Making a 2D Physics Engine

Start with the project in:

\\adlfs1\shared\Students\Games Programming Year 2\Physics\OGLPhysics2DBase

This sets up an orthographic camera and OpenGL rendering for you using the Gizmos 2D library

Milestones

1. Get a PhysicsObject class and a RigidBody subclass that can move under constant velocity and draw itself to the screen. For now RigidBody can just be a Circle. The Circle should be updated and drawn as part of a list of PhysicObject pointers in your application class. Use dt correctly.
2. Add Gravity to the RigidBody, using an ApplyForce member function.
3. Create a Plane class derived from PhysicsObject that can draw itself
4. Set up a collision system so that each PhysicsObject is tested against each other one.
5. Write a Circle-Plane collision algorithm. Express the collision response in terms of applying a force to the Circle. (An equal and opposite force is assumed to be applied to the Plane, but it has infinite mass so nothing happens!)
6. Create Circle and Box subclasses of RigidBody. Move the Circle specific bits into the Circle class. Get the Box drawing itself.
7. Get rotation and angular velocity working so that a box can be given a constant angular rotation at startup. Add rotation and torque to ApplyForce, which will now need to take a position as well as a force.
8. Box-Plane collision, with collision response so that a Box dropped at an angle starts to rotate.
9. Circle–Circle collision
10. Circle-Box collision, which imparts correct torque to the box
11. Box-Box collision
12. Drag and friction
13. Make a simple game eg Angry Birds/Pool

Setting up the Gizmo system

The Gizmo system by default doesn't allocate much space for drawing 2D primitives. Initialise it with these arguments:

```
Gizmos::create(65335U, 65535U, 65535U, 65535U);
```

Setting up a Physics Scene

Make a PhysicsObject base class with virtual abstract functions Draw() and Update(float dt)

Add a std::list of PhysicsObject pointers to the Application derived class

In the Application derived Update iterate over this list and call update

In the Application derived Draw iterate over the list and call Draw on each PhysicsObject

You can then add PhysicsObjects to your list in startup().

It will be helpful to create constructors for your classes, so that you can add a PhysicsObject in one line.

e.g. put this in RigidBody,

```
RigidBody() {}
RigidBody(glm::vec2 p, glm::vec2 v)
{
	position = p;
	velocity = v;
}
```

And in startup you can add a RigidBody in one line

```
objects.push_back(new RigidBody(glm::vec2(0, 0), glm::vec2(1, 0)));
```

It will be helpful to move all these to a function and clear the list first. That way you can easily make a restart button in you PhysicsApplication::update as follows

```
if (glfwGetKey(window, GLFW_KEY_P))
	MakeScene();
```

Adding Gravity

Gravity can be a static vec2 member of PhysicsObject. Set it in the application to either zero (for top-down pool) or non-zero (for Angry Birds). Apply it as a Force, but multiply by the objects's mass first so that it acts as a pure acceleration.

Planes

I find it easiest to represent a Plane as a point on the plane P, and a normal N. This gives the equation of points on the plane as

(x-P) dot N = 0

e.g. a plane at y=-5 along the x-axis would have P = (0, -5), N = (0, 1)

This is a little bit redundant, as you can represent a plane with just three numbers, but it's easier to work with.

To draw a plane, draw a line between the points P-100*parallel and P+100*parallel

Where parallel = (normal Y, -normal X), is the direction along the plane.

Collision System

Each object needs to check against each other object every frame.

We can do this in an update loop like so, where each object checks all objects higher up the list from it, so that each pair is processed once:

```
for (auto it = m_physicsObjects.begin(); it != m_physicsObjects.end(); it++)
{
	PhysicsObject* obj = *it;
	obj->Update(dt);

	// collisions - check this object with everything further up the list
	for (auto it2 = it; it2 != m_physicsObjects.end(); it2++)
	{
		if (it != it2)
		{
			PhysicsObject* obj2 = *it2;
			obj2->CheckCollisions(obj);
		}
	}
}
```

CheckCollisions is a virtual function. Each object could be a Plane or Circle (or eventually Box). You'll want to set up individual functions like Box::CollideWithPlane, Circle::CollideWithCircle and so on. How you engineer these with either virtual functions, or switch statements based on type, or a mix of both, is up to you.

Possibly the simplest way is to give PhysicsObject an enumerated type for object types.

```
enum PhysicsObjectType
{
	PLANE,
	CIRCLE,
	BOX,
	SPRING,
};

PhysicsObjectType objectType;
```

You then need to make sure that every constructor sets the correct type.

e.g.

```
Circle() { objectType = CIRCLE; }
Circle(glm::vec2 p, glm::vec2 v, float m = 1, float a = 0)
{
	position = p;
	velocity = v;
	angle = a;
	mass = m;
	oType = CIRCLE;
}
```

We can then write a function in PhysicsObject called CheckCollisions that calls the correct virtual function, and set up the declarations for pure virtual functions in there.

```
virtual void CollideWithPlane(Plane* plane) = 0;
virtual void CollideWithCircle(Plane* plane) = 0;

void CheckCollision(PhysicsObject* other)
{
	switch (other->objectType)
	{
	case PLANE: CollideWithPlane((Plane*)other); break;
	case BOX: CollideWithCircle((Circle*)other); break;
	…
	}
}
```

When these functions are overridden, they are unambiguous, and can contain actual code to do the collisions.

e.g.

```
Circle::CollideWithCircle(Circle* other)

Plane::CollideWithCircle(Circle* other)

Circle::CollideWithPlane(Plane* plane)
```

Note that some pairs do the same thing but in a different order. We can write the actual maths code in one function, and get the other function to call it. e.g

```
Circle::CollideWithPlane(Plane* plane)
{
	plane->CollideWithCircle(this);
}
```

Circle-Plane collisions

A circle has a centre and radius.

A plane has a point and a normal.

We can calculate the perpendicular distance of the centre of the circle to the plane as

d = (circle Position – Plane P) dot (Plane n)

The velocity of the circle in the perpendicular direction of the plane is:

V perp = (circle Velocity) dot (Plane n)

If we're closer than the circle radius to the plane, and heading towards it, we want to reverse or stop the component of velocity into the plane.

The collision test is:

if ((d > 0 && d < radius && vperp < 0) || (d<0 && d> -radius && vperp > 0))

Our collision response should be to apply a force to the circle to stop its velocity in the direction of the plane.

The change in velocity we desire is -(vperp* Plane Normal) to stop the circle dead. To create a perfect elastic bounce, we want twice that. For an intermediate coefficient of restitution r between 0 and 1, we want to change the velocity by -(vperp*Plane Normal) *(1+r)

So the force we apply is

-Circle Mass * (vperp*Plane Normal) *(1+r),

and we do this via the ApplyForce method.

Drawing a Box

I did this. I stored the local x and y axes of the box based on its angle of rotation and use them to draw it based on its width and height using two triangles. You could use Gizmos::add2DAABBFilled instead and supply a transform if you prefer that method.

Angle is the current rotation, rotation is the angular velocity, ie how much angle is changing every second.

```
void RigidBody::Update(float dt)
{
	// apply gravity to the centre of mass as a straight acceleration
	velocity += gravity * dt;

	angle += rotation * dt;
	position +=velocity * dt;

	//store the local axes
	float cs = cosf(angle);
	float sn = sinf(angle);
	localX = glm::vec2(cs, sn);
	localY = glm::vec2(-sn, cs);
}


void PhysicsRectangle::MakeGizmo()
{
	glm::vec2 p1 = position - localX * width / 2.0f - localY * height / 2.0f;
	glm::vec2 p2 = position + localX * width / 2.0f - localY * height / 2.0f;
	glm::vec2 p3 = position - localX * width / 2.0f + localY * height / 2.0f;
	glm::vec2 p4 = position + localX * width / 2.0f + localY * height / 2.0f;
	Gizmos::add2DTri(p1, p2, p4, color);
	Gizmos::add2DTri(p1, p4, p3, color);
}
```

Rotational Physics

Now that we have a rotatable object (the Box) we need to modify ApplyForce to impart rotation as well as linear acceleration.

RigidBody should add a moment member, which represents the moment of inertia, the rotational equivalent of mass. Moment can be initialized based on the object's mass and its shape.

For a circle:

```
moment = 0.5f* mass * radius*radius;
```

For a Box:

```
moment = 1.0f/12.0f * mass * width*height;
```

We now need to pass a contact point into ApplyForce to say where the force is applied, so that we can calculate the torque applied to the rigidbody.

```
void RigidBody::ApplyForce(glm::vec2 force, glm::vec2 pos)
{
	velocity += force / mass;
	rotation += (force.y * pos.x - force.x * pos.y) / (moment);
}
```

Box-Plane collisions

For a box-plane collision, we want to check the four corners of the box and look at their distance to the plane. If their sign is opposite to the velocity of the Box along the normal of the plane, we have penetrated it and need to resolve the collision.

Using localX and LocalY as calculated above, we check the four corners. At each corner, we calculate the velocity of that point, based on the sum of the object's linear velocity and the velocity of that point due to rotation.

For each corner that makes a contact, we store its position and, add to the total number of contacts. At the end of the loop we find an average point of contact to apply the force to. In general, only one, or rarely two, contacts will occur.

We then apply a force at that point in the same manner as we did to the ball-plane collision, so that the velocity component along the plane normal is either removed (restitution = 0) or reversed (restitution = 1), or something in between.

We then apply a force sufficient to stop the contact point from moving into the plane due to the combined effects of the linear acceleration and the torque produced by the force.

(Contact force code is added in yellow below)

```
void Box::CollideWithPlane(Plane* plane)
{
```

```
	int numContacts = 0;
	glm::vec2 contact(0, 0);
	float contactV = 0;
	float radius = 0.5f * std::fminf(width, height);

	// which side is the centre of mass on?
	float comFromPlane = glm::dot(position - plane->origin, plane->normal);
	float penetration = 0;

	// check all four corners to see if we've hit the plane
	for (float x = -width / 2; x < width; x += width)
	{
		for (float y = -height / 2; y < height; y += height)
		{
			// get the position of the corner in world space
			glm::vec2 p = position + x*localX + y*localY;
			float distFromPlane = glm::dot(p - plane->origin, plane->normal);

			// this is the total velocity of the point
			float velocityIntoPlane = glm::dot(velocity + rotation*(-y*localX+
x*localY), plane->normal);

			// if this corner is on the opposite side from the COM,
			// and moving further in, we need to resolve the collision
			if ((distFromPlane > 0 && comFromPlane < 0 && velocityIntoPlane > 0)
				|| (distFromPlane < 0 && comFromPlane > 0 && velocityIntoPlane
< 0))
			{
				numContacts++;
				contact += p;
				contactV += velocityIntoPlane;
				if (comFromPlane >= 0)
				{
					if (penetration > distFromPlane)
						penetration = distFromPlane;
				}
				else
				{
					if (penetration < distFromPlane)
						penetration = distFromPlane;
				}
			}
		}
	}

	// we've had a hit - typically only two corners can contact
	if (numContacts > 0)
	{
		// get the average collision velocity into the plane
		// (covers linear and rotational velocity of all corners involved)
		float collisionV = contactV / (float)numContacts;

		// get the acceleration required to stop (restitution = 0) or reverse
(restitution = 1) the average velocity into the plane
		glm::vec2 acceleration = -plane->normal * ((1.0f+restitution) *
collisionV);
		// and the average position at which we'll apply the force (corner or edge
centre)
```

```
            glm::vec2 localContact = (contact / (float)numContacts) - position;
            // this is the perpendicular distance we apply the force at relative to the
COM, so Torque = F*r
            float r = glm::dot(localContact, glm::vec2(plane->normal.y, -plane-
>normal.x));
            // work out the "effective mass" - this is a combination of moment of
            // inertia and mass, and tells us how much the contact point velocity
            // will change with the force we're applying
            float mass0 = 1.0f / (1.0f / mass + (r*r) / moment);
            // and apply the force
            ApplyForce(acceleration*mass0, localContact);
            position -= plane->normal* penetration;
      }
}
```

Circle to Circle Collisions

Determining whether two circles have collided is pretty easy. Find the distance between their centres, and if its less than the sum of their radii, they've touched.

(pos2-pos1). magnitude() < r1+r2

Collision Response

Collision response is a little bit trickier.

We take their velocity components along the direction of a line of force, either joining the centres of mass, (or provided by the calling code as an optional parameter. We only need to apply a corrective force if the contact points are moving closer along this line.

Newton's Third Law says we will have to apply equal and opposite forces to the two objects. Calculate the "effective mass" for each object again, ie how it will respond to the force both linearly and rotationally.

```
void RigidBody::ResolveCollision(RigidBody* other, glm::vec2 contact, glm::vec2* direction /*=NULL*/)
{
	// find the vector between their centres, or use the provided direction of force
	glm::vec2 unitDisp = direction ? *direction : glm::normalize(other->position - position);

	// get the component along this axis for each object
	glm::vec2  unitParallel(unitDisp.y, -unitDisp.x);

	// determine the total velocity of the contact points, boith linear and rotational
	float r1 = glm::dot(contact - position, -unitParallel);
	float r2 = glm::dot(contact - other->position, unitParallel);
	float v1 = glm::dot(velocity, unitDisp) + r1*rotation;
	float v2 = glm::dot(other->velocity, unitDisp) + r2*other->rotation;

	if (v1 > v2) // they're moving closer
	{
		// calculate the effective mass at contact point for each object
		// ie how much the contact point will move due to the force applied.
		float mass1 = 1.0f / (1.0f / mass + (r1*r1) / moment);
		float mass2 = 1.0f / (1.0f / other->mass + (r2*r2) / other->moment);

		glm::vec2 force = (1.0f + restitution)*mass1*mass2 / (mass1 + mass2)*(v1-v2)*unitDisp;

		//apply equal and opposite forces
		ApplyForce(-force, contact-position);
		other->ApplyForce(force, contact-other->position);
	}
}
```

Energy Calculations

This collision response conserves total kinetic energy when restitution = 1.

Total kinetic energy is:

½ * mass * velocity^2 + ½ * Moment * rotation^2

Gravititional Potential Energy is:

Mass * gravity acceleration

You can calculate total energy and output it of you like, so see how the total changes over time, and how energy is transferred between these three components (linear kinetic, rotational kinetic and gravitational)

Circle to Box Collisions

How do we determine when a circle and a box have collided?

We have a collision when:

1) A corner of the box is inside the circle
2) The circle has touched one of the edges of the box

Checking the first one is relatively simple, using a radius test.

```
void Box::CollideWithCircle(Circle* circle)
{
	glm::vec2 circlePos = circle->position - position;
	float w2 = width / 2, h2 = height / 2;

	int numContacts = 0;
	// contact is in our box coordinates
	glm::vec2 contact(0, 0);


	// check the four corners to see if any of them are inside the circle
	for (float x = -w2; x < width; x += width)
	{
		for (float y = -h2; y < height; y += height)
		{
			glm::vec2 p = x*localX + y*localY;
			glm::vec2 dp = p - circlePos;
			if (dp.x*dp.x + dp.y*dp.y < circle->radius*circle->radius)
			{
				numContacts++;
				contact += glm::vec2(x,y);
			}
		}
	}
```

To check the second condition, it's easier to rotate the whole situation into Box space coordinates first, so that the box's local x and y axes become our coordinate space. We're looking for cases where the circle centre lies within the box extents in one direction, but outside the box in the other direction.

Think of the rectangle dividing space into 8 sections. We test the four corner sections first using the corner-in-circle test, and the four edge quadrants with the distance from plane check. We calculate a direction of force as normal to the edge of the rectangle

```
        glm::vec2* direction = NULL;
        // get the local position of the circle centre
        glm::vec2 localPos(glm::dot(localX, circlePos), glm::dot(localY, circlePos));
        if (localPos.y < h2 && localPos.y > -h2)
        {
                if (localPos.x > 0 && localPos.x < w2 + circle->radius)
                {
                        numContacts++;
                        contact += glm::vec2(w2, localPos.y);
                        direction = new glm::vec2(localX);
                }
                if (localPos.x < 0 && localPos.x > -(w2 + circle->radius))
                {
                        numContacts++;
                        contact += glm::vec2(-w2, localPos.y);
                        direction = new glm::vec2(-localX);
                }
        }
        if (localPos.x < w2 && localPos.x > -w2)
        {
                if (localPos.y > 0 && localPos.y < h2 + circle->radius)
                {
                        numContacts++;
                        contact += glm::vec2(localPos.x, h2);
                        direction = new glm::vec2(localY);
                }
                if (localPos.y < 0 && localPos.y > -(h2 + circle->radius))
                {
                        numContacts++;
                        contact += glm::vec2(localPos.x, -h2);
                        direction = new glm::vec2(-localY);
                }
        }
```

Finally, we can average our contact points and resolve the collision using the same function we used before.

```
        if (numContacts > 0)
        {
                // average, and convert back into world coords
                contact = position + (1.0f / numContacts) * (localX*contact.x + localY*contact.y);
                ResolveCollision(circle, contact, direction);
        }
        delete direction;
}
```

Box to Box Collisions

Box to box collision is determined as a special case of Separating Axes theorem. Each box has two axes - local X and localY. The projection of the other box on that axis is found, and we find the case where the overlap is smallest, and resolve the collision using that normal and penetration.

We do this by transforming the corners of Box into the space of Box A, and vice versa.

The following function does just this, and updates a sum total of contact points in world space, the number of contacts, and returns the normal to the edge which has just been penetrated by a corner.

```
// check if any of the other boxes corners are inside us
bool Box::CheckBoxCorners(Box* box, glm::vec2& contact, int& numContacts, float &pen, glm::vec2&
edgeNormal)
{
	float minX, maxX, minY, maxY;
	float w2 = width / 2, h2 = height / 2;
	int numLocalContacts = 0;
	glm::vec2 localContact;

	bool first = true;
	for (float x = -box->width / 2; x < box->width; x += box->width)
	{
		for (float y = -box->height / 2; y < box->height; y += box->height)
		{
			glm::vec2 p = box->position + x*box->localX + y*box->localY; // position in
worldspace
			glm::vec2 p0(glm::dot(p - position, localX), glm::dot(p - position, localY)); //
position in our box's space

			if (first || p0.x < minX) minX = p0.x;
			if (first || p0.x > maxX) maxX = p0.x;
			if (first || p0.y < minY) minY = p0.y;
			if (first || p0.y > maxY) maxY = p0.y;

			if (p0.x >= -w2 && p0.x <= w2 && p0.y >=-h2 && p0.y <= h2)
			{
				numLocalContacts++;
				localContact += p0;
			}
			first = false;
		}
	}

	if (maxX <-w2 || minX >w2 || maxY<-h2 || minY >h2)
		return false;
	if (numLocalContacts == 0)
		return false;

	bool res = false;

	contact += position + (localContact.x*localX + localContact.y*localY) / (float)numLocalContacts;
	numContacts++;

	float pen0 = w2 - minX;
	if (pen0 > 0 && (pen0 < pen || pen==0))
	{
		edgeNormal = localX;
		pen = pen0;
		res = true;
	}
	pen0 = maxX + w2;
	if (pen0 > 0 && (pen0 < pen || pen == 0))
	{
		edgeNormal = -localX;
```

```
                pen = pen0;
                res = true;
        }
        pen0 = h2 - minY;
        if (pen0 > 0 && (pen0 < pen || pen == 0))
        {
                edgeNormal = localY;
                pen = pen0;
                res = true;
        }
        pen0 = maxY + h2;
        if (pen0 > 0 && (pen0 < pen || pen == 0))
        {
                edgeNormal = -localY;
                pen = pen0;
                res = true;
        }
        return res;
}
```

To check a Box-Box collision, we apply this to both boxes against each other, and call our ResolveCollision function as before.

```
void Box::CollideWithBox(Box* box)
{
        glm::vec2 boxPos = box->position - position;

        {
                glm::vec2 norm;
                glm::vec2 contact;
                float pen = 0;
                int numContacts = 0;
                CheckBoxCorners(box, contact, numContacts, pen, norm);
                if (box->CheckBoxCorners(this, contact, numContacts, pen, norm))
                        norm = -norm;

                if (pen > 0)
                {
                        ResolveCollision(box, contact/float(numContacts), &norm);
                        float numDynamic = (fixed ? 0 : 1) + (box->fixed ? 0 : 1);
                        if (numDynamic > 0)
                        {
                                glm::vec2 contactForce = norm * pen / numDynamic;
                                if (!fixed)
                                        position -= contactForce;
                                if (!box->fixed)
                                        box->position += contactForce;
                        }
                }
        }

}
```

Note the n1-n2 for calculating the total edgeNormal. If we have two boxes with vertical faces about to collide side on, one will return an edgeNormal of (-1,0), the other of (1,0). We dont want these to cancel out, but to reinforce each other, so the edgeNormal from this to the other box is n1-n2.

Contact Forces and Penetration

If two RigidBodys are overlapping, but have no velocity (say they've been initialized this way), then our force-based collision detection and resolution can do nothing to separate them.

We can fix this problem by applying "contact forces" that directly move the objects so that they don't overlap, and do this along with the application of forces.

We need to do this for every type of interaction. For example, in a circle to circle collision, we would add the following highlighted code:

```
        // find the vector between their centres
        glm::vec2 disp = circle->position - position;
        // and get its length
        float d = sqrtf(disp.x*disp.x + disp.y*disp.y);

        // if we're closer than the two radii, we're touching
        if (d > 0 && d < (radius + circle->radius))
        {
                // apply a direct position adjustment to stop them overlapping
                glm::vec2 contactForce = 0.5f * (d - (radius + circle->radius))*disp / d;
                position += contactForce;
                circle->position -= contactForce;
                ResolveCollision(circle, 0.5f*(position + circle->position));
        }
```

The contact force is calculated as the amount of penetration of the bodies (d-(r1+r2)) times the normalized displacement vector (disp/d) times half, so that we apply half of the adjustment to each object.

For collisions between Rigidbodys and planes, we calculate the penetration of the object into the plane, and apply all the adjustment to the Rigidbody.

For a circle against a plane:

```
        //contact force
        position += plane->normal * (distFromPlane - radius);
```

For a box against a plane, accumulate the maximum penetration for each corner:

```
        if (comFromPlane > 0)
                penetration = std::fminf(penetration, distFromPlane);
        else
                penetration = std::fmaxf(penetration, distFromPlane);
```

And apply at the end of the loop over each corner

```
      position -= penetration* plane->normal;
```

Box to Box Contact Forces

For Box to Box contact forces, we add an extra parameter to our CheckBoxCorners function to return a contact force for each box against the other based on the maximum penetration and edge normal.

```
void Box::CheckBoxCorners(Box* box, glm::vec2& contact, int& numContacts, glm::vec2& edgeNormal,
glm::vec2& contactForce)
{
	float penetration = 0;

	for (float x = -box->width / 2; x < box->width; x += box->width)
	{
		for (float y = -box->height / 2; y < box->height; y += box->height)
		{
			glm::vec2 p = box->position + x*box->localX + y*box->localY; // pos in worldspace
			glm::vec2 p0(glm::dot(p - position, localX), glm::dot(p - position, localY));
			float w2 = width / 2, h2 = height / 2;

			if (p0.y < h2 && p0.y > -h2)
			{
				if (p0.x > 0 && p0.x < w2)
				{
					numContacts++;
					contact += position + w2 * localX + p0.y * localY;
					edgeNormal = localX;
					penetration = w2 - p0.x;
				}
				if (p0.x < 0 && p0.x > -w2)
				{
					numContacts++;
					contact += position - w2 * localX + p0.y * localY;
					edgeNormal = -localX;
					penetration = w2 + p0.x;
				}
			}
			if (p0.x < w2 && p0.x > -w2)
			{
				if (p0.y > 0 && p0.y < h2)
				{
					numContacts++;
					contact += position + p0.x * localX + h2 * localY;
					float pen0 = h2 - p0.y;
					if (pen0 < penetration || penetration == 0)
					{
						penetration = pen0;
						edgeNormal = localY;
					}
				}
				if (p0.y < 0 && p0.y > -h2)
				{
					numContacts++;
					contact += position + p0.x * localX - h2 * localY;
					float pen0 = h2 + p0.y;
					if (pen0 < penetration || penetration == 0)
					{
						penetration = pen0;
						edgeNormal = -localY;
					}
				}
			}

		}
	}
	contactForce = penetration*edgeNormal;

}
```

We then take the average of the two (inverting one of them) and apply to the boxes in Box::CollideWithBox().

```
        glm::vec2 n1, n2;
        glm::vec2 cf1, cf2;
        CheckBoxCorners(box, contact, numContacts, n1, cf1);
        box->CheckBoxCorners(this, contact, numContacts, n2, cf2);

        // we subtract because they're facing in diofferent directions
        glm::vec2 edgeNormal = glm::normalize(n1-n2);

        glm::vec2 contactForce = 0.5f*(cf1 - cf2);
        position -= contactForce;
        box->position += contactForce;
```

Springs

Springs can be derived from PhysicsObject. They have the following members:

```
        RigidBody* body1;
        RigidBody* body2;

        glm::vec2 contact1;
        glm::vec2 contact2;
        float restLength;
        float restoringForce;
```

which describe which two bodies they connect, and where they are attached, and the ideal length between attachment points and the strength of the restoring force.

In the Spring's update function, they check the distance between the attachment points, and apply a restoring force to both bodies proportional to the difference between the actual distance and the rest length of the spring.

```
void Spring::Update(float dt)
{
        glm::vec2 p2 = body2->ToWorld(contact2);
        glm::vec2 p1 = body1->ToWorld(contact1);
        glm::vec2 dist = p2 - p1;
        float len = sqrtf(dist.x*dist.x + dist.y* dist.y);

        // apply damping
        glm::vec2 dv = body2->velocity - body1->velocity;

        float damping = 0.1f; // make this a member
        glm::vec2 force = dist * restoringForce * (restLength - len) - damping * dv;


        body1->ApplyForce(-force*dt, p1 - body1->position);
        body2->ApplyForce(force*dt, p2 - body2->position);
}
```

Here I've used a RigidBody::ToWorld function to translate a position in the RigidBody's coordinate system into world coordinates. You can write this yourself.

Note that the force here is applied every timestep, so we multiply it by dt, unlike the single frame impulse forces of collision resolution.

Sleeping Objects

Most Physics engines have a system where an object can be flagged as asleep, and will wake up when it encounters a collision. This saves processing power, and adds stability to stacks of objects that are initialized as sitting on top of each other.

To implement this:

1) Add a m_awake member to Rigid Body
2) Set this to false or true via the Constructor, so it can be set on a per object basis
3) In the Update, don't update position and velocity if m_awake is false;
4) In the update, set velocity and angular velocity to zero if the object is asleep. Otherwise collision calculations can be compromised if the object appears to be moving or spinning.
5) In the event of a collision, set m_awake to true if the other object is also awake. Add these lines to ResolveCollision()

```
if (awake || other->awake)
{
        awake = true;
        other->awake = true;
}
```

6) In RigidBody::Update, we could check if an object's velocity and rotation are very small for a number of consecutive frames. If so, we put it to sleep again.

You could also add a "fixed" or "kinematic" member to permanently disable an objects update. Handy for representing cushions on a pool table, or pins in a pinball machine or other fixed collision shapes. These fixed shapes should be given a high mass to work best.

Creating a player controlled Physics Object

You can derive a Player class from Circle or Box, and give it an override to the base Update function.

In the update you'd call the base update, so that the object still behaves like a RigidBody, and then perform custom actions, such as adding forces based on keyboard or mouse input

To read the keyboard from an object's code, you need to access the OpenGL window. You can do this by making your PhyscisApplication a Singleton.

Add this code to your PhysicsApplication class

```
static PhysicsApplication* theApp;
PhysicsApplication() { theApp = this; }
```

And declare the static variable in the .cpp file to prevent an unresolved external.

You can then access the app anywhere like so:

```
GLFWwindow* window = PhysicsApplication::theApp->window;
```

Creating a whacking stick/pool cue

We can add a mouse driven stick to the application that allows us to draw a line from one point to another, and create a force at the end along the line.

Here's the code for drawing a line with the mouse when it's held down:

Add this to your PhysicsApplication class

```
glm::vec2 m_contactPoint;
glm::vec2 m_mousePoint;
bool m_mouseDown;
```

And in the update, add the following code to track the mouse

```
// check for mousedown
bool mouseDown = glfwGetMouseButton(window, 0);

double x0, y0;
glfwGetCursorPos(window, &x0, &y0);
mat4 view = camera.getView();
mat4 projection = camera.getProjection();

glm::vec3 windowCoordinates = glm::vec3(x0, y0, 0);
glm::vec4 viewport = glm::vec4(0.0f, 0.0f, 1280, 720);
glm::vec3 worldCoordinates = glm::unProject(windowCoordinates, view, projection, viewport);

m_mousePoint = vec2(worldCoordinates[0] * camera.getDistance(), worldCoordinates[1] * (-
camera.getDistance()));

if (mouseDown != m_mouseDown)
{
        if (mouseDown)
        {
                m_contactPoint = m_mousePoint;
        }
        m_mouseDown = mouseDown;
}
```

You'll have to add this to the Camera class.

```
float getDistance() { return 10 * radius; }
```

This now keeps track of the world coordinates under the mouse in 2D every frame, and stores the position where the mouse was pressed dow.

Add this to your drawing code, before Gizmos::Draw2D gets called:

```
if (m_mouseDown)
        Gizmos::add2DLine(m_contactPoint, m_mousePoint, white);
```

And you should be able to see a line as you hold the mouse down and drag.

To get this to apply a force to any object at the end point when you release, you could do this when the mouse is released in PhysicsApplication::Update() - this extends the code shown earlier.

```
	if (mouseDown != m_mouseDown)
	{
		if (mouseDown)
		{
			m_contactPoint = m_mousePoint;
		}
		else
		{
			for (auto it = m_physicsObjects.begin(); it != m_physicsObjects.end(); it++)
			{
				PhysicsObject* obj = *it;
				if (obj->IsInside(m_mousePoint))
				{
					// must be a RigidBody, so we can safely cast!
					RigidBody* rb = (RigidBody*)obj;
					rb->ApplyForce(2.0f*(m_mousePoint - m_contactPoint),
m_contactPoint - rb->position);
				}
			}
		}
		m_mouseDown = mouseDown;
	}
```

IsInside is a virtual function we add to PhysicsObject. For non-RigidBody objects, it just returns false, so put this in PhysicsObject.

```
virtual bool IsInside(glm::vec2 pt) { return false; }
```

We can then override this in Box and Circle. Here's my Box function, where I transform the world coordinates point into boxspace so I can do the test. You can write your own one for Circle, it's pretty straightforwards. Remember that the point you're passed in as an argument is in world coordinates.

```
bool Box::IsInside(glm::vec2 pt)
{
	pt -= position;
	glm::vec2 boxPt(glm::dot(pt, localX), glm::dot(pt, localY));
	return (fabs(boxPt.x) < width*0.5f && fabs(boxPt.y) < height*0.5f);
}
```