extracted from **http://www.catsinthesky.com/blog/article/2012/03/5/farseer-physics-box2d-and-unity-part-1**

Farseer Physics (Box2D) and Unity (Part #1 - The Setup)

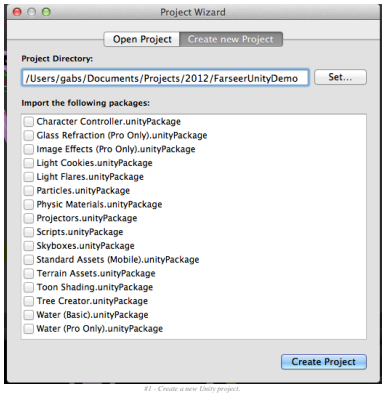
posted by gabs 1 Mar 23rd, 00:48

Part #1 | Part #2 | Part #3 | Part #4 | .unitypackage | GitHub

You're probably wondering why would you use a ported physics engine if Unity is shipped with a version of Physx, right? Well, Unity's version of Physx is not bad for 3d games and most 2d games, but when you have a physics driven 2d game, you need to choose between using Physx with tons of workarounds and hacks or implementing a robust 2d physics engine.

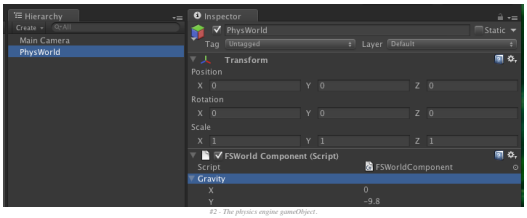
For example, there is no easy way to implement buoyancy in Physx. The most efficient way to do it is by attaching a mesh to your physics object (that's game over for 2d) and run a custom script that calculates the submerged area based on the attached mesh. The script will then add an impulse to simulate buoyancy. Imagine having to model 3d meshes for every object on your 2d game just to simulate buoyancy. A lot of time wasted and the mesh will never be used for rendering (the proper way a mesh should be used, anyway). With Farseer/Box2d, this is just a single line of code!

Farseer Physics is mostly a C# Box2D. I could say that it's Box2D with steroids regarding how fast you can make things happen with it. There are factory classes that do most of the work for you when you need basic shapes. To begin with our demonstration, grab this Unity package (Unity 3.5). I ported the core and I made some components that we will use in our scene objects. **Create a new Unity3D project (#1).**



#1 - Create a new Unity project.

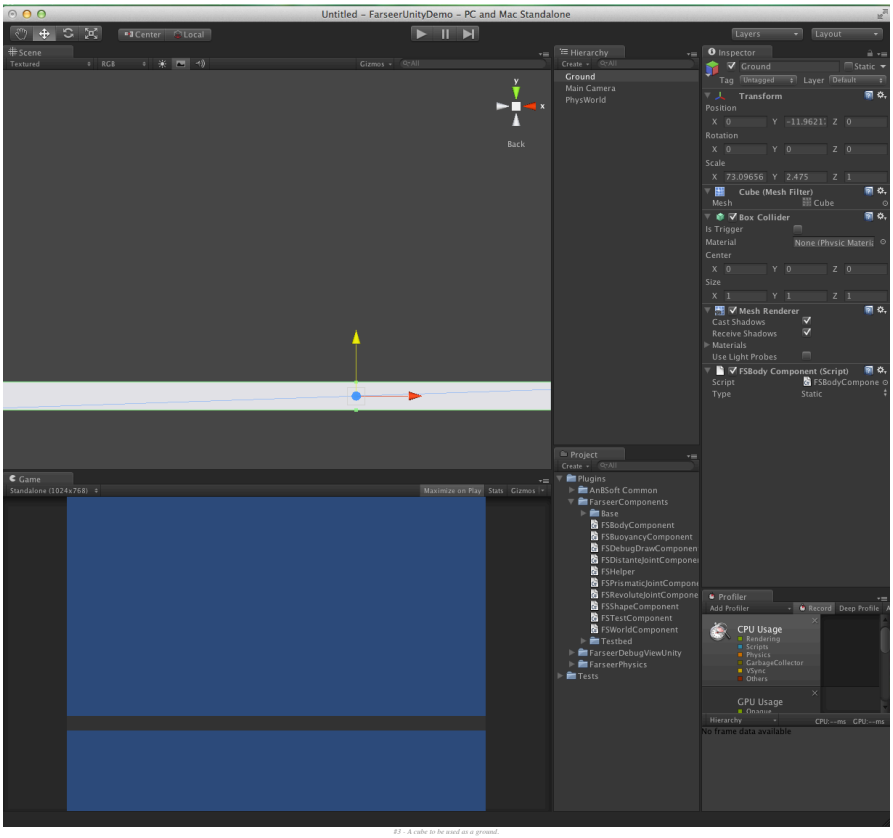
Import the package you just downloaded. Now, create a new scene. Set the camera to orthographic (I use size 27) and create a new empty gameObject. Attach the FSWorldComponent to this new gameObject. This will be what runs the physics engine (#2).



#2 - The physics engine gameObject.

The default gravity is negative because in unity, positive Y means UP.

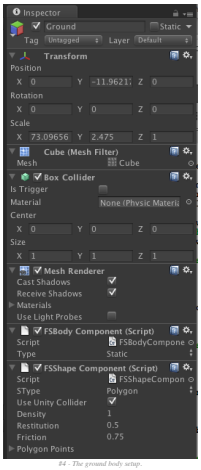
Now create a basic Cube, scale it enough because we will use it as a ground, now attach the FSBodyComponent to it (#3). Set the body type to static.



#3 - A cube to be used as a ground.

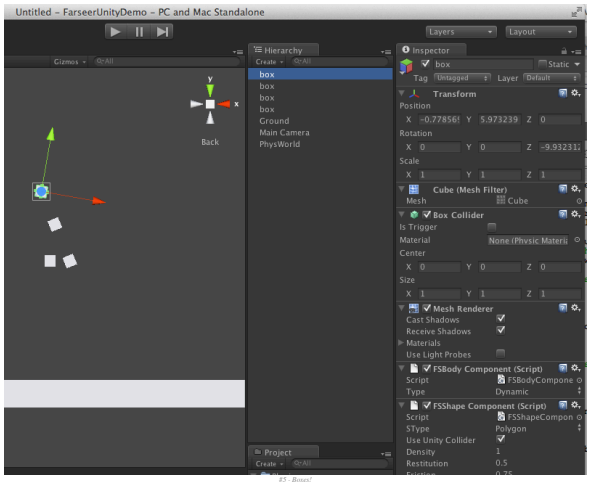
Now, there are three ways of setting the shape(s) for a physics body:
1 - Attach the FSShapeComponent directly to the gameObject with the FSBodyComponent and use Unity colliders to generate the shape;
2 - Create child game objects and attach colliders and FSShapeComponents to have a body with multiple fixtures (Using Unity colliders to define shapes);
3 - Do #2 but with custom shapes (we will get to this part);

For now, just drag the FSShapeComponent directly to the body's game object, and make sure its type is "Polygon" and that the "Use Unity collider" is checked (#4).



#4 - The ground body setup.

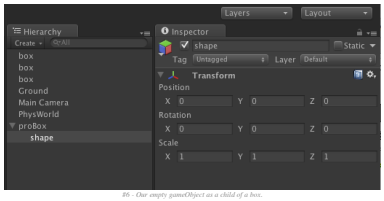
Duplicate the ground gameObject, change the scale, rename if to "box" (or anything but ground) then change the body type to "Dynamic". Duplicate this box several times and position them in your scene (#5).



#5 - Boxes!

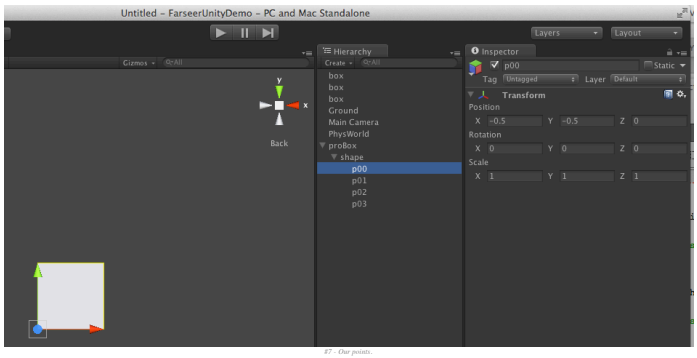
You can press play now to check if everything is working properly. Also, if you have the Pro version you can attach the **FSDebugDrawComponent** to the camera.

Now, in one of your duplicated boxes, let's try the other method of generating shapes. **Delete the FSShapeComponent from its gameObject.** Create a new gameObject, then make it a child of your box gameObject. Reset the position, rotation and scale. (#6)

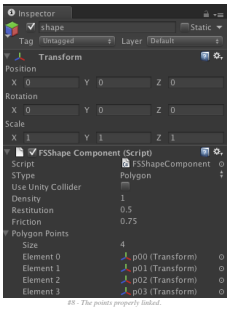


#6 - Our empty gameObject as a child of a box.

Attach the FSShapeComponent to this gameObject. Uncheck the "Use Unity collider" checkbox. We will make our shape by linking empty gameObjects as points. Arrange 4 empty gameObjects (children of this shape gameObject), name them (P00, P01, P02, P03) (#7). The shape algorythm creates the shapes via CCW (counter-clockwise), so make sure that your points are at the right order. After this, drag the points gameObjects (in order) to the Polygon Points array (of the parent's FSShapeComponent) (#8).



#7 - Our points.



#4 : The points properly linked.

With this method of defining shapes, a single body can have multiple shapes, each with independent properties (friction, restitution and density), and they don't need to be rectangles and circles, any convex polygon will do!

That's it for the first part of this tutorial. The next part will be about joints. Feel free to check the two scenes included in this package.