Physics tutorial

Summary

See previous basic tutorials for more info about basic object creation, clock handling, frames hierarchy, animations, cameras & viewports, sounds & musics and FXs.

This tutorial shows how to add physical properties to objects. By adding a body (and body parts) you can handle collisions between objects.

As you can see, the physical properties are completely data-driven. Thus, creating an object with physical properties (ie. with a body) or without results in the exact same line of code, eg:

```
orxObject_CreateFromConfig("MyObject");
```

Objects can be linked to a body which can be static or dynamic.
Each body can be made of up to 8 parts.

A body part is defined by:

- its shape (currently box, sphere and mesh (ie. convex polygon) are the only available)
- information about the shape size (corners for the box, center and radius for the sphere, vertices for the mesh)
- if no size data is specified, the shape will try to fill the complete body (using the object size and scale)
- collision “self” flags that defines this part
- collision “check” mask that defines with which other parts this one will collide [1)]
- a flag (`Solid`) specifying if this shaped should only give information about collisions or if it should impact on the body physics simulation (bouncing, etc...)
- various attributes such as restitution, friction, density, ...

In this tutorial we create static solid walls around our screen. We then spawn boxes in the middle.
The number of boxes created is tweakable through the config file and is 100 by default.

The only interaction possible is using left and right mouse buttons (or left and right keys) to rotate the camera.
As we rotate it, we also update the gravity vector of our simulation.
Doing so, it gives the impression that the boxes will be always falling toward the bottom of our screen no matter how the camera is rotated.

We also register to the physics events to add a visual FXs on two colliding objects.
By default the FX is a fast color flash and is, as usual, tweakable in realtime (ie. reloading the config history will apply the new settings immediately as the FX isn't kept in cache by default).

Updating an object scale (including changing its scale with FXs) will update its physical properties (ie. its body).
Keep in mind that scaling an object with a physical body is more expensive as we have to delete the current shapes and recreate them at the correct size.

This is done this way as our current single physics plugin is based on Box2D which doesn't allow realtime rescaling of shapes.

This tutorial does only show basic physics and collision control, but, for example, you can also be notified with events for object separating or keeping contact.

Details

As usual, we begin by creating a viewport, getting the main clock and registering our `Update` function to it.
Please refer to the previous tutorials for more details.

We also creates our walls. Actually we won't create them one by one, we'll group them in a `ChildList` of a parent object.

```
orxObject_CreateFromConfig("Walls");
```

This looks like we only create one object called `Walls`, but as we'll see in the config file, it's actually a container that will spawn a couple of walls.

Lastly, we create our boxes.

```
for(i = ; i < orxConfig_GetU32("BoxNumber"); i++)
{
  orxObject_CreateFromConfig("Box");
}
```

As you can see, we don't specify anything regarding the physics properties of our walls or boxes, this is entirely done in the config file and is fully data-driven.

We then register to physics events.

```
orxEvent_AddHandler(orxEVENT_TYPE_PHYSICS, EventHandler);
```

Nothing really new here, so let's have a look directly to our `EventHandler` callback.

```
if(_pstEvent->eID == orxPHYSICS_EVENT_CONTACT_ADD)
{
  orxOBJECT *pstObject1, *pstObject2;

  pstObject1 = orxOBJECT(_pstEvent->hRecipient);
  pstObject2 = orxOBJECT(_pstEvent->hSender);

  orxObject_AddFX(pstObject1, "Bump");
```

```
  orxObject_AddFX(pstObject2, "Bump");
}
```

Basically we only handle the new contact event and we add a FX called Bump on both colliding objects. This FX will make them flash in a random color.

Let's now see our Update function.

```
void orxFASTCALL Update(const orxCLOCK_INFO *_pstClockInfo, void
*_pstContext)
{
  orxFLOAT fDeltaRotation = orxFLOAT_0;

  if(orxInput_IsActive("RotateLeft"))
  {
    fDeltaRotation = orx2F(4.0f) * _pstClockInfo->fDT;
  }
  if(orxInput_IsActive("RotateRight"))
  {
    fDeltaRotation = orx2F(-4.0f) * _pstClockInfo->fDT;
  }

  if(fDeltaRotation != orxFLOAT_0)
  {
    orxVECTOR vGravity;

    orxCamera_SetRotation(pstCamera, orxCamera_GetRotation(pstCamera) +
fDeltaRotation);

    if(orxPhysics_GetGravity(&vGravity))
    {
      orxVector_2DRotate(&vGravity, &vGravity, fDeltaRotation);
      orxPhysics_SetGravity(&vGravity);
    }
  }
}
```

As you can see, we get the rotation update from the RotateLeft and RotateRight inputs.
If a rotation needs to be applied, we then update our camera with orxCamera_SetRotation() and we update our physics simulation gravity accordingly.
This way, our boxes will always look like they fall toward the bottom of our screen, whichever the camera rotation is.
Note the use of orxVector_2DRotate() so as to rotate the gravity vector.

NB: All rotations in orx's code are always expressed in radians!

Let's now have a look at our config data. You can find more info on the config parameters in the body section of config settings.
First, we created implicitely many walls using the ChildList property. See below how it is done.

```
[Walls]
ChildList = Wall1 # Wall2 # Wall3 # Wall4; # Wall5 # Wall6
```

As we can see, our `Walls` object is empty, it will just create `Wall1`, `Wall2`, `Wall3` and `Wall4` (note the `';'` ending the list there).
You can remove this `';'` to create 2 additional walls.

Let's now see how we define our walls and their physical properties.
Let's begin with the shape we'll use for both `WallBody` and `BoxBody`.

```
[FullBoxPart]
Type        = box
Restitution = 0.0
Friction    = 1.0
SelfFlags   = 0x0001
CheckMask   = 0xFFFF
Solid       = true
Density     = 1.0
```

Here we request a part that will use a box shape with no `Restitution` (ie. no bounciness) and some `Friction`.
We also define the `SelfFlags` and `CheckMask` for this wall.
The first ones defines the identity flags for this part, and the second ones define to which idendity flags it'll be sensitive (ie. with who it'll collide).
Basically, if we have two objects: `Object1` and `Object2`. They'll collide if the below expression is TRUE.

```
(Object1.SeflFlags & Object2.CheckMask) && (Object1.CheckMask &
Object2.SelfFlags)
```

NB: As we don't specify the `TopLeft` and `BottomRight` attributes for this `FullBoxPart` part, it will use the full size of the body/object that will reference it.

Now we need to define our bodies for the boxes and the walls.

```
[WallBody]
PartList = FullBoxPart

[BoxBody]
PartList  = FullBoxPart
Dynamic   = true
```

We can see they both use the same part [2)].
As `Dynamic` is set to `true` for the `BoxBody`, this object will move according to the physics simulation.
For the `WallBody`, nothing is specified for the `Dynamic` attribute, it'll then default to `false`, and

walls won't move no matter what happen to them.
NB: As there can't be any collision between two non-dynamic (ie. static) objects, walls won't collide even if they touch or overlap.

Now that we have our bodies, let's see how we apply them to our objects.
First, our boxes.

```
[Box]
Graphic   = BoxGraphic
Position  = (50.0, 50.0, 0.0) ~ (750.0, 550.0, 0.0)
Body      = BoxBody
Scale     = 2.0
```

As you can see, our Box has a Body attribute set to BoxBody.
We can also notice it's random position, which means everytime we create a new box, it'll have a new random position in this range.

Let's now see our walls.

```
[WallTemplate]
Body = WallBody

[VerticalWall@WallTemplate]
Graphic = VerticalWallGraphic;
Scale   = @VerticalWallGraphic.Repeat;

[HorizontalWall@WallTemplate]
Graphic = HorizontalWallGraphic;
Scale   = @HorizontalWallGraphic.Repeat;

[Wall1@VerticalWall]
Position = (0, 24, 0)

[Wall2@VerticalWall]
Position = (768, 24, 0)

[Wall3@HorizontalWall]
Position = (0, -8, 0)

[Wall4@HorizontalWall]
Position = (0, 568, 0)

[Wall5@VerticalWall]
Position = (384, 24, 0)

[Wall6@HorizontalWall]
Position = (0, 284, 0)
```

As we can see we use inheritance once again.
First we define a `WallTemplate` that contains our `WallBody` as a `Body` attribute.
We then inherits from this section with `HorizontalWall` and `VerticalWall`. They basically have the same physical property but a different `Graphic` attribute.
Now that we have our wall templates for both vertical and horizontal wall, we only need to specify them a bit more by adding a position.
That's what we do with `Wall1`, `Wall2`, etc...

Resources

Source code: 08_Physics.c

Config file: 08_Physics.ini

Video: Video by acksys

[1)]
two parts in the same body will never collide
[2)]
they can have up to 8 parts, but only 1 is used here

From:
https://orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://orx-project.org/wiki/en/tutorials/physics

Last update: **2018/06/21 16:48 (23 months ago)**