Simulating the Rigid World

# Separating Axis Theorem

The Separating Axis Theorem (SAT) is the foundation for one of the most popular algorithms used for detecting collision between general convex shapes in 2D. Since the derived algorithm can be overly computationally intensive for real time systems, it is typically preceded with an initial pass of broad phase method, as introduced in the previous section. The SAT states that:

Two convex polygons are not colliding if there exists a line (or axis) that is perpendicular to one of the given edges of the two polygons and when projecting all edges of the two polygons onto this axis results in no overlaps of the projected edges.

In other words, given two convex shapes in 2D space you can iterate through all of the edges of the convex shapes, one at a time. For each of the edges, compute a line (or axis) that is perpendicular to the edge, project all edges of the two convex shapes onto this line, and compute for overlaps of the projected edges. If you can find one of the perpendicular lines where none of the projected edges overlaps, then the two convex shapes do not collide.

Figure 3-9 illustrates this description using two axes-aligned rectangles. In this case, there are two lines that are perpendicular to the two given shapes, the X and Y axes.



Figure 3-9. There exist a projection that does not overlap.

When projecting all of the shape edges onto these two lines, note that the projection results on the Y-axis overlaps, while there is no overlap on the X-axis. Since there exist one line that is perpendicular to one of the rectangle edges where the projected edges do not overlap, the SAT concludes that the two given rectangles do not collide.

The main strength of algorithms derived from the SAT is that for non-colliding shapes it has an early exit capability. As soon as an axis with no overlapping projected edges is detected, an algorithm can report no collision and does not need to continue with the testing for other axes. In the case of Figure 3-9, if the algorithm began with processing the X-axis, there would be no need to perform the computation for the Y-axis.

## A Simple SAT Based Algorithm

Algorithms derived based on the SAT typically consists of four steps:

* **Step 1 Compute Face Normals**: Compute the perpendicular axes, or face normals for projecting the edges. As illustrated in Figure 3-10, a rectangle has four edges and each edge has a corresponding perpendicular axis. For example, A1 is the corresponding axis for and thus is perpendicular to the edge eA1. Note that in your rigid rectangle implementation, mFaceNormal, or face normals, are the perpendicular axes A1, A2, A3, and A4.



Figure 3-10. Rectangle Edges and Face Normals

* **Step 2 Project Vertices**: Project each of the vertices of the two convex shapes onto the face normals. Figure 3-11 illustrates this projection of all vertices onto the A2 axis from Figure 3-10.



Figure 3-11. Project each vertex onto face normals (example shows A1)

* **Step 3 Identify Bounds**: Identifies the min and max bounds for the projected vertices of each convex shape. Continue with the previous rectangle example, Figure 3-12 shows identifying the min and max positions for each of the two rectangles. Notice that the min/max positions are defined with respect to the direction of the given axis.



Figure 3-12. Identify the min and max bound positions for each rectangle.

* **Step 4 Determine overlaps**: Determines if the two min/max bounds overlap. Figure 3-13 shows that the two projected bounds do indeed overlap. In this case, the algorithm cannot conclude and must proceed to process the next face normal. Notice that as illustrated in the drawing on the right of Figure 3-10, process of face normal B1 will result in a deterministic conclusion of no collision.



Figure 3-13. Test for overlap for every axis of projection (example using A1).

The given algorithm is capable of determining if a collision has occurred with no additional information. Recall that after detecting a collision, the physics engine must also resolve potential interpenetration and derive a response for the colliding shapes. Both of these computations require additional information--the collision information as introduced in Figure 3-6. The next section introduces an efficient SAT-based algorithm that computes support points to both inform the true/false outcome of the collision detection and serve as the basis for deriving collision information.

## An Efficient SAT Algorithm: The Support Points

A support point for a face normal of shape-A is defined to be the vertex position on shape-B where the vertex has the most negative distant from the corresponding edge of Shape-A. This is illustrated in Figure 3-14 for the face normal A1 of shape-A. The vertex SA1 on shape-B has the largest negative distant from edge eA1 when measured along the A1 direction, and thus SA1 is the support point for face normal A1. The negative distance signifies that the measurement is directional and that a support point must be in the reversed direction from the face normal.



Figure 3-14. Support Points of Face Normals

In general, the support point for a given face normal may be different during every update cycle and thus must be recomputed during each collision invocation. In addition, and very importantly, it is entirely possible for a face normal to not have a defined support point.

### Support Point May Not Exist for a Face Normal

A support point is defined only when the measured distance along the face normal has a negative value. For example, the face normal B1 of shape-B in Figure 3-14 does not have a corresponding support point on shape-A. This is because all vertices on shape-A are positive distances away from the corresponding edge eB1 when measured along B1. The positive distances signify that all vertices of shape-A are *in front* of the edge eB1. In other words, the entire shape-A is in front of the edge eB1 of shape-B and thus the two shapes are not physically touching, and thus they are not colliding.

It follows that, when computing the collision between two shapes, if any of the face normals does not have a corresponding support point, then the two shapes are not colliding. Once again, the early exit capability is an important advantage--the algorithm can return a decision as soon as the first case of undefined support point is detected.

For convenience of discussion and implementation, the distance between a support point and the corresponding edge is referred to as the support point distance and this distance is computed as a positive number. In this way, the support point distance is actually measured along the negative face normal direction. This will be the convention followed in the rest of the discussions in this book.

### The Axis of Least Penetration and Collision Information

When support points are defined for all face normals of a convex shape, the face normal of the smallest support point distance is the axis leading to the least interpenetration. Figure 3-15 shows the collision between two shapes where supports points for all of the face normals of shape-B are defined: vertex SB1 on shape-A is the corresponding support point for face normal B1, SB2 for face normal B2, and so on. In this case, SB1 has the smallest corresponding support point distance and thus the face normal B1 is the axis that leads to the least interpenetration. The illustration on the right on Figure 3-15 shows that in this case, support point distance is the collision depth, face normal B1 is collision normal, support point SB1 is the start of the collision, and the end of the collision can be readily computed, it is simply SB1 offset by collision depth in the collision normal direction.



Figure 3-15. Axis of Least Penetration and The Corresponding Collision Information.

### The Algorithm

With the background description, the efficient SAT-based algorithm to compute the collision between two convex shapes, A and B, can be summarized as:

* Compute the support points for all the face normals on shape-A

If any of the support points is not defined, there is no collision

If all support points are defined, compute the axis of least penetration

* Compute the support point for all the face normals on shape-B

If any of the support points is not defined, there is no collision

If all support points are defined, compute the axis of least penetration

The collision information is simply the smaller collision depth from the above two results. You are now ready to implement the support point SAT algorithm.

## The Rectangle Collisions Project

This This project will guide you to implement the support point SAT algorithm.. This project introduces the rigid shapes and the bounds. You can see an example of this project running in Figure 9-X2. The source code to this project is defined in chapter9/9.3.rectangle\_collisions.

Figure 9-X1. Running the Rectangle Collisions project

The controls of the project are as follows, for both scenes:

* **This and that**
* **This and that**.

The goals of the project are as follows:

* To To gain insights into and implement the support point SAT algorithm
* To lay the foundation for building a narrow phase collision detection algorithm
* To define collision information
* To compute and display collision information for circles

You can find the following external resource files in the assets folder: this file and tht file (no changes)

### Modify Rectangle Collision

Begin by modifying the Rectangle\_collision.js file to implement the collision detection between rectangles.

1. Edit the Rectangle\_collision.js file in the RigidBody folder.
2. Create a new function findSupportPoint to compute a support point based on, dir, the negated face normal direction, ptOnEdge, a position on the given edge (e.g., a vertex). The following code marches through all the vertices; compute vToEdge, the vector from vertices to ptOnEdge; project this vector onto the input dir; and record the largest positive projected distant. Recall that dir is the negated face normal direction, and thus the largest positive distant corresponds to the furthest vertex position. Additionally, it is entirely possible for all of the projected distances to be negative. In such cases, all vertices are in front of the input dir, a support point does not exist for the given edge, and thus the two rectangles do not collide.

Rectangle.prototype.findSupportPoint = function (dir, ptOnEdge) {

//the longest project length

var vToEdge;

var projection;

// initialize the computed results

tmpSupport.mSupportPointDist = -9999999;

tmpSupport.mSupportPoint = null;

//check each vector of other object

for (var i = 0; i < this.mVertex.length; i++) {

vToEdge = this.mVertex[i].subtract(ptOnEdge);

projection = vToEdge.dot(dir);

//find the longest distance with certain edge

//dir is -n direction, so the distance should be positive

if ((projection > 0) && (projection > tmpSupport.mSupportPointDist)) {

tmpSupport.mSupportPoint = this.mVertex[i];

tmpSupport.mSupportPointDist = projection;

}

}

};

1. With the ability to locate a support point for any face normal, the next step is the find the axis of least penetration by implementing the findAxisLeastPenetration function. Recall that the axis of least penetration is derived based on the support point with the least support point distant. The following code loops over the four face normals; finds the corresponding support point and support point distance; and records the shortest distance. The while-loop signifies that if a support point is not defined for any of the face normals then the two rectangles do not collide.

Rectangle.prototype.findAxisLeastPenetration = function (otherRect, collisionInfo) {

var n;

var supportPoint;

var bestDistance = 999999;

var bestIndex = null;

var hasSupport = true;

var i = 0;

while ((hasSupport) && (i < this.mFaceNormal.length)) {

// Retrieve a face normal from A

n = this.mFaceNormal[i];

// use -n as direction and the vectex on edge i as point on edge

var dir = n.scale(-1);

var ptOnEdge = this.mVertex[i];

// find the support on B

// the point has longest distance with edge i

otherRect.findSupportPoint(dir, ptOnEdge);

hasSupport = (tmpSupport.mSupportPoint !== null);

//get the shortest support point depth

if ((hasSupport) && (tmpSupport.mSupportPointDist < bestDistance)) {

bestDistance = tmpSupport.mSupportPointDist;

bestIndex = i;

supportPoint = tmpSupport.mSupportPoint;

}

i = i + 1;

}

if (hasSupport) {

//all four directions have support point

var bestVec = this.mFaceNormal[bestIndex].scale(bestDistance);

collisionInfo.setInfo(bestDistance, this.mFaceNormal[bestIndex], supportPoint.add(bestVec));

}

return hasSupport;

};

1. You can now implement the collidedRectRect function by computing the axis of least penetration with respective each of the two rectangles and choosing the smaller of the two results.

Rectangle.prototype.collidedRectRect = function (r1, r2, collisionInfo) {

var status1 = false;

var status2 = false;

//find Axis of Separation for both rectangle

status1 = r1.findAxisLeastPenetration(r2, collisionInfoR1);

if (status1) {

status2 = r2.findAxisLeastPenetration(r1, collisionInfoR2);

if (status2) {

//if both of rectangles are overlapping, choose the shorter normal as the normal

if (collisionInfoR1.getDepth() < collisionInfoR2.getDepth()) {

var depthVec = collisionInfoR1.getNormal().scale(collisionInfoR1.getDepth());

collisionInfo.setInfo(collisionInfoR1.getDepth(),

collisionInfoR1.getNormal(),

collisionInfoR1.mStart.subtract(depthVec));

} else {

collisionInfo.setInfo(collisionInfoR2.getDepth(),

collisionInfoR2.getNormal().scale(-1),

collisionInfoR2.mStart);

}

}

}

return status1 && status2;

};

1. Complete the implementation by modifying the collisionTest function to call the newly defined collidedRectRect function to compute the collision between two rectangles.

Rectangle.prototype.collisionTest = function (otherShape, collisionInfo) {

var status = false;

if (otherShape.mType === "Circle") {

status = false;

} else {

status = this.collidedRectRect(this, otherShape, collisionInfo);

}

return status;

};

### Observations

You can now run the project to test your implementation. Try creating multiple rectangles with the F key. You can see an orange line representing collision information (collision depth, in the collision normal direction, from start to end) when two or more rectangles collide. Remember that this line shows the least amount of positional correction needed to resolve the collision. Use to up and down arrows to select and rotate the rectangles and observe how the collision info changes accordingly. At this stage you have implemented collision detection between a circle and a circle, as well as a rectangle and another rectangle. If you try to collide a rectangle and a circle, no collision info is generated because you have not implemented support for this type of collision. This is will be resolved in the next project.