The Binding of Objects in orx/Scroll

The code for this tutorial is available at https://github.com/orx/scroll-tutorial

What is "Object Binding"?

When we refer to “Object Binding” in this sense, we're describing “hooking up” a game object to a C++ class defining behaviors of the object.

This means when an Orx/Scroll Object is created, it can automatically be set to use a C++ class of choice for its implementation. This makes it easy to implement behavior that is specific to certain object types.

For instance, you want game objects to do certain things every frame. You want enemies to move on a path, or possibly attack. You want the player's character to be moved based on user input.

Additionally, binding objects to classes makes it easy to handle Orx events on an object-specific basis. For example, each type of object can have its own `OnCreate` function which is called whenever an object of that type is created. `OnDelete` is called when the object is deleted, etc.

In this tutorial, we're going to create a small game with two specific examples of object binding. We'll create an Enemy Bug object and a Hero object and bind them to classes. Our enemy bugs will move semi-randomly across the screen. Our Hero will be controlled by the player. The Hero flashes red on collision with an enemy bug as an example of FX to indicate getting hurt.

First, some preparation…

Trouble?

If you have trouble following this tutorial, please reach out in the #support channel on Discord. The community is very helpful.

If your problems are related to physics (collision detection), it can be very useful to turn on physics debugging. This is done in .ini config. See the comments in this tutorial's config and in `SettingsTemplate.ini` linked in Latest config settings for the Development Version.

Create a new Scroll Project

Before you begin this tutorial, you need a basic Orx/Scroll project created with ore's `init` tool for

initializing new projects. For details on doing this, see An Introduction to Scroll.

This tutorial assumes you're starting from a fresh `init` project with Scroll support included!

Get the Config Ready

You'll need to download these textures [1] for use in your config:

Then, you'll need to prepare this config in your main project `.ini` file for use with this tutorial:

```
; BindingOfObjects - Template basic config file

[Display]
; FullScreen = false + Decoration = false + no dimension -> Borderless
FullScreen
Title           = The Binding of Objects
FullScreen      = false
Decoration      = false
Smoothing       = true
VSync           = true

[Physics]
; Uncomment to show object bounding boxes
; ShowDebug = true

[Resource]
Texture         = bundle: # bundle:BindingOfObjects.obr # ../data/texture
Sound           = bundle: # bundle:BindingOfObjects.obr # ../data/sound

[Input]
KEY_ESCAPE      = Quit
KEY_LEFT        = MoveLeft
KEY_RIGHT       = MoveRight
KEY_UP          = MoveUp
KEY_DOWN        = MoveDown

[Main]
ViewportList    = MainViewport

[MainViewport]
Camera          = MainCamera
BackgroundColor = (50, 50, 50)
```

```
[MainCamera]
FrustumWidth    = 1280
FrustumHeight   = 720
FrustumFar      = 10
FrustumNear     = 0
Position        = (0, 0, -2)

[Scene]
ChildList       = O-Hero # O-EnemyBug # O-EnemyBug # O-EnemyBug # O-
EnemyBug # O-EnemyBug # O-EnemyBug

[O-Hero]
Graphic         = G-Hero
Position        = (0, 0, 0)
Body            = B-Hero

; Hero Class Data
MovementSpeed   = 100.0

[G-Hero]
Texture         = Character_Boy.png

[B-Hero]
PartList        = BP-Hero
Dynamic         = true

[BP-Hero]
Type            = box
SelfFlags       = 0x0001
CheckMask       = 0xFFFF

[O-EnemyBug]
Position        = (-600, 200, 0) ~ (600, 200, 0)
Graphic         = G-EnemyBug
Body            = B-EnemyBug

; EnemyBug Class Data
MovementSpeed   = 25.0
DirectionChangeInterval = 0.5 ~ 2.5

[G-EnemyBug]
Texture         = Enemy_Bug.png
Pivot           = center

[B-EnemyBug]
PartList        = BP-EnemyBug

[BP-EnemyBug]
Type            = box
SelfFlags       = 0x0002
```

```
CheckMask       = 0xFFFF

[FX-Flash]
SlotList        = FXS-FlashRed # FXS-Unflash

[FXS-FlashRed]
Type            = color
Curve           = smoother
StartTime       = 0
EndTime         = 0.1
Absolute        = true
StartValue      = (255, 255, 255)
Period          = 0.5
EndValue        = (255, 0, 0)

[FXS-Unflash]
Type            = color
Curve           = smoother
StartTime       = @FXS-FlashRed.EndTime
EndTime         = 0.2
Absolute        = true
StartValue      = @FXS-FlashRed.EndValue
EndValue        = (255, 255, 255)
```

It's all the typical object, graphic, physics stuff you've seen in previous Orx tutorials. Notice, however, we've added some extra properties to our EnemyBug and Hero objects. The Orx engine doesn't use these, but we'll use them later in this tutorial.

A newly created `init` project creates a `Scene` object by default, with other objects created as children. In the config above, we have the Hero object and several Enemy Bug objects in the `ChildList` for `Scene`.

Run your project. As you might expect, you'll see a hero and a few randomly placed enemy bugs in a gray expanse. Nothing happens since we've defined no behavior to control them. Pressing `Escape` will quit the game.

Stop and think for a moment about how you might add behavior to the objects in Orx.

Giving the Bugs a Brain (Deriving the ScrollObject Class)

The first step to object binding is to create a binding class. To do that, we derive from the `ScrollObject` base class. `init` gives us an `Object` class which derives from `ScrollObject` and we can derive our objects from `Object`.

First, let's create the interface for our derived class. Create a file called `EnemyBug.h` and add it to your project. Add the following code to `EnemyBug.h`:

```
#pragma once
```

```
#include "Object.h"

enum Direction
{
  NORTH,
  SOUTH,
  EAST,
  WEST,
  LAST = WEST,
};

class EnemyBug : public Object
{
public:
protected:
  void OnCreate();
  void OnDelete();
  void Update(const orxCLOCK_INFO &_rstInfo);

private:
  // Direction of movement
  Direction m_direction;
  // Speed of movement
  orxFLOAT m_movementSpeed;
  // Time since change of direction
  orxFLOAT m_timeSinceDirectionChange;
  // Time interval between direction changes
  orxFLOAT m_directionChangeInterval;
};
```

This class represents a single enemy bug. In our derived class, we override ScrollObject's OnCreate, OnDelete, and Update functions [2)].

Let's create the class implementation. Add a file called EnemyBug.cpp to your project and add the following code to it:

```
#include "EnemyBug.h"

void EnemyBug::OnCreate()
{
  // Set initial movement direction
  m_direction = SOUTH;
  // Get movement speed from config value
  m_movementSpeed = orxConfig_GetFloat("MovementSpeed");
  // Get direction change interval from config value
  m_directionChangeInterval = orxConfig_GetFloat("DirectionChangeInterval");
}

void EnemyBug::OnDelete()
{
```

```
  // Do nothing when deleted
}

void EnemyBug::Update(const orxCLOCK_INFO &_rstInfo)
{
  // Always initialize thy variables
  orxVECTOR speed = orxVECTOR_0;

  // Set rotation, flip, and speed based on the object's
  // current direction of movement.
  switch (m_direction)
  {
    orxBOOL flipX, flipY;
  case NORTH:
    speed.fY = -m_movementSpeed;
    SetRotation(270 * orxMATH_KF_DEG_TO_RAD);
    SetFlip(false, false);
    break;
  case SOUTH:
    speed.fY = m_movementSpeed;
    SetRotation(90 * orxMATH_KF_DEG_TO_RAD);
    SetFlip(false, false);
    break;
  case WEST:
    speed.fX = -m_movementSpeed;
    SetRotation(0 * orxMATH_KF_DEG_TO_RAD);
    SetFlip(true, false);
    GetFlip(flipX, flipY);
    break;
  case EAST:
    speed.fX = m_movementSpeed;
    SetRotation(0);
    SetFlip(false, false);
    GetFlip(flipX, flipY);
    break;
  default:
    orxASSERT(false);
  }

  // Update object's speed of movement
  SetSpeed(speed);

  // Time since direction change exceeds interval of direction change?
  if ((m_timeSinceDirectionChange += _rstInfo.fDT) >=
m_directionChangeInterval)
  {
    // Reset time
    m_timeSinceDirectionChange = 0;
    // Pick random number between bounds of Direction enum
    orxU32 randomNum = orxMath_GetRandomU32(0, static_cast<orxU32>(LAST));
```

```
    // Update object's direction of movement
    m_direction = static_cast<Direction>(randomNum);
  }
}
```

This is all the code we need to bring our enemy bug to life. The code comments should explain what is happening, but note the following:

- An instance of the EnemyBug class is created for every enemy bug object created. Recall in our Scene object we create 5 enemy bug objects as children. Therefore, 5 instances of EnemyBug are created. Each enemy bug shown on the screen has a class instance defining its behavior.
- This class makes use of the SetRotation, SetFlip, and SetSpeed functions defined in the ScrollObject base class.
- OnCreate is called when the object is first created. We didn't define a constructor, so data members must be initialized here.
- In OnCreate, we query values in config without pushing the object's section first. That's okay, because Scroll pushes the binding Orx object's config section as a convenience before calling OnCreate.
- We initialize our class members using the “custom” values we defined in config. While not strictly necessary, this is good data-driven design. It means we can adjust these variables and run again without recompiling.
- OnDelete is called when the object is deleted. We must provide a body for the function, but it does nothing in our case.
- Update is called on every frame. This is the interesting part of EnemyBug. In our case, we update its rotation and speed based on its currently direction of travel. [3)]

ScrollObject Callbacks and Accessors

- OnCreate, OnDelete, and Update are protected callbacks from the ScrollObject class. That means these functions are called by Orx when these events occur on the object, allowing you to easily override their behavior. You should never call these functions directly.
- SetRotation, SetFlip, and SetSpeed are public accessors from the ScrollObject class. You call them directly from other object class functions (and any other class can call them if it has a pointer to the class instance). The accessors correspond to the similar orxObject_ functions (e.g. SetPosition == orxObject_SetPosition).

If you want to see all the accessors and callbacks available for overriding, see the ScrollObject class interface in the ScrollObject.h file.

We've now programmed a much more interesting bug. If you run the game, however, you'll still see nothing but still objects. That's because we haven't yet told Scroll that we want our enemy bug objects to take on the behavior in our EnemyBug class. One step remains.

Telling Scroll about the Enemy Bug Class (Overriding BindObjects)

The ScrollBindObject function accepts as a template parameter a class deriving from ScrollObject. It accepts as a regular parameter an Orx config section name. Then it binds any

instance of the Orx object to the class.

Include `EnemyBug.h` after the inclusion of `Object.h` in `binding-of-objects.cpp`:

```
#include "EnemyBug.h"
```

Add the following lines to `binding_of_objects::BindObjects` in `BindingOfObjects.cpp` after the line for "`Object`":

```
  // Bind the EnemyBug class to the O-EnemyBug config section
  ScrollBindObject<EnemyBug>("O-EnemyBug");
```

The order of calls to `ScrollBindObject` does not matter as long as all of the necessary bindings occur in `BindObjects`.

The `BindObjects` function is called when the game starts. It basically says, "Whenever we create an object of `O-EnemyBug` as defined in Orx config, make it take on all the behavior defined in the `EnemyBug` class."

Of course, the `EnemyBug` class must exist for this to work, which is why we made it first.

Run the game and you should see all your enemy bugs come to life.

Our Unfortunate Hero (Another Derived ScrollObject)

The bugs in our game must be very hungry! Let's add another bound object. First, add `Hero.h` to your project and write its interface.

```
#pragma once

#include "Object.h"

class Hero : public Object
{
public:
protected:
  void OnCreate();
  void OnDelete();
  void Update(const orxCLOCK_INFO &_rstInfo);
  void OnCollide(ScrollObject *_poCollider, orxBODY_PART *_pstPart,
orxBODY_PART *_pstColliderPart, const orxVECTOR &_rvPosition, const
orxVECTOR &_rvNormal);

private:
  orxFLOAT m_movementSpeed;
};
```

Hero has a similar interface, but notice we've added an override for `OnCollide`. This function will be called whenever Orx detects a physics collision between this object and another.

And add this code:

```
#include "Hero.h"

void Hero::OnCreate()
{
  // Get movement speed from config value
  m_movementSpeed = orxConfig_GetFloat("MovementSpeed");
}

void Hero::OnDelete()
{
  // Do nothing when deleted
}

void Hero::Update(const orxCLOCK_INFO &_rstInfo)
{
  // Use movement input to initialize a vector to scale movement speed
  orxVECTOR speed = {
      // Vector's x component is right - left input strength. It will
      // be 0.0 if the inputs are either inactive or both equally active.
      orxInput_GetValue("MoveRight") - orxInput_GetValue("MoveLeft"),
      // Vector's y component is down - up input strength. It will
      // be 0.0 if the inputs are either inactive or both equally active.
      orxInput_GetValue("MoveDown") - orxInput_GetValue("MoveUp"),
      0.0};

  // Normalize the input vector if it has a length > 1
  if (orxVector_GetSquareSize(&speed) > 1.0)
  {
    orxVector_Normalize(&speed, &speed);
  }

  // Scale the raw input vector by the our movement speed
  orxVector_Mulf(&speed, &speed, m_movementSpeed);

  // Update our speed
  SetSpeed(speed, false);
}

void Hero::OnCollide(ScrollObject *_poCollider, orxBODY_PART *_pstPart,
orxBODY_PART *_pstColliderPart, const orxVECTOR &_rvPosition, const
orxVECTOR &_rvNormal)
{
  // Add flash effect
  AddFX("FX-Flash");
}
```

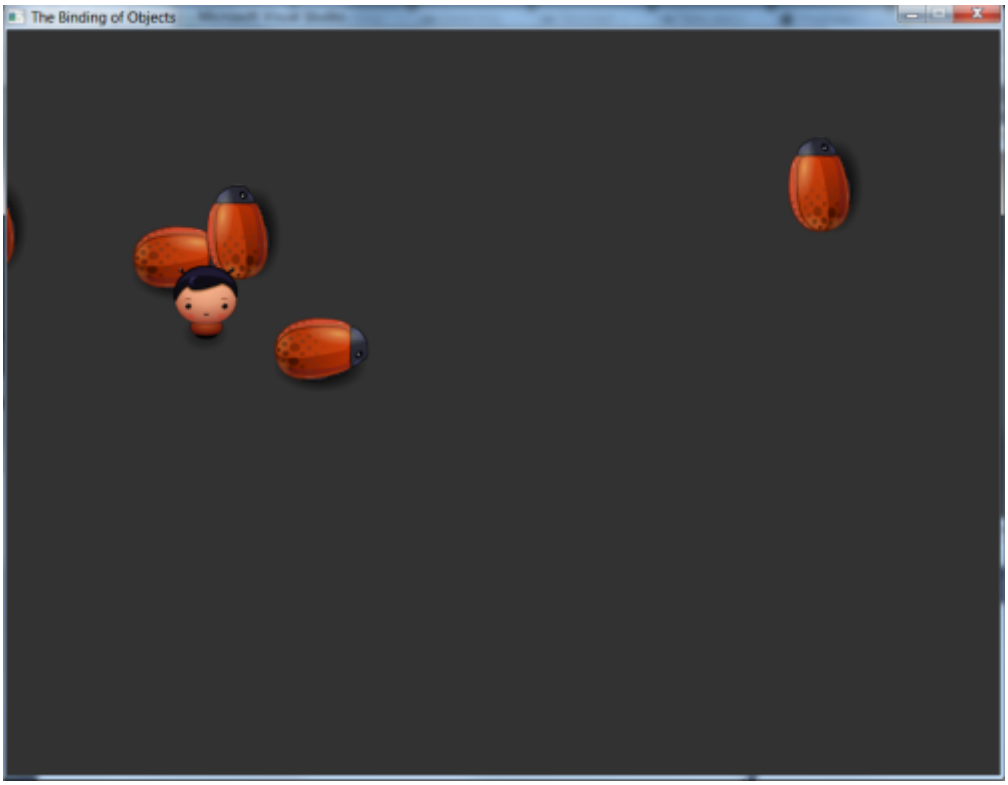
The code should be almost self-explanatory. The hero's movement speed will be pulled from its config value. The update function (called every frame) sets the speed of the character based on what keyboard arrow is pressed. The `OnCollide` function adds a “flash” effect to the character.

You have to modify the `BindingOfObjects::BindObjects` function to make it bind the new `Hero` class to the `O-Hero` object. Otherwise, the Hero will not be bound to its class and will just stand still in the middle of the screen!

Try to do those things yourself. If you need help, though, here are the lines to add:

```
  // Bind the Hero class to the O-Hero config section
  ScrollBindObject<Hero>("O-Hero");
```

When you run the game, you'll be able to control the hero with the arrow keys. Be careful, the bugs will bite him if he gets too close and the OnCollide callback will make him “flash” red.



What Now?

Well, you just finished making what could loosely be considered a game! Here are some additions you could make.

- Add a “Life” property to the Hero. Modify `OnCollide` so the Hero loses life each time he's hit. Give him an untimely death when life reaches 0.
- Give the Hero a weapon to fire at the bugs. The `OnCollide` callback for the enemy bugs can be used to make them take damage from the weapon.
- Be sure to add interesting death animations in the `OnDelete` callback.
- Add a more interesting background, of course.

1)

Thanks to Daniel Cook of www.lostgarden.com for the great prototyping graphics
2)

Overriding OnCreate and OnDelete is required for any class deriving from ScrollObject [3)]

Why use `SetSpeed` and not `SetPosition`? Because the objects in this tutorial have physics bodies. Calling `SetPosition` on objects with physics bodies messes up the physics simulation. If you want to watch that happen, change `SetSpeed` to `SetPosition` and turn on physics debugging as described in the config.