us with just a single contact point and not a line or polygon.

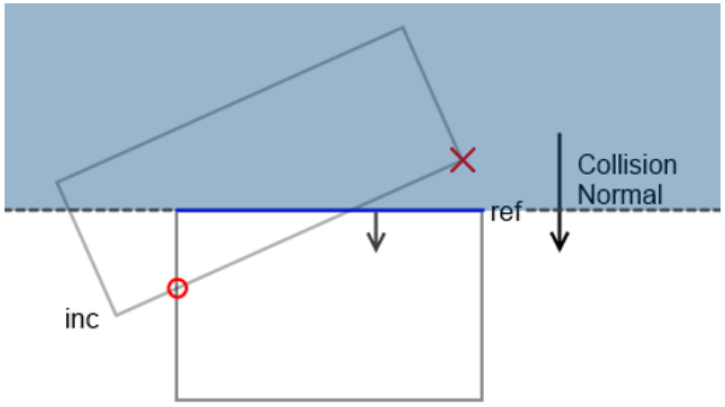


Figure 6: Adjacent Face Clipping - Right Hand Face

Although at first glance it seems like we are ignoring critical contact points, this is in fact correct. What we are trying to infer are the points of contact when the two objects first touched, not all which have occurred since they overlapped. In this example, only the corner of the reference face would be in contact with the other object; that is obvious when comparing the shapes with the direction of travel along the collision normal.

This may seem a wasteful check, given that the manifold produced is only a single contact point. It is important to remember, though, that the collision manifold can be one point, two points, or many; the reason we undertake the clipping process is to obtain the most accurate idea of the collision manifold as possible, irrespective of the number of points generated. If we consider the nature of collisions between convex objects in general, it should be obvious that most collision manifolds will only be a single point - as they represent the manner in which the objects first began to interface; the clipping method lets us resolve the more complex scenarios where this is not the case.

Implementation

Review day 4 of the Practical Tasks handout. Try to make good use of additional time this afternoon to extend your collision detection approach to suit even more complex objects. If you have the opportunity, look into extending your approach to broad phase culling.

Tutorial Summary

In this tutorial we have introduced the concept of the collision manifold, and explained its importance in obtaining believable collision responses, particularly in the context of angular motion. We have determined step-by-step how to extract the collision manifold in an efficient manner which is easily adaptable to suit our physics engine. We now have all collision data required to implement the final stage of our physics update loop: collision response.

```

// PhysicsEngine::NarrowPhaseCollisions()

// After:
    bool okA =
        cp.pObjectA->FireOnCollisionEvent(cp.pObjectA, cp.pObjectB);
    bool okB =
        cp.pObjectB->FireOnCollisionEvent(cp.pObjectB, cp.pObjectA);

// Insert:

if(okA && okB)
{
    // Build full collision manifold that will also handle the
    // collision response between the two objects in the solver
    // stage

    Manifold* manifold = new Manifold();

    manifold->Initiate(cp.pObjectA, cp.pObjectB);

    // Construct contact points that form the perimeter of the
    // collision manifold

    colDetect.GenContactPoints(manifold);

    if (manifold->contactPoints.size() > 0)
    {
        // Add to list of manifolds that need solving
        manifolds.push_back(manifold);
    }
    else
        delete manifold;
}
```

PhysicsEngine.cpp

```

// CollisionDetectionSAT::GenContactPoints()

if (!out_manifold || !areColliding)
    return;

if (bestColData._penetration >= 0.0f)
    return;

// Get the required face information for the two shapes around the
// collision normal

std::list<Vector3> polygon1, polygon2;
Vector3 normal1, normal2;
std::vector<Plane> adjPlanes1, adjPlanes2;

cshapeA->GetIncidentReferencePolygon(
    bestColData._normal, polygon1, normal1, adjPlanes1);

cshapeB->GetIncidentReferencePolygon(
    -bestColData._normal, polygon2, normal2, adjPlanes2);
```

```
22
23 // If either shape1 or shape2 returned a single point, then it must
24 // be on a curve and thus the only contact point to generate is
25 // already availble
26
27 if (polygon1.size() == 0 || polygon2.size() == 0)
28 {
29    return; // No points returned, resulting in no possible contact
30          // points
31 }
32 else if (polygon1.size() == 1)
33 {
34    out_manifold->AddContact(
35       polygon1.front(), //Polygon1 -> Polygon 2
36       polygon1.front() + bestColData._normal
37       * bestColData._penetration, bestColData._normal,
38       bestColData._penetration);
39 }
40 else if (polygon2.size() == 1)
41 {
42    out_manifold->AddContact(
43       polygon2.front() - bestColData._normal
44       * bestColData._penetration,
45       polygon2.front(), //Polygon2 <- Polygon 1
46       bestColData._normal,
47       bestColData._penetration);
48 }
49 else
50 {
51    // Otherwise use clipping to cut down the incident face to fit
52    // inside the reference planes using the surrounding face planes
53
54    // First we need to know if have to flip the incident and reference
55    // faces around for clipping
56
57    bool flipped = fabs(Vector3::Dot(bestColData._normal, normal1))
58       < fabs(Vector3::Dot(bestColData._normal, normal2));
59
60    if (flipped)
61    {
62       std::swap(polygon1, polygon2);
63       std::swap(normal1, normal2);
64       std::swap(adjPlanes1, adjPlanes2);
65    }
66
67    // Clip the incident face to the adjacent edges of the reference
68    // face
69
70    if (adjPlanes1.size() > 0)
71       SutherlandHodgmanClipping(polygon2, adjPlanes1.size(),
72          &adjPlanes1[0], &polygon2, false);
73
74    // Finally clip (and remove) any contact points that are above
75    // the reference face
76
77    Plane refPlane =
78       Plane(-normal1, -Vector3::Dot(-normal1, polygon1.front()));
79    SutherlandHodgmanClipping(polygon2, 1, &refPlane, &polygon2, true);
```

```
80
81      // Now we are left with a selection of valid contact points to be
82      // used for the manifold
83
84      for (const Vector3& point : polygon2)
85      {
86          //Compute distance to reference plane
87
88          Vector3 pointDiff =
89              point - GetClosestPointPolygon(point, polygon1);
90          float contact_penetration =
91              Vector3::Dot(pointDiff, bestColData._normal);
92
93          // Set Contact data
94
95          Vector3 globalOnA = point;
96          Vector3 globalOnB =
97              point - bestColData._normal * contact_penetration;
98
99          // If we flipped incident and reference planes, we will
100         // need to flip it back before sending it to the manifold.
101         // e.g. turn it from talking about object2->object1 into
102         // object1->object2
103
104         if (flipped)
105         {
106             contact_penetration = -contact_penetration;
107             globalOnA =
108                 point + bestColData._normal * contact_penetration;
109
110             globalOnB = point;
111         }
112
113         // Just make a final sanity check that the contact point
114         // is actual a point of contact not just a clipping bug
115
116         if (contact_penetration < 0.0f)
117         {
118             out_manifold->AddContact(
119                 globalOnA,
120                 globalOnB,
121                 bestColData._normal,
122                 contact_penetration);
123         }
124     }
125 }
```

CollisionDetectionSAT.cpp