Basic Physics and Collisions

This is a very simple introduction to physics and collisions in Orx. In order to best understand it, we'll start with a ball, bouncing inside four walls.

To enable collisions and physics, the following are needed:

1. Two objects
2. Bodies added to the objects
3. At least one body part added to a body
4. Flags set on parts to make collisions active against other parts
5. Gravity if you want sideways physics

Note: If you need an empty project as an easy way to get started, see how to use one from the Project Tutorials.

First, here is a ball you can use which will bounce around the screen:

Save the ball.png to your project's data folder.

The ball data configuration is the first job, in order to position a single, regular ball object on the screen:

```
[BallObject]
Graphic = BallGraphic
Position = (400, 300, 0)

[BallGraphic]
Texture = ball.png
```

In code, create the ball onscreen:

```
orxObject_CreateFromConfig("BallObject");
```

Compile, and run. You will see a ball sitting in the middle of the screen.

The next step is to enable the gravity for our world. This can be done with:

```
[Physics]
Gravity = (0.0, 1000.0, 0.0)
```

If you re-run your game now, you won't see any change. The ball will not fall regardless of gravity.

This is because, in order for objects to be affected by gravity, or for them to be able to collide, they need to have a body defined (and at least one body part).

Adding a body and part is very simple, just change the config to be:

```
[BallObject]
Graphic  = BallGraphic
Position = (400, 300, 0)
Body     = BallBody

[BallGraphic]
Texture = ball.png

[BallBody]
PartList        = BallBodyPart
Dynamic         = true

[BallBodyPart]
Type            = sphere
Solid      = true
```

Re-run, and the ball should fall through the bottom of the screen and disappear forever. Excellent! Gravity is affecting the object.

A couple of things to note is the “Dynamic = true” on the body, which means that the body, and hence the object, will be affected by gravity, and will “fall”.

The body part type is a sphere. There is a body part sphere surrounding the object. This defines the region that can collide with other objects.

Because the ball falls down forever, it is best to constrain it with four walls to make a border. The walls will be given physical bodies as well so that the ball can bounce off them and stay on screen.

```
[Border]
Body     = BorderBody
Position = (0, 0, 0)

[BorderBody]
PartList        = BorderBodyPartTop # BorderBodyPartRight #
BorderBodyPartBottom # BorderBodyPartLeft
Dynamic         = false

[BorderBodyPartDefaults]
Type            = box
Solid           = true

[BorderBodyPartTop@BorderBodyPartDefaults]
TopLeft         = (0, 0, 0)
BottomRight     = (800, 20, 0)

[BorderBodyPartLeft@BorderBodyPartDefaults]
TopLeft         = (0, 0, 0)
```

```
BottomRight     = (20, 600, 0)

[BorderBodyPartRight@BorderBodyPartDefaults]
TopLeft         = (780, 0, 0)
BottomRight     = (800, 600, 0)

[BorderBodyPartBottom@BorderBodyPartDefaults]
TopLeft         = (0, 580, 0)
BottomRight     = (800, 600, 0)
```

Add the following to the code to create the Border onscreen:

```
orxObject_CreateFromConfig("Border");
```

Compile and re-run. The ball will still fall through the bottom of the screen. But why?

Best to turn on the physics debug to see what is going on. Add the ShowDebug property to the Physics section like so:

```
[Physics]
Gravity        = (0.0, 1000.0, 0.0)
ShowDebug  = true
```

Re-run. This is a very handy feature of Orx. You can see the debug borders on the bodies clearly marking where collisions can occur.

And notice that the ball has a spherical body and the borders have box bodies. Yet the ball is still falling through. Clearly something isn't right.

Masks and Flags have not been set on both object's body parts. Set them like this:

```
[BorderBodyPartDefaults]
Type            = box
SelfFlags       = border
CheckMask       = ball
Solid           = true

[BallBodyPart]
Type            = sphere
SelfFlags       = ball
CheckMask       = border
Solid           = true
```

Notice that SelfFlags is set to some named text that identifies the body part. Then the CheckMask indicates which parts can collide. In this case, we need to say that the ball can collide with the border, and that the border can collide with the ball.

Re-run and the ball should fall down and stay at the bottom of the screen. Great, which means the parts are working well.

Feel free to turn off debugging again with:

```
[Physics]
Gravity        = (0.0, 1000.0, 0.0)
ShowDebug  = false
```

Not very interesting yet is it? Let's make the ball 100% bouncy by adding full restitution to the body part:

```
[BallBodyPart]
Type            = sphere
SelfFlags       = ball
CheckMask       = border
Solid           = true
Restitution    = 1.0
```

Rerun and the ball should be bouncing. Add a little sideways movement with a Speed setting on the ball object:

```
[BallObject]
Graphic    = BallGraphic
Position   = (400, 300, 0)
Body       = BallBody
Speed      = (200, 0, 0)
```

That looks good. Now what about colliding with another object? Another ball? We can make a second ball based off the first one?

```
[Ball2Object@BallObject]
Position   = (200, 300, 0)
Speed      = (-200, 0, 0)
```

This will start a second ball based on the properties of “BallObject” but the Position is changed and the speed with start in the opposite direction.

Create the second ball in code with:

```
orxObject_CreateFromConfig("Ball2Object");
```

Compile and re-run. This looks good with two balls bouncing off the borders, but notice now that the two balls simply pass through each other.

In this case, we need to tell the ball body part to collide with itself:

```
[BallBodyPart]
Type            = sphere
SelfFlags       = ball
CheckMask       = border # ball
```

```
Solid           = true
Restitution     = 1.0
```

The CheckMask on a ball body part means that it can collide with both the border and any other ball object that it encounters.

The last thing to cover is the ability to respond to a collision event in code.

Start by defining a physics event handler in your Init() function:

```
orxEvent_AddHandler(orxEVENT_TYPE_PHYSICS, PhysicsEventHandler);
```

There is no PhysicsEventHandler function yet, so create it with:

```
orxSTATUS orxFASTCALL PhysicsEventHandler(const orxEVENT *_pstEvent)
{
}
```

The code we need to determine if a collision event has occured looks like this:

```
orxSTATUS orxFASTCALL PhysicsEventHandler(const orxEVENT *_pstEvent)
{
    if (_pstEvent->eID == orxPHYSICS_EVENT_CONTACT_ADD) {
        orxPHYSICS_EVENT_PAYLOAD *pstPayload;
        pstPayload = (orxPHYSICS_EVENT_PAYLOAD *)_pstEvent->pstPayload;
    }
    return orxSTATUS_SUCCESS;

}
```

And the complete example to handle if there has been a collision between the two ball objects is:

```
orxSTATUS orxFASTCALL PhysicsEventHandler(const orxEVENT *_pstEvent)
{
    if (_pstEvent->eID == orxPHYSICS_EVENT_CONTACT_ADD) {
        orxPHYSICS_EVENT_PAYLOAD *pstPayload;
        pstPayload = (orxPHYSICS_EVENT_PAYLOAD *)_pstEvent->pstPayload;

        /* Gets colliding objects */
        orxOBJECT *recipient, *sender;
        recipient = orxOBJECT(_pstEvent->hRecipient);
        sender = orxOBJECT(_pstEvent->hSender);

        if (orxString_Compare(orxObject_GetName(recipient), "BallObject") ==
0 && orxString_Compare(orxObject_GetName(sender), "Ball2Object") == 0) {
            orxLOG("The balls collided!");
        }
        if (orxString_Compare(orxObject_GetName(recipient), "Ball2Object")
== 0 && orxString_Compare(orxObject_GetName(sender), "BallObject") == 0) {
            orxLOG("The balls collided!");
        }
```

```
    }

    return orxSTATUS_SUCCESS;
}
```

Firstly, we check that the type of event that has occurred is a CONTACT_ADD event (collision between a sender and recipient objects). Then the payload is retrieved. The sender and recipient objects are retrieved from the payload, and the names of the objects are checked from both.

If one is a “BallObject” and the other is a “Ball2Object” then output a message to the log.

Compile and run.

Not exactly exciting, but we do get a successful message when the balls collide.

That takes care of demonstrating physics and gravity, and also collisions between object body parts. If you would like to experiment a little further with physics, collisions and resulting FX, feel free to move on to the Physics Tutorial.