Tutorial – Linear Force and Momentum

Introduction and Objective

Over the next few tutorials we are going to produce our own physics engine from scratch. This will help you to understand the principles and mathematics being taught in the lectures.

Our engine won't be able to compete with the commercial engines on the market but it will teach you some valuable lessons about classical mechanics and optimization.

Required Resources

You are provided with the 'BootStrap' project which is a small project of pre-existing code that allows you to render simple geometric shapes.

All submissions are to adhere to the coding standards found in the PDF linked below and will not be marked otherwise.

http://aieportal.aie.edu.au/pluginfile.php/36727/mod_resource/content/0/AIE%20C%20%20C%2B%2B%20Coding%20Standards.pdf

Process

In this first tutorial you will

- Build a simple framework to allow you to spawn objects into a scene
- Add an update function to allow objects to be updated.
- Remove objects from the scene
- Talk about the main parts of a physics engine

This tutorial is **not going to go provide you with implementation detail**. Instead it provides a suggested framework for a physics engine which you will flesh out. You are free, encouraged in fact, to make design choices in your code but you must have:

1) A physics scene class which is responsible for control of your physics scene. It will contain:
   a) Variables which effect all the objects in the scene such as gravity
   b) A list of all the objects which make up the scene. From now on we'll refer to this objects as "actors"
   c) Functions to add and remove actors from the scene
   d) An update function which updates the state of all the actors in the scene
   e) A render function which renders the objects. For simplicity you can use the 2D Gizmos

2) Classes to represent the various types of actors which make up your physics scene

1) The Physics Scene Class

We'll begin by briefly looking at the physics scene class. This is quite a simple class whose function is to keep all the physics functionality together in one place

The class definition will look something like this:

```
Class PhysicScene
{
public:
	void AddActor(PhysicsObject* a_actor);
	void RemoveActor(PhysicsObject* a_actor);
	void GetActors() const;
	void Update();
	void DebugScene();
	void Draw();
	void SetGravity(float a_gravity);
	float GetGravity() const;
	void SetTimeStep(float a_timeStep);
	float GetTimeStep() const;

private:
	std::vector<PhysicsObject*> m_actors;
	float m_gravity;
	float m_timeStep;
```

We are using vector to store a list of pointers to actors. Don't forget the appropriate includes in order to use this. The update() and draw() functions simply iterate over all the actors in the vector calling the appropriate function.

You will need to provide the function bodies for these functions.

This is not the only way to set up your scene. Use the check list on the previous page to make sure you have everything you need covered and plan out how you will process your scene in advance.

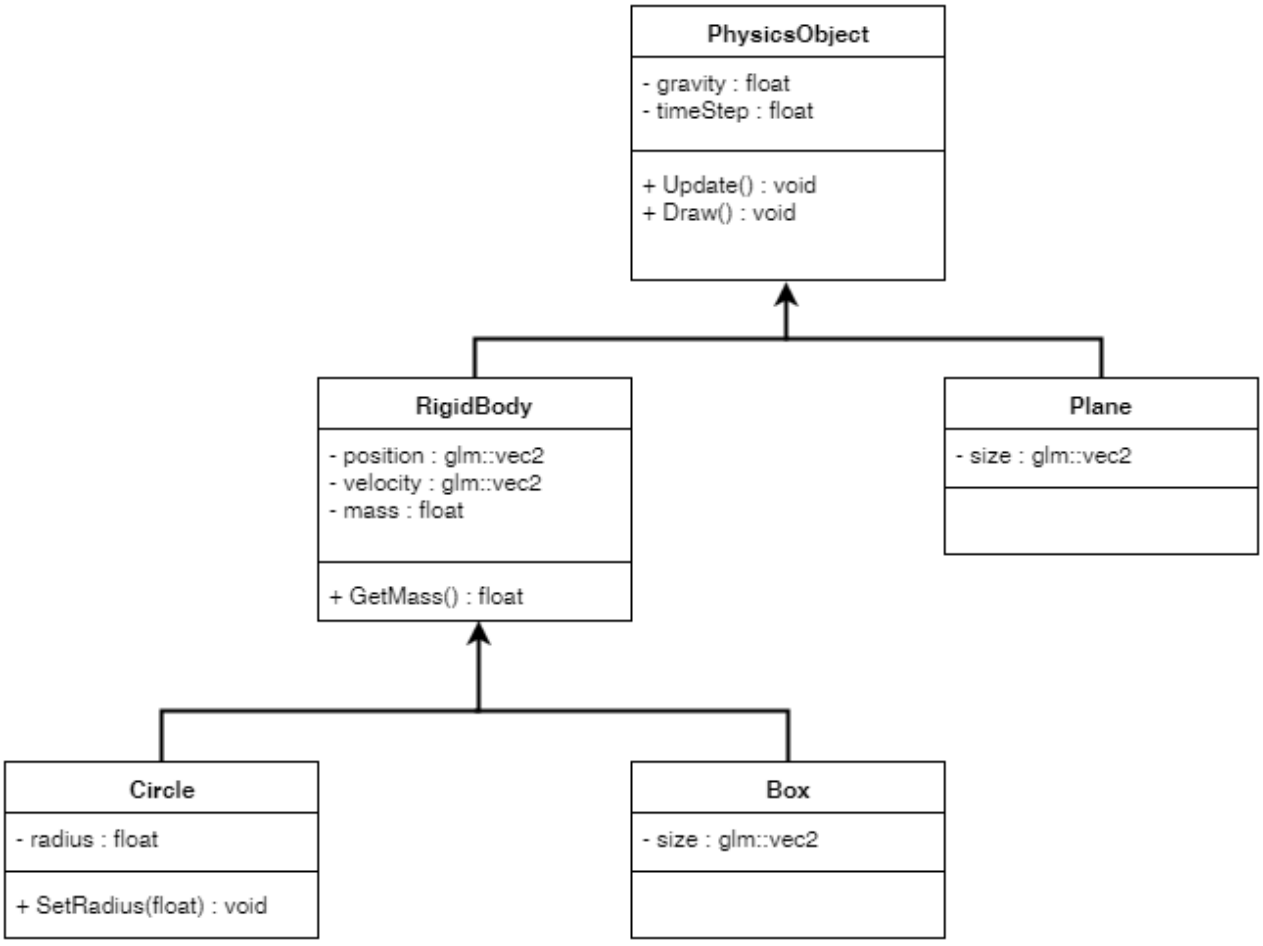
2) Classes to represent the various types of actors which make up your physics scene

You now need to set up the class hierarchy and structure of the objects that are going to populate the scene.

For our purposes we can assume that each physics actor in the scene has a single collision volume and that there are initially only three types:

- Plane
- Sphere
- Box

The following UML diagram illustrates the basic actor class hierarchy but is incomplete. **You need to produce your own diagram over the course of these tutorials to describe your complete system.**



Physics Object

Note how everything is derived from a single base class PhysicsObject. This is so we can iterate through a single list when updating and adding gizmos to our scene and also so that we can write our collision detection routines in a neater form later on. The base class contains a single variable: ShapeID. This is used for collision detection in a later tutorial. The code will be neater if you use an enumerated type for this.

Your definition for the base class will look something like this:

```
enum ShapeType
{
	PLANE = 0,
	SPHERE = 1,
	BOX = 2		/* 3 states */
};

class PhysicsObject
{
public:
	virtual void Update(float s_gravity, float a_timeStep);
	virtual void Debug();
	virtual void Draw(aie::Renderer2D*) = 0;
	virtual void ResetPosition();
	virtual void ApplyForce(glm::vec2 a_force);
	virtual void ApplyForceToActor(RigidBody* a_actor, glm::vec2 a_force);
	ShapeType GetShapeID() const;
	Void SetShapeID(ShapeType a_shapeID);

private:
	ShapeType shapeID;

};
```

Note how update, debug and draw are pure virtual functions. These functions are specific to the derived classes we are going to create and we don't want to instantiate the physics object because it doesn't have enough functionality for a real object in our world anyway. Therefore it makes sense for it to be a virtual class. We could put properties for position and orientation in the base class but that makes it harder to derive a plane class because planes don't have those properties. The debug function is useful for printing debug information to the console.

You will need to give some thought on how you are going to render your objects. Now is the time to look at the Application2D example and thinking about how you will handle this with the class hierarchy we have here. The simple draw function outlined above probably won't be enough, you will likely have to pass it something.

Rigid body

The class definition for the rigid body should look something like this:

```
class RigidBody: public PhysicsObject
{
public:
	RigidBody(glm::vec2 a_position, glm::vec2 a_velocity, float a_rotation, float a_mass);


	virtual void Update(float a_gravity,float a_timeStep);
	virtual void Debug();
	void ApplyForce(glm::vec2 a_force) override;
	void ApplyForceToActor(RigidBody* a_otherActor, glm::vec2 a_force) override;

	void SetPosition(glm::vec2 a_position);
	glm::vec2 GetPosition() const;
	void SetVelocity(glm::vec2 a_velocity);
	glm::vec2 GetVelocity() const;
	void SetMass(float a_mass);
	float GetMass() const;
	void Setrotation2D (float a_rotation);
	float Getrotation2D () const;

private:
	glm::vec2 m_position;
	glm::vec2 m_velocity;
	float m_mass;
	float m_rotation2D; //2D so we only need a single float to represent our rotation

};
```

The final function prototype 'applyForceToActor()' is a variation of the applyforce() function which will be used to demonstrate newtons third law of motion. It allows us to simulate one actor "pushing" another.

Circle

The definition for the sphere class will look something like:

```
class CircleBody: public RigidBody
{
public:
	CircleBody(	glm::vec2 a_position,
			glm::vec2 a_velocity, float a_mass, float a_radius,
			glm::vec4 a_colour);
	void Draw(aie::Renderer2D* a_2dRenderer) override;
	void SetRadius(float a_radius);
	float GetRadius() const;

private:
	float m_radius;

};
```

We have a single constructor which allows us to instantiate an actor in our scene with a starting position, velocity, mass radius and colour.

Note how the *Update()* and *Applyforce()* functions for all our rigid bodies are done in the rigid body class so we don't need to overload the update function for each child class, they are always the same.

In this tutorial you don't need to derive classes for the plane or the boxes. All our experiments with newtons laws of motion will be done with spheres.

The update function will add the actor's velocity to position and modify the actor's velocity based on any forces currently applied to it. Because gravity is applied to all dynamic actors in the scene it makes sense to do that every frame in the update function.

For the *applyForceToActor()* function you will need to refer to the notes and understand how Newtons third law of motion works. **This will be an important function for both part 3 of this tutorial and all future tutorials!**

You should now create the bodies for the functions we have just outlined.

This is going to require planning and thought about how objects will interact in the future, plan this properly!

Creating a Scene

You will instantiate your physics scene using the Bootstrap project template provided. You actual implementation could look this within that framework.

```
void PhysicsSetupTute1()
{
	physicsScene = new PhysicScene();
	physicsScene->gravity = -9.8f;
	physicsScene->timeStep = 1/60f;
	//add four balls to our simulation
	CircleClass *newBall;
	newBall = new CircleClass( glm::vec2(-40, 0),		//pos
				   glm::vec2(0, 0),		//vel
				   3.0f, 40,			//mass, radius
				   glm::vec4(1, 0, 0, 1));	//colour

	physicsScene->addActor(newBall);
}

void OnUpdate(float a_deltaTime)
{
	physicsScene->Update();
}
```

In the above case we create a single actor which is a sphere with a starting position in the world of -40,0 a starting velocity of 0,0, a radius of 3 units, a mass of 1 unit and the colour red.

Now you can go through and implement your system for rendering these objects.
The class of example code has examples using sprites and the line renderer or box and draw circle calls. You are free to use any of these systems for your engine.

At this point you should have all your class working and you can test them by rendering some physics shapes like with the example above to ensure everything is working correctly. Once you are satisfied your logic is correct move onto the following section.

Demonstrating Newtons first and second laws

The first law can be demonstrated by setting gravity to zero and creating an actor with an initial velocity. It will continue to move at the same velocity.

The second law can be observed by setting gravity to a non-zero value for an actor with a non-zero starting velocity. The actor should move in a parabolic path simulating the movement of a projectile. Applying a force to the actor whilst the simulation is running further demonstrates the second law.

Demonstrating Newtons third law

Newton's third law is trickier to demonstrate. First you will need to implement the body for the `applyForceToActor(DIYRigidBody* a_otherActor, glm::vec2 a_force)` function. This function needs to call the *applyForce()* function on both the actor it's called on and the actor passed in that has the equal and opposite force applied to it (`a_otherActor`). The force applied to the first actor is *force* and the force applied to the passed in actor is *-force*.

To demonstrate this working, instantiate two spheres of the same mass and radius, next to each other, in the centre of the screen. In the update function scan for a key press, when the key is pressed use the *applyForceToActor()* to apply a force, once, to one of the actors. The force you apply must be towards the second actor. You should find that the two actors move away from each other with equal velocities.

Repeat the experiment but try it with spheres of different masses.

Simulating a rocket motor

Finally, you are going to simulate a simple rocket motor. Create an actor in the centre of the screen that will be your rocket. Set gravity to zero. In the update function, at time intervals, you need to:

Reduce the mass of the rocket by M to simulate fuel being used

Create a new sphere of mass M next to the rocket to simulate an exhaust gas particle

Use *applyForceToActor()* to apply force to the exhaust gas from the rocket (make sure it is in the correct direction).

If this seems confusing: you are going to need to think about how to apply the force to the actor in the update function. Refer to your physics equations, ie. F = ma. *(*__hint__* you can use basic algebra to figure out an acceleration you need to add)*

Repeat until all the mass has been used up

You will need to experiment with different forces and masses, firing rate, to make it work properly. For added effect make the exhaust gas particles smaller than the rocket and a different colour. Try turning gravity on and see if you can get the rocket to lift off against gravity. Try changing the direction of the force you apply to steer the rocket around the screen.

Your complete project file demonstrating each of the sections here is to be submitted via portal.