My design for the fallingBricks app is as follows:

first of all, I need a class called PERectagle: this is my model in the MVC design pattern. This class stores the essentially information for one brick. These information includes center(coordinate in the superView's coordinate system), dimension of the rectangle(width and height), mass, moment of inertia (it should be calculated from the user's input data instead of assigned directly), friction and restitution coefficient, velocity(a 2-D vector) and angular velocity (a CGFloat value), identity(to distinguish between the walls and game bricks), rotation, rotation matrix(a 2*2 matrix calculated from rotation), hVector(a 2-D vector pointing from the rectangle's center to its up-right corner), the UIColr, and most of all, a delegate. These data will be used for display and simulation later.

Then I need a class called PERectangleView Controller, which serves as child viewController of the mainViewController. Each instance of this class will have a strict one to one relationship with an instance of the PERectangle class, because PEractangle is the model for this view controller. It is responsible for updating the position of each rectangle on the screen, thus it conforms a update position delegate, and will assign itself to its model as th delegate, later in the simulating stage, the model will use this delegate to update position.

MyWorld is the simulated world where everything happens. It will loop through all the objects added in this world, use an instance of collisionDetector to find all the collision points and also apply impulse to corresponding objects to change the velocity and angular velocity.

CollisionDetector checks if two objects in the simulated world collide or not, and if they do collide, find the contact points and stores them in an array. After finding all the contact points, MyWorld instance will call the apply impulse method to make required changes.

ContactPoint is just the class needed to store necessary information about a contact point between two colliding rectangles.

MainViewController is just required to perform the standard stuff: add objects and call appropriate method, in our case, the run method in myWorld instance.

The advantage of this design is:

1. It is efficient and clear. I create a viewController for each game brick. Alternatively I can just create an UIView for each game brick and let the mainViewController to control all the UIViews, but it is not efficient because as I have tested, each time removing all the views and then re-add the adjusted views is very expensive. Also if I want to add objects of other shape, the code will be really messy. Creating a viewController for each object, otherwise, eliminates this problem, because each viewController just need to deal with a particular task.

2. It can be easily modified. Everything will almost remains the same if I want to add objects of other type. The only things needed to be added is: the model and viewController for this particular object type, and the collision detection method and the applying impulse method.

3. I t conforms strictly to the MVC design pattern, which separates concerns for different parts of the program.

Testing:

Because complexity of calculating the velocity of colliding objects, I cannot come up with a good solution or strategy of testing the apply impulse method in the CollisionDetector method. Also because there is a UIAccelerometer object in myWorld class, in Unit test it cannot be compiled, so I also neglect it from Unit test. ALL THIS WILL BE DOWN BY BLACK-BOX TESTING .

Black-box testing:

-test gravity:

use ios simulator (thus no uiacclerometer), add some objects, and adjust the default gravity coefficient, see if the objects fall at different velocity.

change the number and position of objects added, and repeat the above step;

use ipad (thus has uiacclerometer), add some objects, and adjust the gravity scaling coefficient, see if the objects fall at different velocity. Tilt device into different direction

change the number and position of objects added, and repeat the above step;

-testCollision

add two objects, use ipad, set the size and mass of the objects, and tilt device into different directions and see if the collision works properly.

adjust the object size and mass, and see repeat the above step.

adjust the restitution coefficient and bias value, repeat the first step and see the effect.

add different number of objects (4, 6, 8), and repeat the above three steps.

-testFriction

repeat the testCollision strategy, but with different set of friction coefficient(0.01, 0.1, 0.5, 1), and observe changes. Note that when friction is large enough, say 1.0, when the objects hit ground, it would appear that they are "adhesive".

glass-box testing:

as the discussion at the begin of this testing section, it is really not easy to test this app with unit test, because it is so "GUI-Oriented", and the unit test framework does not support some essential element in this program, like UIAccelerometer. Also the formulas are too complex, and I cannot honestly provide the theoretical values for verification, so I just test very basic stuff.

the tested terms are clear in the unit test files. Basically my testing focus on initialization.

Also the detect collision method in the CollisionDetector class is tested.

I spent around 60 hours on this problem set.

Well if I can make less typos, I can finish it in 35 hours.....

The TAs have done an outstanding job, I find the forum very helpful.