National University of Singapore
School of Computing
CS3217: Software Engineering on Modern Application Platforms
AY2010/2011, Semester 2

Problem Set 4: **The Physics Engine**

Issue date:	1 February 2011
Due date:	**11 February 2011 23:59**

Introduction

In this assignment, you will practice software engineering skills you are supposed to have picked up by building a simple 2D physics engine from scratch. The process involves understanding how physics can be simulated, deciding on the ADTs that you need, designing the classes and modules that are necessary and finally, implementing and testing your design.

You should also keep in mind that the physics engine needs to work together with what you have already done in Problem Set 3 because you will integrate this engine with your game in the next problem set. You do not necessarily have to implement your engine according to the rules given in the Appendix, and you are free refer to other references or physics engines. Your physics engine simply has to be able to simulate physical interactions and motion in a “natural” way. You must however write all the code for your engine yourself and are not allowed to include an external physics engine library. You are however allowed to use any Apple frameworks and libraries.

A physics engine is the virtual simulation of the physical aspects of an abstract world, like the mass, shape, velocity, acceleration of each physical body, the forces and torques acting on the bodies. These objects and entities interact based on the laws of physics derived for real world physical entities.

The physics engine will serve as an oracle for the main game engine. After setting up the physics world, the main game engine periodically queries the physics engine for the status of the world; mainly the position and angle of rotation of the physical bodies in the world (by angle of rotation, we mean the angle at which the object is rotated from its standard upright position). Whenever the engine is queried, it simulates the interactions between the physical bodies in the time interval from the last query to the current one. This is therefore a discrete simulation of the physical world. Usually the time interval between each successive queries is fixed. (for example, 1/60 seconds)

In this assignment, we’ll only provide the guidelines on how to build a physics engine. The pleasure of personally designing it is all yours. :)

The entities that you may need to model in a physics engine include:

- The world, which contains a set of physical bodies, the direction and magnitude of the gravity, and some other information specifically related with the simulation, instead of the physics, which are discussed in the Appendix.
- Physical bodies. The attributes for a body includes: mass, moment of inertia, position, shape (in this assignment, we’ll only deal with rectangular shapes, but it would be advisable for your design to be extensible to other types of shapes), velocity, angular velocity, force (you only need to keep a vector sum of all the forces acting on the body), torque and friction coefficient.

- The contact points, which are the points where the physical bodies collide with each other. Note that we hardly encounter cases where two colliding bodies are barely touching each other. Usually, since we are running a discrete simulation, when two bodies collide, they share an intersection area. We therefore need to approximate; find two contact points such that we can think of the two bodies as colliding at these points only. (more on this later)

Make sure you have downloaded `ps04-code.zip`. The teaching staff have provided implementations of an immutable 2-dimensional vector class and an immutable 2x2 matrix class to aid you with some mathematics for this assignment. While these classes may seem simple, please take some time to go through its methods and comments.

Now, let us examine a little example that demonstrates what we can do with the physics engine. We build a world, and add four thin immobile bodies to form walls at the edges of the screen(we make a body immobile by giving it an infinite mass). Next we place some bodies inside the box and start the simulation. As gravity pulls the bodies downward, collisions may happen and afterwhich, the bodies reach the ground.

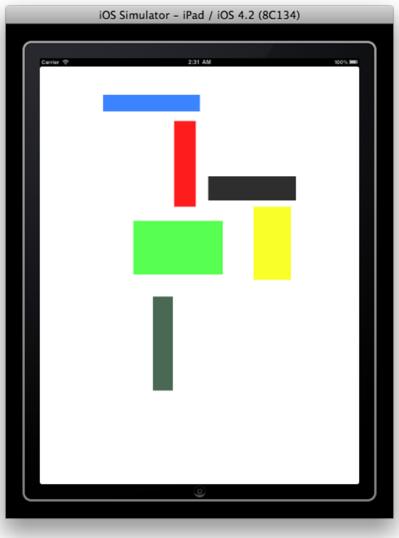

Figure 1: Screenshot of initial state of *Falling Bricks* app.

Figure 1 shows the world at the start of the simulation. We visualize only the part of the world in the rectangular box, since there's nothing going on elsewhere. Figure 2 shows a snapshot of the world during the simulation. Figure 3 shows the world at the end of the simulation.

Important Reminder:	Please read the entire assignment before starting.

Problem 1: Design *(40 points)*

Before you begin, please read through the Appendix to understand how a physics engine works and the requirements for Problem 2 below.

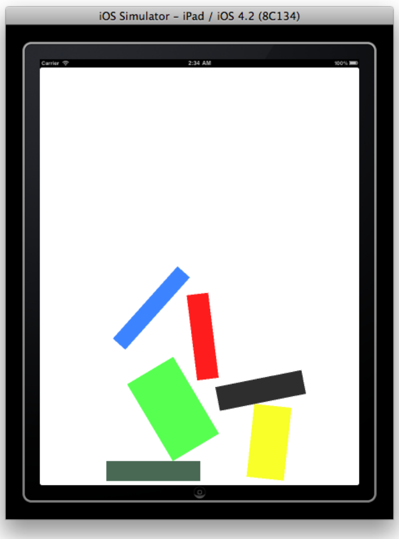

Figure 2: Screenshot of *Falling Bricks* app in the middle of a simulation.

You are expected to write a physics engine capable of simulating the example in the introduction. As stated before, you are free to refer to other physics engines, but you are to write your engine completely from scratch. Note however that if you choose to do something different from what is described in the Appendix, you will thereby bear the risk that it might not work out.

Please answer the following questions by including a PDF file `design.pdf` at the root directory of your project folder.

a. Draw a data model by identifying the data types and how they are related.

b. Draw a module-dependency diagram (MDD) and describe what each module is used for. Explain the rationale for your design over alternatives.

c. Explain how you would extend your design to support more complex shapes.

Problem 2: The Engine *(80 points)*

The physics engine should support a "world" object to which you can add objects. These objects have the following properties:

- mass
- moment of inertia
- position
- shape (though in this assignment, you only need to support rectangular shapes)
- velocity
- angular velocity (about the center of mass)

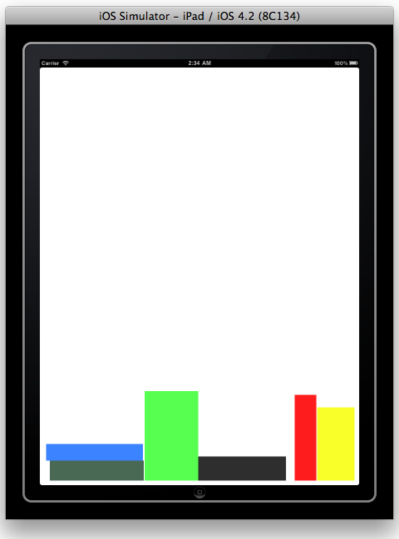

Figure 3: Screenshot of *Falling Bricks* app after objects have come to rest.

In addition you will also need to support (i) forces, (ii) torques and (iii) friction. The world object needs to support a step function that takes in small time step dt and update the state of all the objects that it contains.

Problem 3: Testing *(50 points)*

Testing is an integral part of software engineering. You are expected to create unit tests for the various components in your physics engine. In addition, you will need to do integration testing to make sure that your physics engine works when the various components are put together. The way to do this is to start from a hierarchy and then break down into smaller and more specific cases. For example:

- Black-box testing
  - Test gravity
    - $\cdots$
  - Test collisions between two objects
    - $\cdots$
  - Test collisions between three objects
    - $\cdots$
  - $\cdots$
  - Test that friction works
    - $\cdots$
- Glass-box testing
  - $\cdots$
    - $\cdots$

Please come up with you testing strategy and describe it in `design.pdf`.

Problem 4: Falling Bricks *(30 points)*

To test that your physics engine *really* works like it is supposed to, you are to create a simple application called *Falling Bricks*. This is the application shown in the screenshots in Figures 1 to 3.

This application is simply a box containing six blocks of different colours and different sizes. You get to pick the sizes and colours. Once the application starts, the blocks will fall down and interact according to the rules described in the physics engine. In addition, the app will detect the orientation of the iPad and the bricks will fall accordingly.

Basically, if you hold up the iPad and let the bricks fall and come to rest and then you rotate the iPad, the bricks will fall again. You can also flip the iPad upside down and the bricks should should react accordingly. What you want to create is an app that simulates a box containing free falling bricks.

Hint: This app is actually simpler that it looks. You just have to override `(BOOL)shouldAutorotateToInterfaceOrientation` to prevent the app from auto-rotating and also read the the orientation of the device with `[UIDevice currentDevice].orientation` and update the gravity vector accordingly.

For 10 Bonus Points: Get your app to work with the accelerometer (use the UIAccelerometer class) instead of the orientation. This will make your app more realistic, but it will require more work to get right.

Tip:	You might not want to leave this app till the end. This app can be a useful preliminary testing tool! :-)

Bonus Problem: Reflection (3 Bonus Points).

Please answer the following questions:

a. How many hours did you spend on each problem of this problem set?

b. In retrospect, what could you have done better to reduce the time you spent solving this problem set?

c. What could the CS3217 teaching staff have done better to improve your learning experience in this problem set?

Your answers to these questions should be appended at the end of `design.pdf`.

Grading Scheme

In this module, you are training to become a good software engineer. The first and basic requirement is that your code must satisfy the requirements and be correct. Above and beyond correctness, you are required to write well-documented code.

For this problem set, we will be testing your code by compiling your app and uploading it an iPad to make sure that the blocks fall and interact as expected. You should also probably upload the application to your iPad to make sure that you have figure out how to get the accelerometer to work. We will be looking at the following:

- Your submission should adhere to the submission format.
- You have answered the questions satisfactorily in a concise manner.
- Your project should build without errors or warnings.
- Your project should run without crashing.
- You have replicated the example of falling objects in the introduction.
- Your physics engine works. By this, we mean that the objects move and interact with each other in a realistic way.

Mode of Submission

Please submit a single `.zip` archive file, uploaded to the appropriate folder in the IVLE workbin. The zip file should unzip into a single directory that contains all the source files, and the project file and so on that are necessary for the compilation of your submission. The submitted zip file should adhere to the following naming convention: `CS3217-PS4-[YourName].zip`.

Clarifications and questions related to this assignment may be directed to the IVLE Forum under the header 'Problem Set 4: The Physics Engine'.

Good luck and have fun!

Appendix: Implementing a Physics Engine

In this Appendix, we provide descriptions of how a physics engine can be implemented. Physics engines used in commercial games can be much more sophisticated but if you follow the instructions in this appendix, you should have a functional physics engine for your game.

The goal of a physics engine is quite straightforward. You model physical objects and place them in a physical (in our case, a 2d) space in the engine. The physics engine then attempt to simulate the motions and interactions of the objects in the modeled world. Using the properties of models in your physics engine, you can update the views or display of your objects in your game.

First, you need to represent and keep track of all the objects in your physics engine system. In discrete time steps, the physics would be simulated. Starting from a given state, the input to the system is a time interval dt. The physics engine then have to update the state of all the objects to that after time dt.

Tip: Do NOT test your application only after you complete coding. You should test your application in each stage, or otherwise your code may become too buggy with bugs too complex to debug. Remember: **Test early, Test often!**

Time Stepping

The stepping function can be implemented in 4 steps:

- Apply the forces and torques acting on the bodies, and updating their velocities and angular velocities.
- Perform collision detection and collect a set of contact points where the bodies collide with each other.
- Loop through all the contact points, calculate the impulses at each contact point and change the velocities and angular velocities of the colliding bodies accordingly.
- Update their positions based on their velocities and angular velocities.

Note that we differentiate between the external forces acting on the bodies and the forces caused by collisions between the bodies in the world. or the former, we assume they are applied first, with the duration of dt, before computing the impulses caused by collisions. For the latter, we do not compute the forces. Instead, we deal directly with the changes in momentum (also called impulse). There's no point in finding the force; we compute the change in velocity by dividing the change in momentum by the mass of the body.

And we assume these impulses affect the velocities after the external forces do. Similarly for torques. This is of course, an approximation. However, as demonstrated by the example in introduction, it works reasonably well. Step 4 may need to be repeated for several iterations, to improve the accuracy of the simulation.

Tip: You may wish to work through the equations to better understand them before you implement the system.

Step 1. Applying Forces and Torques

At the start of each time step, we apply the forces and torques on each body according to the following procedure:

$$\vec{v}' = \vec{v} + dt(\vec{g} + \frac{\vec{F}}{m}) \tag{1}$$

$$\vec{\omega}' = \omega + dt\frac{\vec{\tau}}{I} \tag{2}$$

where

- $\vec{g}$ is the gravitational acceleration
- m is the mass of the body
- I is the moment of inertia of the body, computed by

$$I = \frac{width^2 + height^2}{12} m$$

- $\vec{F}$ is the vector sum of forces acting on the body
- $\vec{\tau}$ is the vector sum of torques acting on the body
- $\vec{v}$ and $\vec{v}'$ are the velocities of the body before and after applying the forces
- $\vec{\omega}$ and $\vec{\omega}'$ are the angular velocities of the body before and after applying the torques

Warning: Be careful with the prime($'$) vectors. $v' \neq v$!

Step 2. Collision Detection

In collision detection, our goal is to determine the contact points at which the bodies collide with each other. Moreover, for each contact point, we need to find the normal $\vec{\mathbf{n}}$ and tangential $\vec{\mathbf{t}}$ unit directional vectors for the colliding edge, and the **separation**, which is a measure of the distance between the two colliding bodies (More about this later).

Before we begin our discussion, let us introduce some terminologies to help us explain better. Please refer to Figure 4, which shows a rectangle A in its upright position. O_A is its centre of mass. X_A and Y_A are horizontal and vertical lines that pass through O_A. We denote by $\mathcal{R}_A$ the coordinate system with origin O_A and axes X_A, Y_A. We denote the direction vector from O_A to the topright corner v_1 in the coordinate system $\mathcal{R}_A$ by $\vec{h}_A$.

Note that $\vec{h}_A = \begin{pmatrix} width_A/2 \\ height_A/2 \end{pmatrix}$. The z-component of the vector is simply omitted since it's always zero. As we are doing only a 2-d physics engine, all the vectors, except torque, angular velocity, and others related to rotation, are parallel to the x-y-plane, and thus have a zero z-component. The vectors related to rotation have zero x- and y-components, and possibly non-zero z-components.

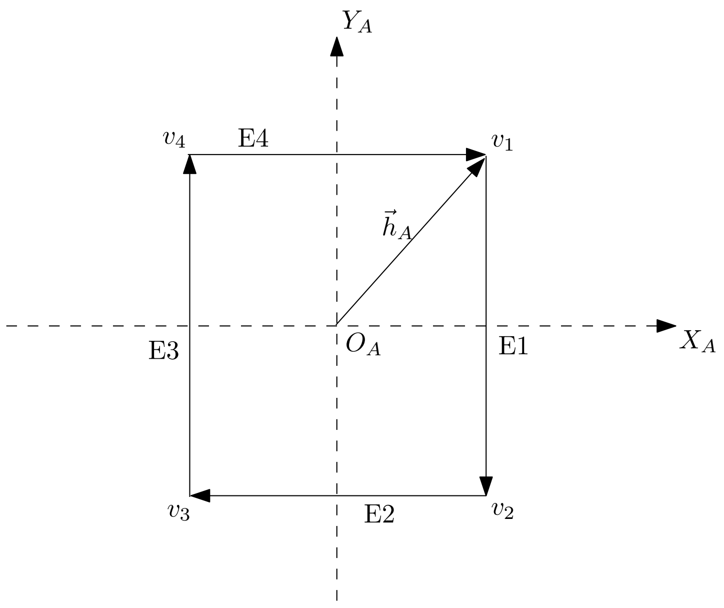

Figure 4: Rectangle Edges

Suppose rectangle A has an angle of rotation θ_A, then the rotation matrices corresponding to this angle is:

$$R_A = \begin{pmatrix} cos\theta_A & -sin\theta_A \\ sin\theta_A & cos\theta_A \end{pmatrix} \tag{3}$$

By the same reasoning above, it is sufficient to formulate the rotation matrices as 2×2 matrices. Note that the first column of R_A is equal to the unit vector of the positive direction of axis X_A in the world coordinate system after the rotation; the second column of R_A is equal to the unit vector of the positive direction of the axis Y_A in the world coordinate system after the rotation. R_A^T, which is the transpose of R_A, corresponds to the rotation of $-\theta_A$ angle.

We refer to the x- and y- and z-components of a vector $\vec{r}$ by $\vec{r}.x$, $\vec{r}.y$ and $\vec{r}.z$ respectively. We refer to the first and second columns of a matrix R by $R.c_1$ and $R.c_2$ respectively.

The four edges of the rectangle are named from E1 to E4, as shown in the figure, where:

- E1 is the vertical edge that intersects with the positive portion of the X_A axis, with endpoints (v_1, v_2)
- E2 is the horizontal edge that intersects with the negative portion of the Y_A axis, with endpoints (v_2, v_3)
- E3 is the vertical edge that intersects with the negative portion of the X_A axis, with endpoints (v_3, v_4)
- E4 is the horizontal edge that intersects with the positive portion of the Y_A axis, with endpoints (v_4, v_1)

We refer to E1 and E4 as the positive edges, and E2 and E3 as the negative edges.

We say Rectangle A collides with Rectangle B at its Ex edge if the intersection area covers the largest portion of Ex as compared to the other edges, and call Ex the collision edge of A. We will not give a

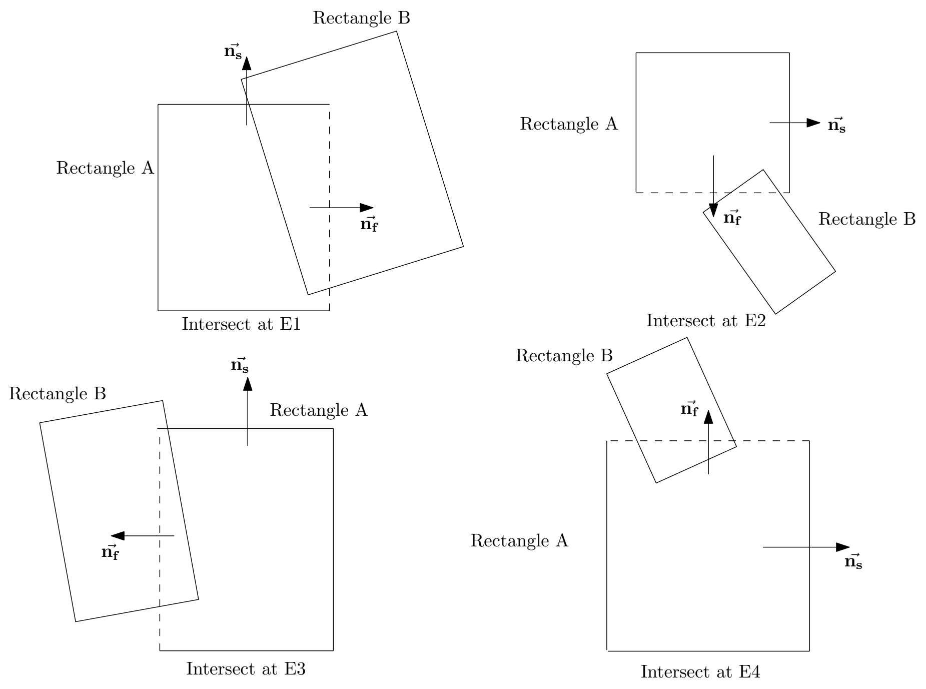

Figure 5: Intersect Edges

formal mathematics definition for this. Instead, please refer to the example in Figure 5 for an intuitive idea.

We denote by

- $\vec{n_f}$, the unit normal vector of the collision edge of A. It's equal to the directional unit vector of the axis that the collision edge intersects with.
- $\vec{n_s}$, the unit normal vector of the positive edge that the collision edge is adjacent to. It's equal to the directional unit vector of the axis with which this adjacent edge intersects.

Note that these two vectors can be easily computed by taking the appropriate column vectors from the rotation matrix (as discussed previously) and flipping them if necessary.

For an arbitary vector $\begin{pmatrix} x \\ y \end{pmatrix}$, we define its non-negative form as

$$\{\begin{pmatrix} x \\ y \end{pmatrix}\} = \begin{pmatrix} |x| \\ |y| \end{pmatrix} \tag{4}$$

We can similarly define the non-negative form of matrices.

For any pair of rectangles A and B, we would like to pick the collision edge of one of the rectangles as the reference edge, and the collision edge of the other rectangle as the incident edge, and think of the other rectangle as colliding on it with the inciden edge parallel to the reference edge. Figure 6 shows such an example. The dashed rectangle is what we approximate the other rectangle to be. The two contact points between rectangle A and B are thus approximated as the projections of the end points of the incident edge onto the reference edge, and clipping it within the end points of the reference edge if necessary.

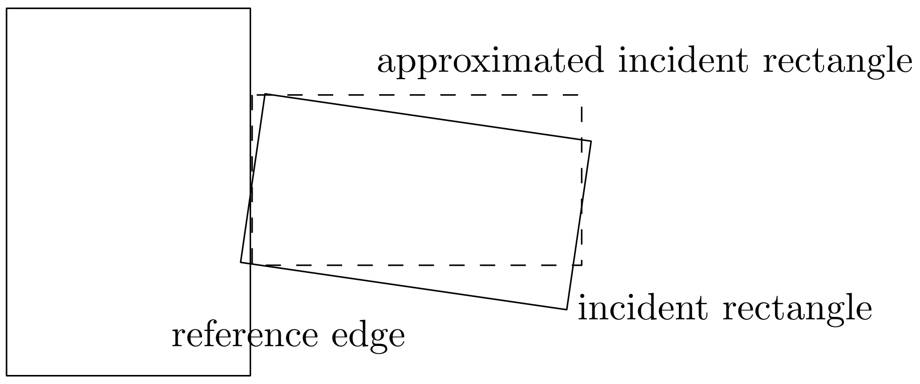

Figure 6: Approximated Collision

Next, we describe how we pick the reference edge and the incident edge. You can refer to Figure 7 for an intuitive idea of the rectangles A and B in the world coordinate system, and refer to Figure 8 for an idea of the rectangles in $\mathcal{R}_{\mathcal{A}}$.

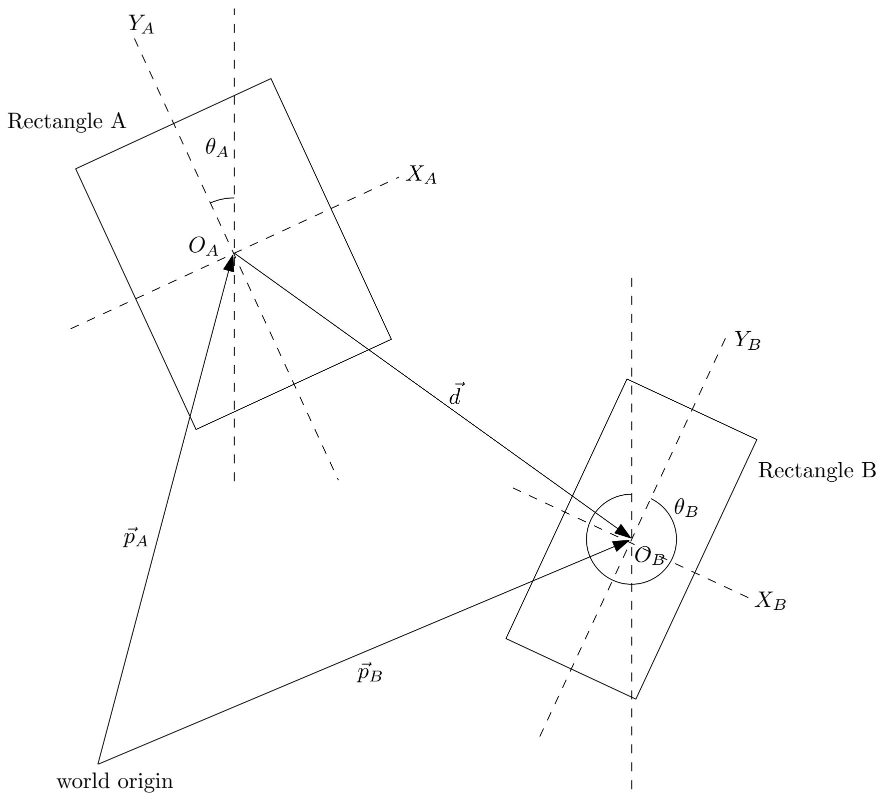

Figure 7: Two Rectangles in World Coordinate System

Let $\vec{p}_A$ and $\vec{p}_B$ be the position vectors of the centres of mass of rectangle A and B respectively, then $\vec{d} = \vec{p}_B - \vec{p}_A$ is the direction vector from O_A to O_B, and

- $\vec{d}_A = R_A^T \times \vec{d}$ is $\overrightarrow{O_A O_B}$ in the coordinate system $\mathcal{R}_A$.
- $\vec{d}_B = R_B^T \times \vec{d}$ is $\overrightarrow{O_A O_B}$ in the coordinate system $\mathcal{R}_B$.

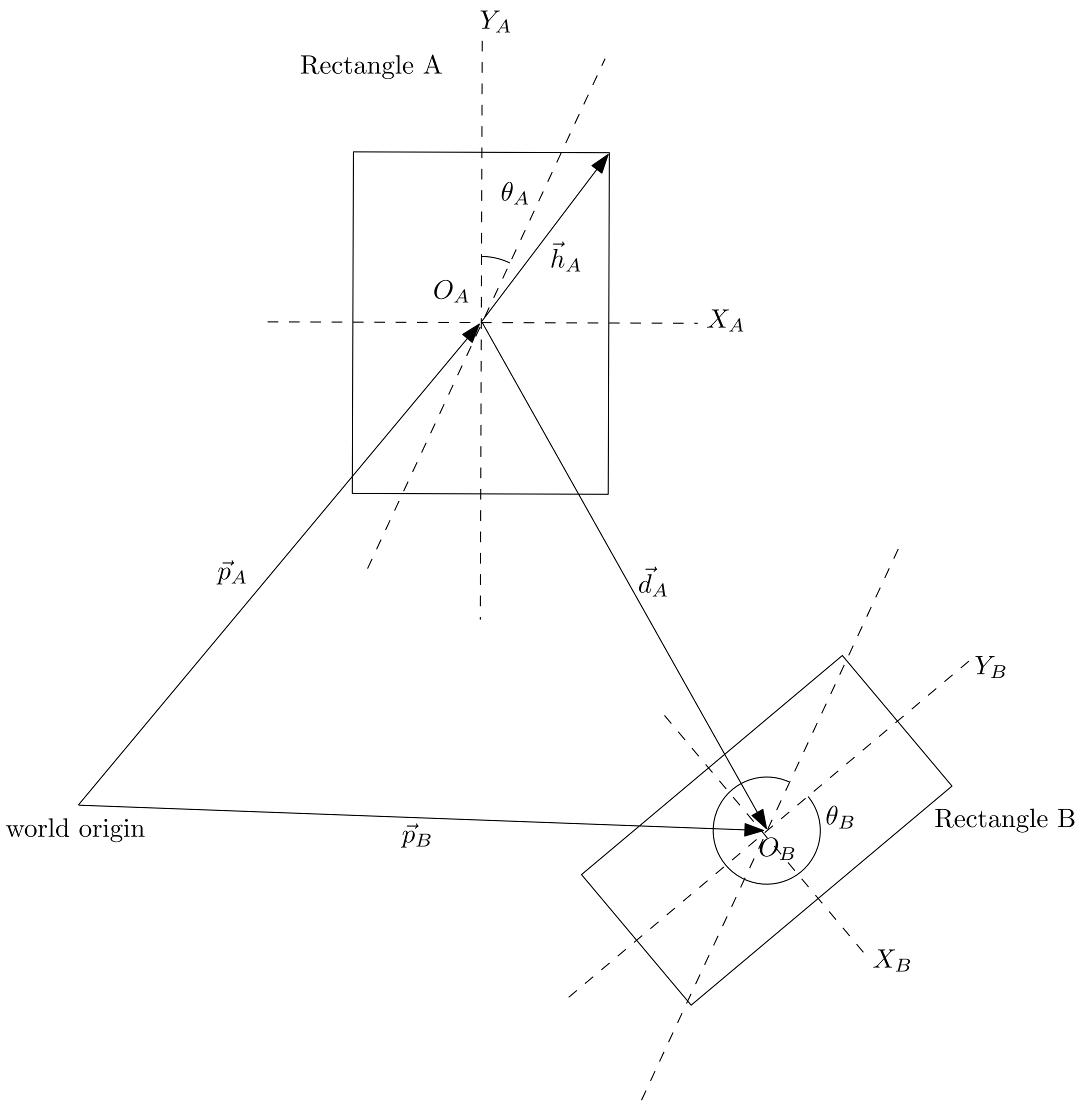

Figure 8: Two Rectangles in Coordinate System $\mathcal{R}_A$

Let

$$C = R_A^T \times R_B$$
$$\vec{f}_A = \{\vec{d}_A\} - \vec{h}_A - \{C\} \times \vec{h}_B$$
$$\vec{f}_B = \{\vec{d}_B\} - \vec{h}_B - \{C^T\} \times \vec{h}_A$$

For the two rectangles to be intersecting with each other, first of all the components of $\vec{f}_A$ and $\vec{f}_B$ must all be negative. Now let

- $f_{ax} = \vec{f}_A.x$, which is the average horizontal distance between the reference edge and the incident edge in $\mathcal{R_A}$ if a vertical edge of A is picked as the reference edge.
- $f_{ay} = \vec{f}_A.y$, which is the average vertical distance between the reference edge and the incident edge in $\mathcal{R_A}$ if a horizontal edge of A is picked as the reference edge.
- $f_{bx} = \vec{f}_B.x$, which is the average horizontal distance between the reference edge and the incident edge in $\mathcal{R_B}$ if a vertical edge of B is picked as the reference edge.

- $f_{by} = \vec{f_B}.y$, which is the average vertical distance between the reference edge and the incident edge in $\mathcal{R_B}$ if a horizontal edge of B is picked as the reference edge.

Of course, these are all approximate values.

Let i, j range over $\{ax, ay, bx, by\}$, we pick the f_i such that $\forall j, j \neq i \rightarrow f_i > f_j$, i.e. we pick the f_i with the smallest magnitude. Note thast these are all negative values, so we want the largest negative value.

Now depending on which i we picked, we do the following:

- for ax:

  If $\vec{d_A}.x > 0$, that means rectangle B is on the right hand side of rectangle A in $\mathcal{R_A}$. We pick E1 of A as the reference edge and set $\vec{\mathbf{n}}$ as $R_A.c_1$.

  Otherwise rectangle B is on the left hand side of rectangle A in $\mathcal{R_A}$. We pick E3 of A as the reference edge and set $\vec{\mathbf{n}}$ as $-R_A.c_1$.

  In addition, set:

$$\begin{aligned}
\vec{\mathbf{n_f}} &= \vec{\mathbf{n}} && (5)\\
D_f &= \vec{p}_A.\vec{\mathbf{n_f}} + \vec{h}_A.x && (6)\\
\vec{\mathbf{n_s}} &= R_A.c_2 && (7)\\
D_s &= \vec{p}_A.\vec{\mathbf{n_s}} && (8)\\
D_{neg} &= \vec{h}_A.y - D_s && (9)\\
D_{pos} &= \vec{h}_A.y + D_s && (10)
\end{aligned}$$

- for ay:

  If $\vec{d_A}.y > 0$, that means rectangle B is on the top of rectangle A in $\mathcal{R_A}$. We pick E4 of A as the reference edge and set $\vec{\mathbf{n}}$ as $R_A.c_2$.

  Otherwise rectangle B is on the bottom of rectangle A in $\mathcal{R_A}$. We pick E2 of A as the reference edge and set $\vec{\mathbf{n}}$ as $-R_A.c_2$.

  In addition, set:

$$\begin{aligned}
\vec{\mathbf{n_f}} &= \vec{\mathbf{n}} && (11)\\
D_f &= \vec{p}_A.\vec{\mathbf{n_f}} + \vec{h}_A.y && (12)\\
\vec{\mathbf{n_s}} &= R_A.c_1 && (13)\\
D_s &= \vec{p}_A.\vec{\mathbf{n_s}} && (14)\\
D_{neg} &= \vec{h}_A.x - D_s && (15)\\
D_{pos} &= \vec{h}_A.x + D_s && (16)
\end{aligned}$$

- for bx:

  If $\vec{d_B}.x > 0$, that means rectangle A is on the right hand side of rectangle B in $\mathcal{R_B}$. We pick E1 of B as the reference edge and set $\vec{\mathbf{n}}$ as $R_B.c_1$.

  Otherwise rectangle A is on the left hand side of rectangle B in $\mathcal{R_B}$. We pick E3 of B as the reference edge and set $\vec{\mathbf{n}}$ as $-R_B.c_1$.

In addition, set:

$$\begin{aligned}
\vec{\mathbf{n_f}} &= -\vec{\mathbf{n}} && (17)\\
D_f &= \vec{p}_B.\vec{\mathbf{n_f}} + \vec{h}_B.x && (18)\\
\vec{\mathbf{n_s}} &= R_B.c_2 && (19)\\
D_s &= \vec{p}_B.\vec{\mathbf{n_s}} && (20)\\
D_{neg} &= \vec{h}_B.y - D_s && (21)\\
D_{pos} &= \vec{h}_B.y + D_s && (22)
\end{aligned}$$

- for by:

  If $\vec{d}_B.y > 0$, that means rectangle A is on the top of rectangle B in $\mathcal{R}_\mathcal{B}$. We pick E4 of B as the reference edge and set $\vec{\mathbf{n}}$ as $R_B.c_2$.

  Otherwise rectangle A is on the bottom of rectangle B in $\mathcal{R}_\mathcal{B}$. We pick E2 of B as the reference edge and set $\vec{\mathbf{n}}$ for both contacts as $-R_B.c_2$.

  In addition, set:

$$\begin{aligned}
\vec{\mathbf{n_f}} &= -\vec{\mathbf{n}} && (23)\\
D_f &= \vec{p}_B.\vec{\mathbf{n_f}} + \vec{h}_B.y && (24)\\
\vec{\mathbf{n_s}} &= R_B.c_1 && (25)\\
D_s &= \vec{p}_B.\vec{\mathbf{n_s}} && (26)\\
D_{neg} &= \vec{h}_B.x - D_s && (27)\\
D_{pos} &= \vec{h}_B.x + D_s && (28)
\end{aligned}$$

where

- D_f is the distance between the world origin and the reference edge
- D_{pos} is the distance between the world origin and the positive edge that is adjacent to the reference edge
- D_{neg} is the distance between the world origin and the negative edge that is adjacent to the reference edge.

For an intuitive idea of these distances, you can refer to Figure 9, which shows them for the case when E1 of A is picked as the reference edge.

Now that we've got the reference edge, we need to find the incident edge. Denote by $\vec{v}_1$ and $\vec{v}_2$, the position vectors of the end points of the incident edge.

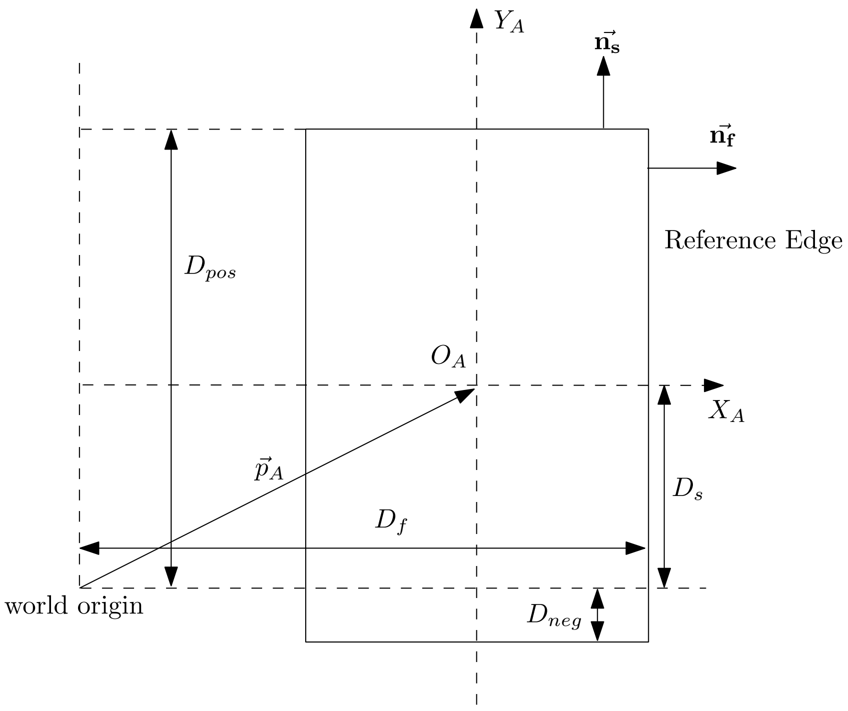

Figure 9: Reference Edge $\mathcal{R}_A$

Depending on which i we've picked, we do:

- For ax and ay, set

  $\vec{\mathbf{n_i}} = -R_B^T\vec{\mathbf{n_f}}$, which is the incident unit directional vector in $\mathcal{R_B}$
  $\vec{p} = \vec{p}_B$, which is the position vector of the centre of mass of B.
  $R = R_B$
  $\vec{h} = \vec{h}_B$

- For bx and by, set

  $\vec{\mathbf{n_i}} = -R_A^T\vec{\mathbf{n_f}}$, which is the incident unit directional vector in $\mathcal{R_A}$
  $\vec{p} = \vec{p}_A$, which is the position vector of the centre of mass of A.
  $R = R_A$
  $\vec{h} = \vec{h}_A$

Then:

- If $\{\vec{\mathbf{n_i}}\}.x > \{\vec{\mathbf{n_i}}\}.y$ and $\{\vec{\mathbf{n_i}}\}.x > 0$ then

$$\vec{v}_1 = \vec{p} + R\begin{pmatrix}\vec{h}.x \\ -\vec{h}.y\end{pmatrix}$$
$$\vec{v}_2 = \vec{p} + R\begin{pmatrix}\vec{h}.x \\ \vec{h}.y\end{pmatrix}$$

- If $\{\vec{\mathbf{n_i}}\}.x > \{\vec{\mathbf{n_i}}\}.y$ and $\{\vec{\mathbf{n_i}}\}.x \le 0$ then

$$\vec{v}_1 = \vec{p} + R\begin{pmatrix} -\vec{h}.x \\ \vec{h}.y \end{pmatrix}$$
$$\vec{v}_2 = \vec{p} + R\begin{pmatrix} -\vec{h}.x \\ -\vec{h}.y \end{pmatrix}$$

- If $\{\vec{\mathbf{n_i}}\}.x \le \{\vec{\mathbf{n_i}}\}.y$ and $\{\vec{\mathbf{n_i}}\}.y > 0$ then

$$\vec{v}_1 = \vec{p} + R\begin{pmatrix} \vec{h}.x \\ \vec{h}.y \end{pmatrix}$$
$$\vec{v}_2 = \vec{p} + R\begin{pmatrix} -\vec{h}.x \\ \vec{h}.y \end{pmatrix}$$

- If $\{\vec{\mathbf{n_i}}\}.x \le \{\vec{\mathbf{n_i}}\}.y$ and $\{\vec{\mathbf{n_i}}\}.y \le 0$ then

$$\vec{v}_1 = \vec{p} + R\begin{pmatrix} -\vec{h}.x \\ -\vec{h}.y \end{pmatrix}$$
$$\vec{v}_2 = \vec{p} + R\begin{pmatrix} \vec{h}.x \\ -\vec{h}.y \end{pmatrix}$$

Now we need to clip the incident edge to make it fall within the boundaries of the edges adjacent to the reference edge. Please take a look at Figure 10, which shows the situation where the reference edge is E1 of A.

We need to make sure that the end points of the incident edge, v_1 and v_2, fall in between the two clipping lines, which are basically extension lines from the two edges adjacent to the reference edge.

- First Clip, with Clipping Line 1 and the direction of the clipping represented by $-\vec{\mathbf{n_s}}$. Compute:

$$dist_1 = -\vec{\mathbf{n_s}}.\vec{v}_1 - D_{neg} \quad (29)$$
$$dist_2 = -\vec{\mathbf{n_s}}.\vec{v}_2 - D_{neg} \quad (30)$$

  which are distances of the points v_1 and v_2 to Clipping Line 1. If both $dist_1$ and $dist_2$ are negative, that means the two points are already above the Clipping Line 1, in which case we don't need to do anything. Points after clipping are:

$$\vec{v}_1' = \vec{v}_1 \quad (31)$$
$$\vec{v}_2' = \vec{v}_2 \quad (32)$$

  If both $dist_1$ and $dist_2$ are positive, that means the rectangles A and B simply do not collide with each other. We skip this pair of bodies.

  Otherwise, $dist_1$ is negative and $dist_2$ is positive. We "clip" v_2 to fall on Clipping Line 1 by interpolating the segment $\overrightarrow{v_1v_2}$. Set

$$\vec{v}_1' = \vec{v}_1 \quad (33)$$
$$\vec{v}_2' = \vec{v}_1 + \frac{dist_1}{dist_1 - dist_2}(\vec{v}_2 - \vec{v}_1) \quad (34)$$

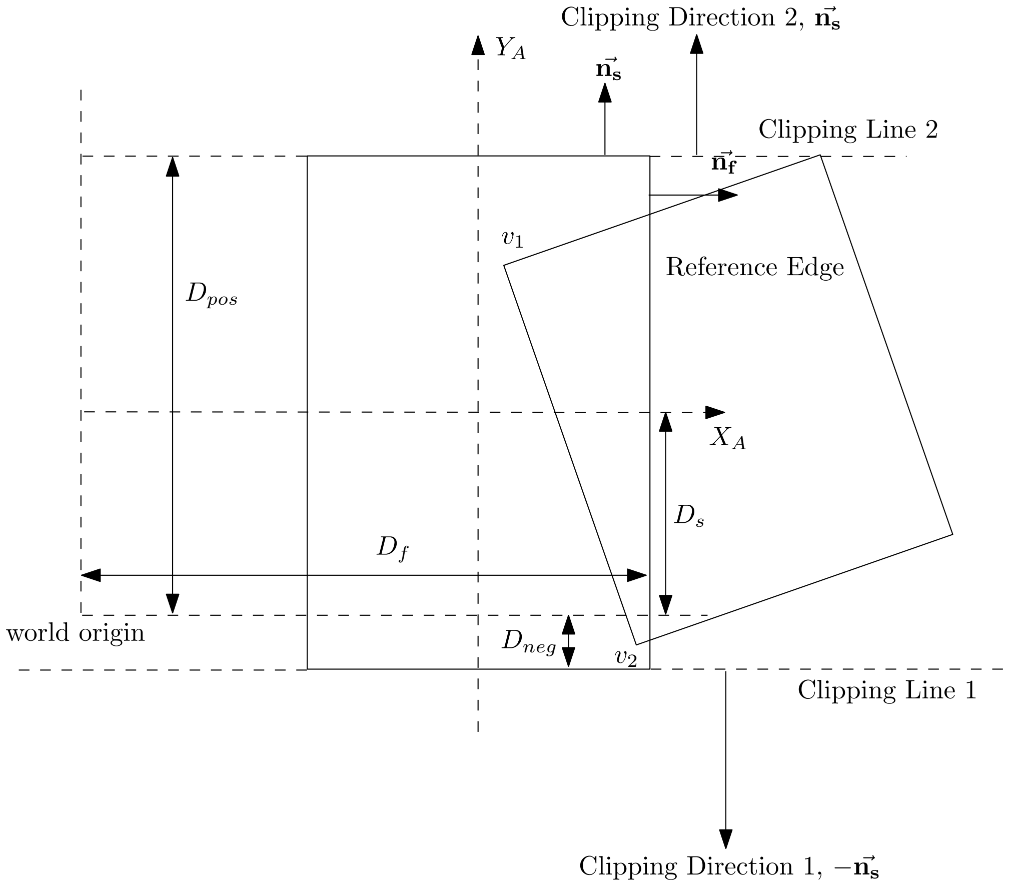

Figure 10: Clipping $\mathcal{R}_A$

- Second Clip, Clipping v_1' and v_2' to Clipping Line 2 with the direction of the clipping represented by $\vec{\mathbf{n_s}}$. Compute:

$$\begin{aligned} dist_1 &= \vec{\mathbf{n_s}}.\vec{v_1'} - D_{pos} && (35)\\ dist_2 &= \vec{\mathbf{n_s}}.\vec{v_2'} - D_{pos} && (36)\end{aligned}$$

which are distances of the points v_1' and v_2' to Clipping Line 2. If both $dist_1$ and $dist_2$ are negative, that means the two points are already below the Clipping Line 2, in which case we don't need to do anything. Points after clipping are:

$$\begin{aligned} \vec{v_1''} &= \vec{v_1'} && (37)\\ \vec{v_2''} &= \vec{v_2'} && (38)\end{aligned}$$

If both $dist_1$ and $dist_2$ are positive, that means the rectangles A and B simply do not collide with each other. We skip this pair of bodies.

Otherwise, $dist_1$ is negative and $dist_2$ is positive. We "clip" v_2' to fall on Clipping Line 2 by interpolating the segment $\overrightarrow{v_1'v_2'}$. Set

$$\begin{aligned} \vec{v_1''} &= \vec{v_1'} && (39)\\ \vec{v_2''} &= \vec{v_1'} + \frac{dist_1}{dist_1 - dist_2}(\vec{v_2'} - \vec{v_1'}) && (40)\end{aligned}$$

Now the end points of the incident edge after clipping are $\vec{v}_1''$ and $\vec{v}_2''$. Note that, it's possible to end up with fewer than two points after the clipping, due to roundoff error of floating point calculation. In that case, we'll skip the pair of bodies A and B. We now project these two points onto the reference edge to obtain the contact points $\vec{c}_1$ and $\vec{c}_2$.

- For $\vec{c}_1$,

$$\begin{aligned} s \text{ (separation)} &= \vec{\mathbf{n_f}}.\vec{v}_1 - D_f && (41)\\ \vec{c}_1 &= \vec{v}_1 - s \times \vec{\mathbf{n_f}} && (42) \end{aligned}$$

- For $\vec{c}_2$,

$$\begin{aligned} s \text{ (separation)} &= \vec{\mathbf{n_f}}.\vec{v}_2 - D_f && (43)\\ \vec{c}_2 &= \vec{v}_2 - s \times \vec{\mathbf{n_f}} && (44) \end{aligned}$$

Both these contact points share the same normal $\vec{\mathbf{n}}$ and tangent $\vec{\mathbf{n}} \times \vec{\mathbf{z}}$, where $\vec{\mathbf{z}}$ is the positive unit directional vector of the z-axis.

Step 3. Apply Impulses at Contact Points

Next, we discuss how to apply the impulses resulted from collisions between two rectangular bodies A and B at a contact point with position vector $\vec{c}$. We adopt the usual notations, where O_A, O_B are the centres of mass of rectangle A and B respectively, with position vectors $\vec{p}_A$ and $\vec{p}_B$. m_A and m_B are the mass of the two bodies; I_A and I_B are the moment of inertia of the two bodies.

The direction vectors of the contact point from the centres of mass are:

$$\begin{aligned} \vec{r}_A &= \vec{c} - \vec{p}_A && (45)\\ \vec{r}_B &= \vec{c} - \vec{p}_B && (46) \end{aligned}$$

Suppose the velocities of A and B are $\vec{v}_A$ and $\vec{v}_B$, and the angular velocities of them are $\vec{\omega}_A$ and $\vec{\omega}_B$ respectively, we compute the velocities of A and B at the contact point by:

$$\begin{aligned} \vec{u}_A &= \vec{v}_A + \vec{\omega}_A \times \vec{r}_A && (47)\\ \vec{u}_B &= \vec{v}_B + \vec{\omega}_B \times \vec{r}_B && (48) \end{aligned}$$

Note that these are **different** from $\vec{v}_A$ and $\vec{v}_B$. These are velocities of a point only, not the entire body. Each body must have a point at the position of the contact point, thus we compute two velocities, one for each. We proceed to compute the relative velocity of the contact point.

$$\vec{u} = \vec{u}_B - \vec{u}_A \qquad (49)$$

Suppose the unit normal and tangent vectors at the contact point are $\vec{\mathbf{n}}$ and $\vec{\mathbf{t}}$ respectively, then the normal and tangential components of the relative velocity are:

$$\begin{aligned} cu_n &= \vec{u}.\vec{\mathbf{n}} && (50)\\ u_t &= \vec{u}.\vec{\mathbf{t}} && (51) \end{aligned}$$

Next, we compute the normal and tangential mass at the contact point:

$$mass_n = \frac{1}{\dfrac{1}{m_A}+\dfrac{1}{m_B}+\dfrac{\vec{r}_A.\vec{r}_A-(\vec{r}_A.\vec{\mathbf{n}})^2}{I_A}+\dfrac{\vec{r}_B.\vec{r}_B-(\vec{r}_B.\vec{\mathbf{n}})^2}{I_B}} \quad (52)$$

$$mass_t = \frac{1}{\dfrac{1}{m_A}+\dfrac{1}{m_B}+\dfrac{\vec{r}_A.\vec{r}_A-(\vec{r}_A.\vec{\mathbf{t}})^2}{I_A}+\dfrac{\vec{r}_B.\vec{r}_B-(\vec{r}_B.\vec{\mathbf{t}})^2}{I_B}} \quad (53)$$

With these, we can compute the normal and tangential impulses by

$$\vec{P}_n = mass_n \times u_n \times \vec{\mathbf{n}} \quad (54)$$

$$\vec{P}_t = mass_t \times u_t \times \vec{\mathbf{t}} \quad (55)$$

You may wonder why we take all the trouble to break the impulse into normal and tangential components; isn't working with a single impulse vector more convenient? Well, the reason is that tangential impulses are subject to friction. Suppose the coefficients of friction for bodies A and B are μ_A and μ_B respectively, then the maximum magnitude for the tangential impulse is $P_{tmax} = \mu_A\mu_B\|\vec{P}_n\|$. We reset $\vec{P}_t$ to:

$$\vec{P}_t = \max(-P_{tmax}, \min(\|\vec{P}_t\|, P_{tmax})) \quad (56)$$

The rest are standard stuff, we compute the new velocities $\vec{v}'_A$, $\vec{v}'_B$, and new angular velocities $\vec{\omega}'_A$, $\vec{\omega}'_B$ by

$$\vec{v}'_A = \vec{v}_A + \frac{1}{m_A}(\vec{P}_n + \vec{P}_t) \quad (57)$$

$$\vec{v}'_B = \vec{v}_B - \frac{1}{m_B}(\vec{P}_n + \vec{P}_t) \quad (58)$$

$$\vec{\omega}'_A = \vec{\omega}_A + \frac{1}{I_A}\vec{r}_A \times (\vec{P}_n + \vec{P}_t) \quad (59)$$

$$\vec{\omega}'_B = \vec{\omega}_B - \frac{1}{I_B}\vec{r}_B \times (\vec{P}_n + \vec{P}_t) \quad (60)$$

For each contact point, we apply the impulses in the above manner. The completion of the processing of all contact points is called an iteration. Doing one iteration of this process is seldom sufficient. To improve the accuracy of the simulation, we suggest that you perform at least 8 to 10 iterations of this process for each stepping of the world.

Step 4. Moving The Bodies

After completing all the iterations of the impulse process, we have the velocities and angular velocities of the bodies. We assume these are the average velocities and angular velocities during the small time interval in a stepping. This is of course an approximation, but should work reasonably well. We thus compute the new status of the bodies by:

$$\vec{p}'_A = \vec{p}_A + dt \times \vec{v}'_A \quad (61)$$

$$\vec{p}'_B = \vec{p}_B + dt \times \vec{v}'_B \quad (62)$$

$$\theta'_A = \theta_A + dt \times \vec{\omega}'_A \quad (63)$$

$$\theta'_B = \theta_B + dt \times \vec{\omega}'_B \quad (64)$$

Note the abuse of the notation here; by $\vec{\omega}'_A$, we actually mean the third component of $\vec{\omega}'_A$, i.e. if $\vec{\omega}'_A = \begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix}$, we mean the value z. Since every rotation we perform is around some axis that is perpendicular to the x-y-plane, the angular velocity vector is necessarily of the form $\begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix}$.

After updating the status of the bodies, one stepping of the physics engine is complete.

Position Correction

Now that you've got the physics engine up and running, you may notice that the bodies often intersect too much. That's because we are running a discrete simulation here; when the velocities of the bodies are too big, in one stepping interval they can already move too much into each other. There are many ways to deal with this problem and here we suggest two methods; one difficult approach and one easy approach.

The Difficult Solution

We can try to find a smaller time interval dt' such that after dt', the status of the world satisfies two conditions:

- for each pair of bodies A and B, A and B are either not intersecting each other, or their intersection is small enough that it is acceptable.
- there are at least a pair of bodies that are intersecting each other.

We can find such a dt' by binary search on the interval $(0, dt)$. After that, we do a stepping of dt', then continue with a stepping of $dt - dt'$.

The Easy Solution

The easy solution is kind of "cheating". When two bodies are intersecting each other, we give a bias to the normal impulse to help nudge the two bodies apart. Remember how we compute the normal impulse?

$$\vec{P}_n = mass_n \times u_n \times \vec{\mathbf{n}} \tag{65}$$

We make a little change here:

$$\vec{P}_n = mass_n \times (u_n - bias) \times \vec{\mathbf{n}} \tag{66}$$

You can put your creativity to good use to conjure up the $bias$. If you do not want to do that, we have a suggestion. Remember that for each contact, we compute the separation between two bodies, which is a negative value if the two bodies intersect each other. The more the two bodies move into each other, the larger the magnitude of separation. We set a limit on the magnitude of separation that we can tolerate, say κ. We then set the $bias$ only when $\kappa < |separation|$. Intuitively, $bias \propto (\kappa + separation)$ and $bias \propto \frac{1}{dt}$. Thus, one way to set the $bias$ is:

$$bias = |\frac{\epsilon}{dt}(\kappa + separation)|$$

where ϵ is a constant factor which you can tweek to achieve satisfactory results. In the sample app implemented by the teaching staff, κ is set to 0.01 and ϵ is set to 0.2

We also need to modify how we pick the reference edge, to favor large edges. Remember that we compute:

$$\begin{aligned} f_{ax} &= \vec{f_A}.x && (67) \\ f_{ay} &= \vec{f_A}.y && (68) \\ f_{bx} &= \vec{f_B}.x && (69) \\ f_{by} &= \vec{f_B}.y && (70) \end{aligned}$$

Now we compute another set of values to help us pick the optimal reference edge:

$$\begin{aligned} \delta_{ax} &= f_{ax} - \kappa(\vec{h}_A.x) && (71) \\ \delta_{ay} &= f_{ay} - \kappa(\vec{h}_A.y) && (72) \\ \delta_{bx} &= f_{bx} - \kappa(\vec{h}_B.x) && (73) \\ \delta_{by} &= f_{by} - \kappa(\vec{h}_B.y) && (74) \end{aligned}$$

Let i, j range over $\{ax, ay, bx, by\}$, we pick the δ_i such that

$$\forall j, j \neq i \rightarrow \delta_i > \eta f_j \tag{75}$$

where η is constant weight we give to the separation. For the example in the introduction, η is set to 0.95. The rest are the same.