Tutorial de Física

Sumario

Ver los anteriores tutoriales básicos para más información sobre la creación básica de objetos, manejo del reloj, jerarquía de fotogramas, animaciones, cámaras & vistas, música & sonido y efectos(FXs).

Este tutorial muestra como añadirle propiedades físicas a los objetos y manejar colisiones.

Como puedes ver, las propiedades físicas son completamente manipuladas por datos. Así, creando un objeto con propiedades físicas (ej. con un cuerpo) o sin ellas, resulta en exactamente la misma linea de código.

Los objetos pueden ser enlazados a un cuerpo, que puede ser estático o dinámico.
Cada cuerpo puede estar hecho de hasta 8 partes.

Un cuerpo es definido por:

- su forma (actualmente caja, esfera y malla(ej. poligono convexo) son los únicos disponibles)
- información acerca del tamaño de la forma (esquinas para la caja, centro y radio para la esfera, vértices para la malla)
- si el tamaño de los datos no es especificado, la forma tratará de llenar el cuerpo completo (usando el tamaño y la escala del objeto)
- las banderas propias de colisión definen esta parte
- la máscara de chequeo de colisión define con que otra parte ella puede colisionar [1)]
- una bandera sólida(`Solid`) especificando si esta forma puede solo brindar información acerca de colisiones o si puede impactar en simulaciones de cuerpos físicos (rebota, etc...)
- varios atributos como son restitución, fricción, densidad, ...

En este tutorial creamos muros estáticos sólidos alrededor de la pantalla. Entonces reproducimos cajas en el medio.
El número de cajas creadas serán ajustadas a través del fichero de configuración y es 100 por defecto.

La única interacción posible es usando los botones izquierdo y derechos del ratón (o las teclas izquierda y derecha) para rotar la cámara.
Como mismo lo rotamos, podemos actualizar el vector de gravedad de nuestra simulación.
Haciendo eso, nos da la impresión de que las cajas siempre están cayendo contra el fondo de nuestra pantalla, no importa como la cámara es rotada.

Registramos también los eventos físicos para añadir un FX visual en los dos objetos que colisionan.
Por defecto el FX es un parpadeo de color rápido y es, como norma, ajustable en tiempo real (ej. recargando la configuración histórica aplicaremos los nuevos ajustes inmediatamente si el FX no se mantiene en la cache por defecto).

Actualizando la escala de un objeto (incluso cambiando su escala con FXs) actualizaremos sus propiedades físicas (ej. su cuerpo).
Ten en mente que escalando un objeto con cuerpo físico es más caro que si tenemos que eliminar la

correspondiente forma y recreándolo en tamaño correcto.
Esto está hecho de esta manera ya que nuestro correspondiente plugin físico está basado en Box2D, que no permite rescalado en tiempo real de formas.

Este tutorial solo nos muestra un control básico de físicas y colisiones, pero, por ejemplo, tú puede también ser notificado con eventos por objetos separados o mantener el contacto.

Detalles

Como es usual, empezamos por cargar nuestro fichero de configuración, obteniendo el reloj principal y registrando nuestra función `Update` a el y, por último, por crear nuestro soldado y objectos de la caja.
Por favor, referirse a los tutoriales anteriores para más detalles.

Creamos también nuestros muros. Actualmente no los crearemos uno por uno, los agrupamos en una lista de hijos(`ChildList`) como objeto padre.

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

Esto luce como si hubiéramos creado un solo objeto llamado `Muros` (`Walls`), pero como vemos en el fichero de configuración, el es actualmente un contenedor que reproducirá varios muros.

Por último, creamos nuestras cajas.

```
for(i = 0; i < orxConfig_GetU32("BoxNumber"); i++)
{
  orxObject_CreateFromConfig("Box");
}
```

Como puedes ver, no especificamos nada respecto a las propiedades físicas de nuestros muros o cajas, eso es enteramente hecho en el fichero de configuración y completamente manejado por datos.

Registramos entonces los eventos físicos.

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

Nada realmente nuevo hasta aquí, miremos directamente a nuestra llamada de retorno `EventHandler`.

```
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");
```

```
}
```

Básicamente, solo manejamos el nuevo evento contactado y añadimos un FX llamado Bump en ambos objetos colisionadores. Este FX los hace parpadear en un color aleatorio.

Veamos ahora nuestra función `Update`.

```
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);
    }
  }
}
```

Como puedes ver, obtenemos la actualización desde las entradas `RotarIzquierda`(`RotateLeft`) y `RotarDerecha`(`RotateRight`).
Si una rotación necesita ser aplicada, entonces actualizamos nuestra cámara con `orxCamera_SetRotation()` y actualizamos nuestra simulación de gravedad física en consecuencia.
De esta manera, nuestras cajas siempre parecerán que caen al fonde de nuestra pantalla, sin importar la rotación de la cámara.
Note el uso de `orxVector_2DRotate()` para rotar el vector de gravedad.

PD: Todas las rotaciones en el código de orx siempre son expresadas en radianes!

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...

Recursos

[1)]

dos partes en el mismo cuerpo nuncá colisionarán

[2)]

they can have up to 8 parts, but only 1 is used here