ISAAC: A RIGID BODY PHYSICS SIMULATOR

This is ISAAC.

ISAAC is capable of simulating rigid body dynamics. Here's a brief description of how the algorithm functions.

Each body in the environment is a polygon. Each polygon is an object of the class body which apart from the variables required for GUI implementation, the following members:

- int Exist
  This is a Boolean that checks whether the body is in the environment or not. No operations are performed if Exists is 0.
- int NumVertices
  This is the number of Vertices in the polygon.
- int IsConcrete
  This is a Boolean that checks whether or not a body is Concrete. Concrete objects behave slightly different.
- double COM[2]
  This is a vector that stores co-ordinates of the Centre of Mass of the body with respect to environment space.
- double LocalSpaceList[50][2]
  This is the Space list containing coordinates of vertices with respect to the Centre of Mass being the origin.
- double EnvSpaceList[50][2]
  This is the list of co-ordinates of vertices in the environment space.
- double Lvel[2]
  This is the Linear Velocity vector of the body.
- double Avel
  This is the angular velocity of the body.
- double Mass
  This is the mass of the body, calculated automatically by calculating surface area and multiplying with density.
- double MOI
  This is the Moment of Inertia, calculated automatically.
- double Spin
  This is the angular displacement of the body.
- double MaxDist
  This is the Maximum Distance for an individual body between the centre of mass and the vertex.

This is all the information that the engine needs to render and simulate the bodies.

The following is the algorithm of the engine:

1.) Run a nested loop for each pair of body though all layers. If the bodies overlap, and if the approach is positive (as in, they are moving towards each other), calculate and impart the impulse (IE. Update each individual velocities.

2.) Integrate over a small finite time, all the positions and update the space vertex position lists in all bodies. Gravity should be taken into account at this step.

3.) Render each body

To render a body in the space, the coordinates of a vertex in Environment space are obtained as the following:

$$\mathbf{r}_{\textbf{in space}} = \mathbf{r}_{\textbf{COM in space}} + \emptyset(\theta)\ \mathbf{r}_{\textbf{with respect to COM}}$$

where **r** is the relevant position vector and $\emptyset(\theta)$ is the angular transformation matrix $\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix}$ where θ is the angular displacement.

Here $\emptyset$ is a function of angular displacement, which for each frame is incremented by angular displacement (ω) times finite time length (dt).

$$\boldsymbol{\theta}_{\textbf{new}} = \boldsymbol{\theta}_{\textbf{old}} + \boldsymbol{\omega}\text{dt}$$

Similarly $\mathbf{r}_{\textbf{COM in space}}$ is incremented by the velocity vector times finite time length of the object stored in Lvel [2] above.

$$\boldsymbol{r}_{\textbf{new}} = \boldsymbol{r}_{\textbf{old}} + \boldsymbol{v}\text{dt}$$

This velocity vector is incremented by gravity vector times finite length in every frame. Hence this is how a free body is simulated.

$$\boldsymbol{v}_{\textbf{new}} = \boldsymbol{v}_{\textbf{old}} + \boldsymbol{g}\text{dt}$$

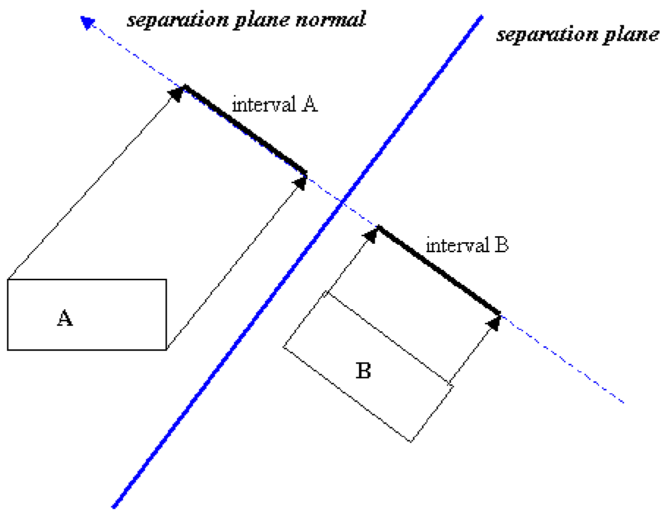
This can be broken down into necessary equations for x and y. Here are is a quote from the program implementing it.

```
void body::FillEnvSpaceList()

    {   double x,y,s=sin(Spin),c=cos(Spin);

        for(int i=0;i<NumVertices;i++)
        {
            x=LocalSpaceList[i][0]*c - LocalSpaceList[i][1]*s;
            y=LocalSpaceList[i][0]*s + LocalSpaceList[i][1]*c;
            EnvSpaceList[i][0]=COM[0]+x;
            EnvSpaceList[i][1]=COM[1]+y;
        }
    }
void body::Update()
    {   FillEnvSpaceList();
        COM[0]+=Lvel[0]*dt;
        COM[1]+=Lvel[1]*dt;
        Spin+=Avel*dt;
        if(Spin>3.1415926)Spin-=2*3.1415926;
        else if(Spin<-3.1415926)Spin+=2*3.1415926;
        Lvel[1]-=g*dt;
    }
```

The most import part is to check whether two bodies are overlapping or not. Polygon Overlap can be tested using Separating Axis Theorem. It states that two objects are non overlapping if and only if there exists a plane (or in our case, a line) separating the two. In case of polygons, there will always be one axis, normal to one of the edges. Hence, for each edge, a normal axis can be taken on which both polygons can be projected. If at least one such axis can be found in which the objects do not intersect, the polygons are not overlapping. The following pictures explain it clearly:



(Picture at http://elancev.name/oliver/2D%20polygon_files/image001a.gif)

So to detect overlapping, a loop has to check normal to each edge, and then project vertices of both the polygons on it. If the regions covered by respective polygons overlap

The objects will then be separated using MTD, which is Minimum Translation Distance, this is the minimum amount of displacement needed for the objects to separate. Also, the normal for which the MTD was obtained is the normal of collision.

Writing a code for this is too long. Here’s a pseudo code of this method.

```
//Checking for body a and body b
for(each edge of a)
{      normal=unitVector(a.NormalforEdge[i]); //Find unit vector
       for(each vertex of a)
       {      if(maxa<ProjectedDistanceAlongNormal(a.EnvSpaceList[i],normal))
                     maxa= ProjectedDistanceAlongNormal(a.EnvSpaceList[i],normal);
              if(mina<ProjectedDistanceAlongNormal(a.EnvSpaceList[i],normal))
                     mina= ProjectedDistanceAlongNormal(a.EnvSpaceList[i],normal);

       }
       for(each vertex of b)
       {      if(maxb<ProjectedDistanceAlongNormal(b.EnvSpaceList[i],normal))
                     maxa= ProjectedDistanceAlongNormal(b.EnvSpaceList[i],normal);
              if(minb<ProjectedDistanceAlongNormal(b.EnvSpaceList[i],normal))
                     mina= ProjectedDistanceAlongNormal(b.EnvSpaceList[i],normal);

       }
       if(overlap(maxa,mina,maxb.minb))
       {      MTD=calculateMTD(maxa,mina,maxb,minb);
              if(MTD<minMTDsoFar)
              {      minMTDsoFar=MTD;
                     probableCandidateForNormal=normal;
                     /*Also, if this is the normal then point of collision wall be
very close to one of the points in mina, maxa, minb, maxb*. So:/

       probablePointOfCollision=IdentifyTheCorrectPointAmong(maxa,mina,maxb,minb);
              }
       }
       else return FALSE;   /*A separating axis has been found if the projected
                                   shadows don’t intersect*/
}

.
.
.
Writing the same code for body b...
.
.
.
//Since no return FALSE has occurred or all edges of a and b, they are overlapping
NormalForCollision=probableCandidateForNormal;
MagnitudeOfPushVector=MTD;
PointOfCollision= probablePointOfCollision;
return TRUE;
```

After this, along the collision axis, and using the magnitude of MTD, a slight displacement would be made on both the objects in the opposite direction. This will successfully prevent overlapping. This push offering displacement is called the Push Vector.

Now having successfully found the points of collision and the axis we must impart the impulse to cause a collision. The necessary equation involved is:
(Complete derivation at http://www.myphysicslab.com/collision.html)

$$j = \frac{-(1+e)\,\bar{\mathbf{v}}_{ab1}\cdot\bar{\mathbf{n}}}{{}^{1}\!/_{m_a} + {}^{1}\!/_{m_b} + (\bar{\mathbf{r}}_{ap}\times\bar{\mathbf{n}})^2/I_a + (\bar{\mathbf{r}}_{bp}\times\bar{\mathbf{n}})^2/I_b}$$

Where

- m_a, m_b = mass of bodies A, B
- $\mathbf{r}_{ap}$ = distance vector from centre of mass of body A to point P
- $\mathbf{r}_{bp}$ = distance vector from centre of mass of body B to point P
- $\mathbf{n}$ = normal (perpendicular) vector to edge of body B
- e = elasticity (0 = inelastic, 1 = perfectly elastic)
- $\mathbf{v}_{ab}$ =velocity of body a with respect to b
- I_a =Moment of inertia of body A
- I_b=Moment of inertia of body B

And that is how the engine does it!

BIBLIOGRAPHY

1.)Separating Axis Theorem and Minimum Translation Distance
http://elancev.name/oliver/2D%20polygon.htm
2.)Rigid Body Simulation - Rahil Baber
3.) Flash-based game called "N-The Way Of The Ninja". The Separating Axis Theorem and MTD.
http://www.metanetsoftware.com/technique/tutorialA.html
http://www.metanetsoftware.com/technique/tutorialB.html
4.) OpenGL programming guide- Dave Shreiner
5.) Tutorial with explanation
http://www.myphysicslab.com/collision.html