This is a document in progress. Intended to be both a video tutorial, and and article served from gamedev.net

Introduction

Welcome to the 2D UFO game guide using the Orx Portable Game Engine.

Hello, my name is Wayne, and my aim for this tutorial is to take you through all the steps to build a UFO game from scratch.

Before I begin, I should openly acknowledge that this series is cheekily inspired by the 2D UFO tutorial written for Unity.

This was chosen as it makes an excellent comparison of the approaches between Orx and Unity. It is also a perfect way to highlight one of the major parts that makes Orx unique among other game engines, its Configuration System.

You'll get very familiar with this system very soon. It's at the very heart of just about every game written using Orx.

If you are very new to game development, don't worry. We'll take it nice and slow and try to explain everything in very simple terms. The only knowledge you will need is some simple c++.

The aim of our game is to allow the player to control a UFO by applying physical forces to move it around. The player must collect pickups to increase their score to win.

1. Introduction to the 2D UFO Project

You will need a blank project for Visual Studio 2015, CodeLite or Code::Blocks in which you can use to work through the steps in this tutorial. We have one already prepared that you can use straight away. Download it here.

Once you have that, unzip the package to somewhere on your harddrive.

Inside the `build` folder, you'll find a `windows` folder. In there are three projects, one for Visual Studio, CodeLite and Code::Blocks.

Open the folder for the IDE you would like to use. Click MyGame.sln to open in Visual Studio, MyGame.workspace for CodeLite, or MyGame.workspace for Code::Blocks.

When the blank template loads, there are two main folders to note in your solution:

- config
- src

Firstly, the `src` folder contains a single source file, MyGame.cpp. This is where we will add the c++ code for the game. The `config` folder contains configuration files for our game.

What is config?

Orx is a data driven 2D game engine. Many of the elements in your game, like objects, spawners, music etc, do not need to defined in code. They can be defined (or configured) using config files.

You can make a range of complex multi-part objects with special behaviours and effects in Orx, and bring them into your game with a single line of code. You'll see this in the following chapters of this guide.

There are three config files in the config folder, but only one will actually be used in our game. This is:

MyGame.ini

All our game configuration will be done there.

There are two other config files:

- CreationTemplate.ini
- SettingsTemplate.ini

These are example configs and they list all the properties and values that are available to you. We will mainly concentrate on referring to the CreationTemplate.ini, which is for objects, sounds, etc.

The code template

Now to take a look at the basic MyGame.cpp and see what is contained there.

The first function is the Init() function.

This function will execute when the game starts up. Here you can create objects have been defined in the config, or perform other set up tasks like handlers. We'll do both of these soon.

The Run() function is executed every main clock cycle. This is a good place to continually perform a task. Though there are better alternatives for this, and we will cover those later. This is mainly used to check for the quit key.

The Exit() function is where things are cleaned up when your game quits. Orx cleans up nicely after itself. We won't use this function as part of this guide.

The Bootstrap() function is an optional function to use. This is used to tell Orx where to find the first config file for use in our game (MyGame.ini). There is another way to do this, but for now, we'll use this function to inform Orx of the config.

Then of course, the main() function. We do not need to use this function in this guide.

You should be able to compile successfully. Run the program and an Orx logo will appear slowly

rotating.

Great. So now you have everything you need to start building the UFO game.

2. Setting up the game assets

Our game will have a background, a UFO which the player will control, and some pickups that the player can collect.

The UFO will be controlled by the player using the cursor keys.

First you'll need the assets to make the game. You can download the gameassets.zip file which will contain:

The background file (background.png):

The UFO and Pickup sprite images (ufo.png and pickup.png):

And a pickup sound effect (pickup.ogg):

Unzip this file to a temporary location on your harddrive.

Copy the three .png files into your data/object folder

Copy the .ogg file into your data/sound folder.

Now these files can be accessed by your project and be included in the game.

Setting up the Playfield

We will start by setting up the background object. This is done using config.

Open the MyGame.ini config file in your editor and add the following:

```
[BackgroundGraphic]
Texture = background.png
Pivot = center
```

The `[BackgroundGraphic]` defined here is called a Graphic Section. It has two properties defined. The first is Texture which has been set as background.png.

The Orx library know where to find this image, due to the properties set in the `[Resource]` section:

```
[Resource]
Texture = ../../data/object
```

So any texture files that are required (just like in our [BackgroundGraphic] section) will be located in the `../../data/object` folder.

The second parameter is `Pivot`. A pivot is the handle (or sometimes “hotspot” in other frameworks). This is set to be center. The position is 0,0 by default, just like the the camera. The effect is to ensure the background sits in the center of our game window.

There are other values available for Pivot. To see the list of values, open the CreationTemplate.ini file in your editor. Scroll to the [GraphicTemplate] section and find Pivot in the list. There you can see all the possible values that could be used.

`top left` is also a popular value.

We need to define an object that will make use of this graphic. This will be the actual entity that is used in the game:

```
[BackgroundObject]
Graphic = BackgroundGraphic
Position = (0, 0, 0)
```

The Graphic property is the section `BackgroundGraphic` that we defined earlier. Our object will use that graphic.

The second property is the `Position`. In our world, this object will be created at (0, 0, 0). In Orx, the coordinates are (x, y, z). It may seem strange that Orx, being a 2D game engine has a Z axis. Actually Orx is 2.5D. It respects the Z axis for objects, and can use this for layering above or below other objects in the game.

To make the object appear in our game, we will add a line of code in our source file to create it.

In the Init() function of MyGame.cpp, remove the default line:

```
orxObject_CreateFromConfig("Object");
```

and replace it with:

```
orxObject_CreateFromConfig("BackgroundObject");
```

Compile and run.

The old spinning logo is now replaced with a nice tiled background object.

Next, the ufo object is required. This is what the player will control.

First create the configuration for the ufo object:

```
[UfoObject]
Graphic = UfoGraphic
```

```
Position = (0, 0, -0.1)
```

Then the `UfoGraphic` which the object needs:

```
[UfoGraphic]
Texture = ufo.png
Pivot = center
```

Unlike the background object, our ufo object will need to be assigned to a variable. This will make it possible to affect the ufo using code:

Create the variable for our ufo object just under the orx.h include line:

```
#include "orx.h"
orxOBJECT *ufo;
```

And in the Init() function, create an instance of the ufo object with:

```
ufo = orxObject_CreateFromConfig("UfoObject");
```

Compile and run. You'll see a ufo object in front of the background. Excellent.

Let's talk a little about the Z axis positions of the ufo, the background and the camera. It's worth explaining to get a good understand of how cameras and frustums work.

The following diagram will illustrate how our camera and objects are laid out:

Note that the Camera z-position in the config is: -1.0 The UFO z-position is: -0.1 The Background z-position is: 0.0

So 0.0 is the maximum distance in our game, and -1.0 is the potential minimum distance (the position of the camera's lens).

Also note the camera's Near and Far frustum range is 0.0 and 2.0 respectively. This range is a relative value based on the camera's z-position. Therefore, our camera can see objects between z-positions -1.0 and 1.0.

That's good, because our ufo and background are in that range: -0.1 and 0.0. As long as any objects have a z-position less than 0.0, they will be sure to be in front of the background.

And as long as any objects have a z-position greater than -1.0, the camera will show them.

Time to move to something a little more fun, moving the ufo.

Controlling the UFO

The ufo is going to be controlled using the cursor arrow keys on the keyboard.

The ufo will be moved by applying forces. Physics will be set up in the project in order to do this.

Also, we will use a clock to call an update function every frame (?) which will read and respond to key presses.

Defining the keys is very straight forward. In the config file, expand the `MainInput` section by adding the four cursor keys:

```
[MainInput]
KEY_ESCAPE = Quit
KEY_UP = GoUp
KEY_DOWN = GoDown
KEY_LEFT = GoLeft
KEY_RIGHT = GoRight
```

Each key is being given a label name. These label names are available in our code to test against.

The next step is to create an update callback function in our code where we will continually check if the keys are pressed:

```
void orxFASTCALL Update(const orxCLOCK_INFO *_pstClockInfo, void *_pContext)
{
}
```

And in order to tie this function to a clock (the clock will execute this function over and over), add the following to the Init() function:

```
orxClock_Register(orxClock_FindFirst(orx2F(-1.0f), orxCLOCK_TYPE_CORE),
Update, orxNULL, orxMODULE_ID_MAIN, orxCLOCK_PRIORITY_NORMAL);
```

That looks very scary and intimidating, but the only part that is important to you is the parameter with “Update”. This means, tell the existing core clock to continually call our “Update” function. Of course you can specify any function name here you like as long as it exists.

Let's test a key to ensure that all is working well. Add the following code into the Update function:

```
void orxFASTCALL Update(const orxCLOCK_INFO *_pstClockInfo, void *_pContext)
{
    if (ufo) {

        if (orxInput_IsActive("GoLeft")) {
        orxLOG("LEFT PRESSED!");
        }
    }
}
```

Every time Update is run, ufo is tested to ensure it exists, and then moves to check the input system for the label “GoLeft”, and if it is Active (or pressed). Remember how GoLeft is bound to KEY_LEFT in the MainInput config section?

If that condition is true, send “LEFT PRESSED!” to the console output window while the key is pressed

or held down.

Soon we'll replace the orxLOG line with a function that places force on the ufo. But before that, we need to add physics to the ufo.

Compile and run.

Press the left arrow key and take note of the console window. Every time you press or hold the key, the message is printed. Good, so key presses are working.

Physics

In order to affect the ufo using forces, physics need to be enabled.

Begin by adding a Physics config section and setting Gravity with:

```
[Physics]
Gravity = (0, 980, 0)
```

In order for an object in Orx to be affected by physics, it needs both a dynamic body, and at least one bodypart. Give the ufo a body with the `Body` property:

```
[UfoObject]
Graphic = UfoGraphic
Position = (0, 0, -0.1)
Body = UfoBody
```

Next, create the `UfoBody` section and define the `UfoBodyPart` property:

```
[UfoBody]
Dynamic = true
PartList = UfoBodyPart
```

The body part is set to `Dynamic` which means that it is affected by gravity and collisions. A body needs at least one part, and so we need to define the `UfoBodyPart`:

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

The body part `Type` is set to be a `sphere` which will automatically size itself around the object's size, and the body is to be solid so that if it should collide with anything, it will not pass through it.

Compile and Run.

The ufo falls through the floor. This is because of the gravity setting of 980 in the y axis which simulates world gravity.

Our game is a top down game. So change the `Gravity` property to:

```
[Physics]
Gravity = (0, 0, 0)
```

Re-run (no compile needed) and the ufo should remain in the centre of the screen.

The `Physics` section has another handy property available to visually test physics bodies on objects: `ShowDebug`. Add this property with true:

```
[Physics]
Gravity = (0, 0, 0)
ShowDebug = true
```

Re-run, and you will see a pinkish sphere outline automatically sized around the ufo object. For now we'll turn that off again. You can do this by changing the value to false, adding a `;` comment in front of the line or simply just deleting the line. We'll set our debug to false:

```
[Physics]
Gravity = (0, 0, 0)
ShowDebug = false
```

Let's add some force to the ufo if the left cursor key is pressed. Change the code in the Update function to be:

```
void orxFASTCALL Update(const orxCLOCK_INFO *_pstClockInfo, void *_pContext)
{
  if (ufo) {

    const orxFLOAT FORCE = 0.8;
    orxVECTOR leftForce= { -FORCE, 0, 0 };

    if (orxInput_IsActive("GoLeft")) {
        orxObject_ApplyForce(ufo, &leftForce, orxNULL);
    }
  }
}
```

The `orxObject_ApplyForce` function takes an orxVECTOR facing left and applies it to the ufo object.

Compile and re-run.

If you press and release the left arrow key, the ufo will move to the left. If you hold the left key down, the ufo will increase its speed and move out the left hand side of the screen.

Even if you tap the left key once quickly, the ufo will still eventually travel out of the left of the screen. There is no friction yet to slow it down, or any barriers to stop it going out of the screen.

Even though the background looks it has a border, it is really only a picture. In order to create a barrier for the ufo, we will need to wrap the edges using some body parts.

This means, the background object will also be given a body, and four body parts, one for each wall.

Start with adding a body to the object:

```
[BackgroundObject]
Graphic = BackgroundGraphic
Position = (0, 0, 0)
Body = WallBody
```

And then the body itself:

```
[WallBody]
Dynamic = false
PartList = WallTopPart # WallRightPart # WallBottomPart # WallLeftPart
```

This is different to the ufo body. This one is not dynamic. This means that it is a static body, one that cannot be affected by gravity. But dynamic objects can still collide with it. Also, there are four parts to this body, unlike the ufo which only had one.

Start with the WallTopPart first:

```
[WallTopPart]
Type = box
Solid = true
TopLeft = (-400, -300, 0)
BottomRight = (400, -260, 0)
```

In this part, use a box body part. Ensure it is solid for collisions, ie so that a dynamic object can collide with it but not pass though it.

Stretch the box to cover the region from (-400,-300) to (400, -260)

At this point, it might be a good idea to turn on the physics debugging to check our work:

```
[Physics]
Gravity = (0, 0, 0)
ShowDebug = true
```

Re-run the project.

The top wall region should cover the top barrier squares:

[]

Great. Next, we'll do the right hand side. But rather than copy all the same values, we'll reuse some from the top wall:

```
[WallRightPart@WallTopPart]
TopLeft = (360, -260,0)
BottomRight = (400, 260, 0)
```

Notice the `@WallTopPart` in the section name? This means: copy all the values from WallTopPart, but any properties in WallRightPart will take priority.

Therefore, use the `Type`, and `Solid` properties from `WallTopPart`, but use our own values for `TopLeft` and `BottomRight` for the `WallRightPart` section.

This is called “Section Inheritance”. This will come in very handy soon when we tweak values or add new properties to all four wall parts.

Re-run the project, and there will now be two walls.

Define the last two walls using the same technique:

```
[WallBottomPart@WallTopPart]
TopLeft = (-400,260,0)
BottomRight = (400, 300, 0)

[WallLeftPart@WallTopPart]
TopLeft = (-400,-260,0)
BottomRight = (-360, 260, 0)
```

Now there are four walls for the ufo to collide with.

Re-run and try moving the ufo left into the wall.

Oops, it doesn't work. It still passes straight though. There is one last requirement for the collision to occur: we need to tell the physics system, who can collide with who.

This is done with flags and masks.

Make a change to the ufo's body part by adding `SelfFlags` and `CheckMask`:

```
[UfoBodyPart]
Type = sphere
Solid = true
SelfFlags = ufo
CheckMask = wall
```

`SelfFlags` is the label you assign to one object, and `CheckMask` is the list of labels that your object can collide with.

These labels don't have to match the names you give objects, however it will help you stay clean and organised.

So above we are saying, the UfoBodyPart is a “ufo” and it is expected to collide with any bodypart marked as a “wall”.

But we haven't done that yet, so let's do it now. We will only need to add it to the `WallTopPart`:

```
[WallTopPart]
Type = box
Solid = true
SelfFlags = wall
CheckMask = ufo
TopLeft = (-400, -300, 0)
```

```
BottomRight = (400, -260, 0)
```

Remember, that the other three wall parts inherit the values from `WallTopPart`. So each now carries the label of “wall” and they will collide with any other body part that carries the label of “ufo”.

Re-run and notice the ufo's debug region becomes mauve rather than pink.

Press the left arrow key and drive the ufo into the left wall. It collides! And it stops.

Now that the collision is working, we can flesh out the rest of the keyboard controls and test all four walls:

```
void orxFASTCALL Update(const orxCLOCK_INFO *_pstClockInfo, void *_pContext)
{
  if (ufo) {

    const orxFLOAT FORCE = 0.8;

    orxVECTOR rightForce = { FORCE, 0, 0 };
    orxVECTOR leftForce = { -FORCE, 0, 0 };
    orxVECTOR upForce = { 0, -FORCE, 0 };
    orxVECTOR downForce = { 0, FORCE, 0 };

    if (orxInput_IsActive("GoLeft")) {
        orxObject_ApplyForce(ufo, &leftForce, orxNULL);
    }
    if (orxInput_IsActive("GoRight")) {
        orxObject_ApplyForce(ufo, &rightForce, orxNULL);
    }
    if (orxInput_IsActive("GoUp")) {
        orxObject_ApplyForce(ufo, &upForce, orxNULL);
    }
    if (orxInput_IsActive("GoDown")) {
        orxObject_ApplyForce(ufo, &downForce, orxNULL);
    }
  }
}
```

Now is a good time to turn off the physics debug as we did earlier on.

Compile and run.

Try all four keys, and you should be able to move the ufo around the screen. The ufo can also collide with each wall.

The ufo is a little boring in the way that it doesn't spin when colliding with a wall.

We need to ensure the UfoBody is not using fixed rotation. While this value defaults to false when not supplied, it will make things more readable if we explicitly set it:

```
[UfoBody]
Dynamic = true
```

```
PartList = UfoBodyPart
FixedRotation = false
```

The active ingredient here is to ensure that both the wall bodypart and the ufo bodypart both have a little friction applied. This way when they collide, they will drag against each other and produce some spin:

```
[UfoBodyPart]
Type = sphere
Solid = true
SelfFlags = ufo
CheckMask = wall
Friction = 1.2

[WallTopPart]
Type = box
Solid = true
SelfFlags = wall
CheckMask = ufo
TopLeft = (-400, -300, 0)
BottomRight = (400, -260, 0)
Friction = 1.2
```

Re-run that and give it a try. Run against a wall on angle to get some spin on the ufo.

The next thing to notice is that both the movement of the ufo and the spin never slow down. There is no friction to slow those down.

We'll deal with the spin first. By adding some `AngularDamping` on the `UfoBody`, the spin will slow down over time:

```
[UfoBody]
Dynamic = true
PartList = UfoBodyPart
AngularDamping = 2
FixedRotation = false
```

Re-run and check the spin. Should be slowing down after leaving the wall.

Now for the movement. That can be done with `LinearDamping` on the `UfoBody`:

```
[UfoBody]
Dynamic = true
PartList = UfoBodyPart
LinearDamping = 5
AngularDamping = 2
FixedRotation = false
```

Re-run and the speed will slow down after releasing the arrow keys. But it's slower overall as well. Not 100% what we want.

You can increase the FORCE value in code, in the Update function to compensate:

```
const orxFLOAT FORCE = 1.8;
```

Compile and run. The speed should be more what we expect.

Another thing to notice about the ufo is that during mid-rotation, the drawing of the object is a little rough. It would be nice to have some anti-alias (smoothing) on the object for rotations.

This is achieved with `Smoothing`:

```
[UfoObject]
Graphic = UfoGraphic
Position = (0, 0, -0.1)
Body = UfoBody
Smoothing = true
```

Another thing that would be nice, is for the ufo to be already spinning a little when the game starts. For this, add a little `AngularVelocity` :

```
[UfoObject]
Graphic = UfoGraphic
Position = (0, 0, -0.1)
Body = UfoBody
Smoothing = true
AngularVelocity = 200
```

Run this and the ship will have a small amount of spin at the start until the `AngularDamping` on he ufo body slows it down again.

Following the ufo with the camera

While we can simply move the ufo around with the keys on a fixed background, it will be a more pleasant experience to have the ufo fixed and have the screen scroll around instead.

This effect can be achieved by parenting the camera to the ufo so that wherever the ufo goes, the camera goes.

Currently, our project is set up so that the viewport has a camera configured to it. But the camera is not available to our code.

We will require the camera to be available in a variable so that it can be parented to the ufo object.

To fix this add a line in the Init() function to extract the camera from the viewport into a variable:

```
  orxVIEWPORT *viewport = orxViewport_CreateFromConfig("Viewport");
  camera = orxViewport_GetCamera(viewport);
```

And because the 'camera' variable isn't defined, do so at the top of the code:

```
#include "orx.h"
orxOBJECT *ufo;
orxCAMERA *camera;
```

Now it is time to parent the camera to the ufo:

```
ufo = orxObject_CreateFromConfig("UfoObject");
orxCamera_SetParent(camera, ufo);
```

Compile and Run.

Woah, hang on. That's crazy, the whole screen just rotated around when ufo. And it continues to rotate when hitting the ufo against the walls. See how the camera is a child of the ufo now? Not only does the camera move with the ufo, it rotates with it as well.

We certainly want it to move with the ufo, but it would be nice ignore the rotation from the parent ufo. Add the `IgnoreFromParent` property:

```
[Camera]
FrustumWidth  = 800
FrustumHeight = 600
FrustumFar    = 2.0 ; WHAT IS THE TEMPLATE USING???
FrustumNear   = 0.0
Position      = (0.0, 0.0, -0.1) ;;PROBLEM: IS OUR TEMPLATE -0.1 or -1.0????
IgnoreFromParent = rotation
```

Re-run. That's got it fixed.

Content to Delete:

Wait the screen went blank. That not good. What happened?

Something changed when we made the camera a parent of the ufo. When we did this, the camera's z position of -1.0 became relative to the ufo, and no longer relative to the world.

Now the camera's z position moves the camera's frustum out of viewing distance from the ufo and the background, giving a blank screen. The camera's z position will need to be adjusted to be closer to the ufo again:

```
[Camera]
FrustumWidth  = 800
FrustumHeight = 600
FrustumFar    = 1.0
FrustumNear   = 0.0
Position      = (0.0, 0.0, -0.1)
```

Re-run.

Now when you move around, the playfield will appear to scroll rather than it being the ufo that moves. This makes for a more dramatic and interesting effect.

Creating Pickup Objects

In our game, the ufo will be required to collect objects scattered around the playfield.

When the ufo collides with one, the object will disappear, giving the impression that it has been picked up.

Begin by creating a config section for the graphic, and then the pickup object:

```
[PickupGraphic]
Texture = pickup.png
Pivot = center

[PickupObject]
Graphic = PickupGraphic
```

The graphic will use the image pickup.png which is located in the project's data/object folder.

It will also be pivoted in the center which will be handy for a rotation effect later.

Finally, the pickup object uses the pickup graphic. Nice and easy.

Our game will have eight pickup objects. We need a simple way to have eight of these objects in various places.

We will employ a nice trick to handle this. We will make an empty object, called `PickupObjects` which will hold eight copies of the pickup object as child objects.

That way, wherever the parent is moved, the children move with it.

Add that now:

```
[PickupObjects]
ChildList = PickupObject1 # PickupObject2 # PickupObject3 # PickupObject4 #
PickupObject5 # PickupObject6 # PickupObject7 # PickupObject8
Position = (-400, -300, -0.1)
```

This object will have no graphic. That's ok. It can still act like any other object.

Notice the position. It is being positioned in the top left hand corner of the screen. All of the child objects PickupObject1 to PickupObject8 will be positioned relative to the parent in the top left corner.

Now to create the actual children. We'll use the inheritance trick again, and just use `PickupObject` as a template:

```
[PickupObject1@PickupObject]
```

```
Position = (370, 70, -0.1)

[PickupObject2@PickupObject]
Position = (210, 140, -0.1)

[PickupObject3@PickupObject]
Position = (115, 295, -0.1)

[PickupObject4@PickupObject]
Position = (215, 445, -0.1)

[PickupObject5@PickupObject]
Position = (400, 510, -0.1)

[PickupObject6@PickupObject]
Position = (550, 420, -0.1)

[PickupObject7@PickupObject]
Position = (660, 290, -0.1)

[PickupObject8@PickupObject]
Position = (550, 150, -0.1)
```

Each of the PickupObject* objects uses the properties defined in `PickupObject`. And the only difference between them are their `Position` properties.

The last thing to do is to create an instance `PickupObjects` in code in the Init() function:

```
orxObject_CreateFromConfig("PickupObjects");
```

Compile and Run.

Eight pickup objects should appear on screen. Looking good.

It would look good if the pickups rotated slowly on screen, just to make them more interesting. This is very easy to achieve in Orx using FX.

FX can also be defined in config.

FX allows you to affect an object's position, colour, rotation, scaling, etc, even sound can use FX.

Change the `PickupObject` by adding a `FXList` property:

```
[PickupObject]
Graphic = PickupGraphic
FXList = SlowRotateFX
```

Clearly being an `FXList` you can have many types of FX placed on an object at the same time. We will only have one.

An FX is a collection of FX Slots. FX Slots are the actual effects themselves. Confused? Let's work through it. First, the FX:

```
[SlowRotateFX]
SlotList = SlowRotateFXSlot
Loop = true
```

This simply means, use some effect called `SlowRotateFXSlot`, and when it is done, do it again in a loop.

Next the slot (or effect):

```
[SlowRotateFXSlot]
Type = rotation
StartTime = 0
EndTime = 10
Curve = linear
StartValue = 0
EndValue = 360
```

That's a few properties. First, the `Type`, which is a rotation FX.

The total time for the FX is 10 seconds, which comes from the `StartTime` and `EndTime` properties.

The `Curve` type is linear so that the values changes are done so in a strict and even manner.

And the values which the curve uses over the 10 second period starts from 0 and climbs to 360.

Re-run and notice the pickups now turning slowly for 10 seconds and then repeating.

The rotation redraw is a little rough, just like the ufo one was. What is a good solution? Add smoothing:

```
[PickupObject]
Graphic = PickupGraphic
FXList = SlowRotateFX
Smoothing = true
```

Re-run. Much better.

Picking up the collectable objects

Time to make the ufo collide with the pickups. In order for this to work (just like for the walls) the pickups need a body.

And the body needs to be set to collide with a ufo and vice versa.

First a body for the pickup template:

```
[PickupObject]
Graphic = PickupGraphic
FXList = SlowRotateFX
```

```
Smoothing = true
Body = PickupBody
```

Then the body section itself:

```
[PickupBody]
Dynamic = false
PartList = PickupPart
```

Just like the wall, the pickup are not dynamic. We don't want them bouncing and traveling around as a result of being hit by the ufo. They are static and need to stay in place if they are hit.

Next to define the `PickupPart`:

```
[PickupPart]
Type = sphere
Solid = false
SelfFlags = pickup
CheckMask = ufo
```

The pickup is sort of roundish, so we're going with a spherical type.

It is not solid. We want the ufo to able to pass through it when it collides. It should not influence the ufo's travel at all.

The pickup is given a label of `pickup` and will only collide with an object with a label of `ufo`.

The ufo must reciprocate this arrangement (just like a good date) by adding pickup to its list of bodypart check masks:

```
[UfoBodyPart]
Type = sphere
Solid = true
Friction = 1.2
SelfFlags = ufo
CheckMask = wall # pickup
```

This is a static bodypart, and we have specified collision actions to occur if the ufo collides with a pickup. But it's a little difficult to test this right now. However you can turn on the debug again to check the body parts:

```
[Physics]
Gravity = (0, 0, 0)
ShowDebug = true
```

Re-run to see the body parts.

Switch off again:

```
[Physics]
Gravity = (0, 0, 0)
```

```
ShowDebug = false
```

To cause a code event to occur when the ufo hits a pickup, we need something new: a physics hander. The hander will run a function of ours whenever two objects collide.

We can test for these two objects to see if they are the ones we are interested in, and run some code if they are.

First, add the physics hander to the end of the Init() function:

```
orxClock_Register(orxClock_FindFirst(orx2F(-1.0f), orxCLOCK_TYPE_CORE),
Update, orxNULL, orxMODULE_ID_MAIN, orxCLOCK_PRIORITY_NORMAL);
orxEvent_AddHandler(orxEVENT_TYPE_PHYSICS, PhysicsEventHandler);
```

This will create a physics handler, and should any physics event occur, (like two objects colliding) then a function called `PhysicsEventHandler` will be executed.

Our new function will start as:

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

  if (_pstEvent->eID == orxPHYSICS_EVENT_CONTACT_ADD) {
    orxOBJECT *pstRecipientObject, *pstSenderObject;

    /* Gets colliding objects */
    pstRecipientObject = orxOBJECT(_pstEvent->hRecipient);
    pstSenderObject = orxOBJECT(_pstEvent->hSender);

    const orxSTRING recipientName = orxObject_GetName(pstRecipientObject);
    const orxSTRING senderName = orxObject_GetName(pstSenderObject);

    orxLOG("Object %s has collided with %s", senderName, recipientName);

    return orxSTATUS_SUCCESS;
  }
}
```

Every handler function passes an orxEVENT object in. This structure contains a lot of information about the event.

The eID is tested to ensure that the type of physics event that has occurred is a orxPHYSICS_EVENT_CONTACT_ADD which indicates when objects collide.

If true, then two orxOBJECT variables are declared, then set from the orxEVENT structure. They are passed in as the hSender and hRecipient objects.

Next, two orxSTRINGs are declared and are set by getting the names of the objects using the orxObject_GetName function. The name that is returned is the section name from the config.

Potential candidates are: `UfoObject`, `BackgroundObject`, and PickupObject1 to PickupObject8.

The names are then sent to the console.

Finally, the function returns orxSTATUS_SUCCESS which is required by an event function.

Compile and run.

If you drive the ufo into a pickup or the edge of the playfield, a message will display on the console. So we know that all is working.

Next is to add code to remove a pickup from the playfield if the ufo collides with it. Usually we could compare the name of one object to another and perform the action.

In our case, however, the pickups are named different things: PickupObject1, PickupObject2, PickupObject3... up to PickupObject8.

So we will need to actually just check if the name contains “PickupObject” which will match well for any of them.

In fact, we don't need to test for the “other” object in the pair of colliding objects. Ufo is a dynamic object and everything else on screen is static. So if anything collides with PickupObject*, it has to be the ufo. Therefore, we won't need to test for that.

First, remove the orxLOG line. We don't need that anymore.

Change the function to become:

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

  if (_pstEvent->eID == orxPHYSICS_EVENT_CONTACT_ADD) {
    orxOBJECT *pstRecipientObject, *pstSenderObject;

    /* Gets colliding objects */
    pstRecipientObject = orxOBJECT(_pstEvent->hRecipient);
    pstSenderObject = orxOBJECT(_pstEvent->hSender);

    const orxSTRING recipientName = orxObject_GetName(pstRecipientObject);
    const orxSTRING senderName = orxObject_GetName(pstSenderObject);

    if (orxString_SearchString(recipientName, "PickupObject") != orxNULL) {
      orxObject_SetLifeTime(pstRecipientObject, 0);
    }

    if (orxString_SearchString(senderName, "PickupObject") != orxNULL) {
      orxObject_SetLifeTime(pstSenderObject, 0);
    }
  }

  return orxSTATUS_SUCCESS;
}
```

You can see the new code additions after the object names.

If an object name contains the word “PickupObject”, then the ufo must have collided with it. Therefore, we need to kill it off. The safest way to do this is by setting the object's lifetime to 0.

This will ensure the object is removed instantly and deleted by Orx in a safe manner.

Notice that the test is performed twice. Once, if the pickup object is the sender, and again if the object is the recipient.

Therefore we need to check and handle both.

Compile and run.

Move the ufo over the pickups and they should disappear nicely.

Sounds

It's great that collecting the pickups work, but a silent game is pretty bland. It would be great to have a sound play whenever a pickup is collected.

Start by configuring a sound:

```
[PickupSound]
Sound      = pickup.ogg
KeepInCache    = true
```

Then as part of the collision detection in the `PhysicsEventHandler` function, we change the code to be:

```
if (orxString_SearchString(recipientName, "PickupObject") != orxNULL) {
    orxObject_SetLifeTime(pstRecipientObject, 0);
    orxObject_AddSound(pstSenderObject, "PickupSound");
}

if (orxString_SearchString(senderName, "PickupObject") != orxNULL) {
    orxObject_SetLifeTime(pstSenderObject, 0);
    orxObject_AddSound(pstRecipientObject, "PickupSound");
}
```

In code above, if the recipient is a pickup object, then use the `orxObject_AddSound` function to place our sound on the sender object. There's little point adding a sound to an object that is about to be deleted.

And of course, if the pickup object is the sender, we add the sound to the recipient object. Also, the `PickupSound` that is added to the object, is the config section name we just defined in the config.

Compile and run.

Hit the pickups and a sound will play.

You can also use sounds without code. There is an `AppearSound` section already available in the

config.

We can use this sound on the ufo when it first appears in the game.

This is as simple as adding a `SoundList` property to the ufo:

```
[UfoObject]
Graphic = UfoGraphic
Position = (0, 0, -0.1)
Body = UfoBody
Smoothing = true
AngularVelocity = 200
SoundList      = SoundAppear
```

Re-run and a nice sound plays at the start of the game.

Adding a score

What's a game without a score? We need to earn points for every pickup that is collected.

The great thing about Orx objects is that they don't have to contain a texture as a graphic. They can contain a font and text rendered to a graphic instead. This is perfect for making a score object.

Start by adding some config for the `ScoreObject`:

```
[ScoreObject]
Graphic = ScoreTextGraphic
Position = (-380, -280, 0)
```

Next, to add the `ScoreTextGraphic` section, which will not be a texture, but text instead:

```
[ScoreTextGraphic]
Text = ScoreText
```

Now to define the `ScoreText` which is the section that contains the text information:

```
[ScoreText]
String = 10000
```

The `String` property contains the actual text characters. This will be the default text when a `ScoreObject` instance is created in code.

Let's now create an instance of the `ScoreObject` in the Init() function:

```
orxObject_CreateFromConfig("ScoreObject");
```

So far, the Init() function should look like this:

```
orxSTATUS orxFASTCALL Init()
{
  orxVIEWPORT *viewport = orxViewport_CreateFromConfig("Viewport");
  camera = orxViewport_GetCamera(viewport);

  orxObject_CreateFromConfig("BackgroundObject");
  ufo = orxObject_CreateFromConfig("UfoObject");

  orxObject_CreateFromConfig("PickupObjects");

  orxObject_CreateFromConfig("ScoreObject");

  orxClock_Register(orxClock_FindFirst(orx2F(-1.0f), orxCLOCK_TYPE_CORE),
Update, orxNULL, orxMODULE_ID_MAIN, orxCLOCK_PRIORITY_NORMAL);
  orxEvent_AddHandler(orxEVENT_TYPE_PHYSICS, PhysicsEventHandler);

  return orxSTATUS_SUCCESS;
}
```

Compile and run.

There should be a score object in the top left hand corner displaying: 10000

The score is pretty small. And it's fixed into the top left corner of the playfield. That's not really what we want.

A score is an example of a User Interface (UI) element. It should be fixed in the same place on the screen. Not move around when the screen scrolls.

The score should in fact, be fixed as a child to the Camera. So that wherever the Camera goes, the score object goes with it.

This can be achieved with the `ParentCamera` property, and then setting the position of the score relative to the camera's centre position:

```
[ScoreObject]
Graphic = ScoreTextGraphic
Position = (-380, -280, 0)
ParentCamera = Camera
UseParentSpace = false
```

With these changes, we've stated that we want the Camera to be the parent of the `ScoreObject`. In other words, we want the `ScoreObject` to travel with the Camera and appear to be fixed on the screen.

By saying that we don't want to `UseParentSpace` means that we want specify relative world coordinates from the centre of the camera. If we said yes, we'd have to specify coordinates in another system.

And `Position`, of course, is the position relative to the center of the camera. In our case, moved to the top left corner position.

Re-run and you'll see the score in much the same position as before, but when you move the ufo around, and the screen scrolls, the score object remains fixed in the same place.

The only thing, it's still a little small. We can double its size using `Scale`:

```
[ScoreObject]
Graphic = ScoreTextGraphic
Position = (-380, -280, 0)
ParentCamera = Camera
UseParentSpace = false
Scale = 2.0
```

Re-run. That looks a lot better.

To actually make use of the score object, we will need a variable in code of type int to keep track of the score.

Every clock cycle, we'll take that value and change the text on the ScoreObject.

That is another cool feature of Orx text objects. The text can be changed any time, and the object will re-render.

Finally, when the ufo collides with the pickup, and the pickup is destroyed, the score variable will be increased. The clock will pick up the variable value and set the score object.

Begin by creating a `score` variable at the very top of the code:

```
#include "orx.h"
orxOBJECT *ufo;
orxCAMERA *camera;
int score = 0;
```

Change the comparison code inside the `PhysicsEventHandler` function to increase the score by 150 points every time a pickup is collected:

```
if (orxString_SearchString(recipientName, "PickupObject") != orxNULL) {
  orxObject_SetLifeTime(pstRecipientObject, 0);
  orxObject_AddSound(pstSenderObject, "PickupSound");
  score += 150;
}

if (orxString_SearchString(senderName, "PickupObject") != orxNULL) {
  orxObject_SetLifeTime(pstSenderObject, 0);
  orxObject_AddSound(pstRecipientObject, "PickupSound");
  score += 150;
}
```

Now we need a way to change the text of the score object. We declared the score object in the Init() function as:

```
orxObject_CreateFromConfig("ScoreObject");
```

But we really need to create it using an orxOBJECT variable:

```
scoreObject = orxObject_CreateFromConfig("ScoreObject");
```

And then declare the `scoreObject` at the top of the file:

```
#include "orx.h"
orxOBJECT *ufo;
orxCAMERA *camera;
orxOBJECT *scoreObject;
int score = 0;
```

Now it is possible to update the score object using our score variable. At the bottom of the Update() function, add the following code:

```
if (scoreObject) {
  orxCHAR formattedScore[5];
  orxString_Print(formattedScore, "%d", score);

  orxObject_SetTextString(scoreObject, formattedScore);
}
```

First, the block will only execute if there is a valid scoreObject.

If so, then create a 5 character string. Then print into the string with the score value, effectively converting an int into a string.

Finally set the score text to the scoreObject using the `orxObject_SetTextString` function.

Compile and Run.

Move the ufo around and collect the pickups to increase the score 150 points at a time.

Winning the game

1200 is the maximum amount of points that can be awarded, and that is what will make us win the game.

If we do win, we want a text label to appear above the ufo, saying “You win!”.

Like the score object, we need to define a `YouWinObject`:

```
[YouWinObject]
Graphic     = YouWinTextGraphic
Position = (0, -60, 0.0)
Scale    = 2.0
```

Just like the camera, the `YouWinObject` is going to be parented to the ufo too. This will give the appearance that the YouWinObject is part of the ufo.

The `Scale` is set to x2.

The `Position` is set offset up in the y axis so that it appears above the ufo.

Next, the actual `YouWinTextGraphic`:

```
[YouWinTextGraphic]
Text = YouWinText
Pivot = center
```

And the text to render into the YouWinTextGraphic:

```
[YouWinText]
String = You Win!
```

We'll test it by creating an instance of the `YouWinObject`, putting it into a variable, and then parent it to the ufo in the Init() function:

```
orxObject_CreateFromConfig("PickupObjects");
scoreObject = orxObject_CreateFromConfig("ScoreObject");

ufoYouWinTextObject = orxObject_CreateFromConfig("YouWinObject");
orxObject_SetParent(ufoYouWinTextObject, ufo);
```

Then the variable:

```
#include "orx.h"
orxOBJECT *ufo;
orxCAMERA *camera;
orxOBJECT *ufoYouWinTextObject;
orxOBJECT *scoreObject;
int score = 0;
```

Compile and Run.

The “You win” text should appear above the ufo. Not bad, but the text it rotating with the ufo much like the camera did. We can ignore the rotation from the parent on this object too:

```
[YouWinObject]
Graphic    = YouWinTextGraphic
Position = (0, -60, 0.0)
Scale  = 2.0
IgnoreFromParent = rotation
```

Re-run. Interesting. It certainly isn't rotating with the ufo, but its position is still being taken from the ufo's rotation. We need to ignore this as well:

```
[YouWinObject]
Graphic            = YouWinTextGraphic
Position = (0, -60, 0.0)
Scale          = 2.0
```

```
IgnoreFromParent = position.rotation rotation
```

Good that's working right.

We want the “You Win!” to appear once all pickups are collected.

The `YouWinObject` object on created on the screen when the game starts. But we don't want it to appear yet. Only when we win. Therefore, we need to disable the object immediately after it is created:

```
orxObject_Enable(ufoYouWinTextObject, orxFALSE);
```

Finally, all that is left to do is add a small check in the `PhysicsEventHandler` function to test the current score after each pickup collision:

```
if (orxString_SearchString(recipientName, "PickupObject") != orxNULL) {
  orxObject_SetLifeTime(pstRecipientObject, 0);
  orxObject_AddSound(pstSenderObject, "PickupSound");
  score += 150;
}

if (orxString_SearchString(senderName, "PickupObject") != orxNULL) {
  orxObject_SetLifeTime(pstSenderObject, 0);
  orxObject_AddSound(pstRecipientObject, "PickupSound");
  score += 150;
}

if (orxObject_IsEnabled(ufoYouWinTextObject) == orxFALSE && score == 1200) {
  orxObject_Enable(ufoYouWinTextObject, orxTRUE);
}
```

We are checking two things: that the `ufoYouWinTextObject` is not yet enabled, and if the score is 1200.

If both conditions are met, enable the `ufoYouWinTextObject`.

Compile and run.

Move the ufo around and collect all the pickups. When all are picked up and 1200 is reached, the “You Win!” text should appear above the ufo signifying that the game is over and we have won.

And that brings us to the end! We have created a simple and complete game with some configuration and minimal code.

Congratulations!

I hope you enjoyed working through making the ufo game. Of course, there are many little extras you can add to give your game that little extra polish. So there a couple more chapters that you can follow along with if you wish.

Shadows

There are many ways to do shadows. One method is to use shaders... though this method is a little beyond this simple guide.

Another method, when making your graphics, would be to add an alpha shadow underneath. This is a good method if your object does not need to rotate or flip.

The method I will show you in this chapter is to have a separate shadow object as a child of an object. And in order to remain independent of rotations is to ignore them from the parent.

First a shadow graphic for the ufo, and one for the pickups:

Save these both into the `/data/objects` folder.

Then create config for the ufo shadow:

```
[UfoShadowGraphic]
Texture = ufo-shadow.png
Alpha   = 0.3
Pivot   = center
```

The only interesting part is the `Alpha` property. 0.1 would be almost completely see-through (or transparent), and 1.0 is not see-through at all, which is the regular default value for a graphic. 0.3 is fairly see-through.

```
[UfoShadowObject]
Graphic  = UfoShadowGraphic
Position = (20, 20, 0.05)
```

Set the `Position` a bit to the right, and down.

Next, add the `UfoShadowObject` as a child of the `UfoObject`:

```
[UfoObject]
Graphic         = UfoGraphic
Position        = (0,0, -0.1)
Body            = UfoBody
AngularVelocity = 200
Smoothing       = true
UseParentSpace  = position
SoundList       = AppearSound
ChildList       = UfoShadowObject
```

Run the project.

The shadow child is sitting properly behind the ufo but it rotates around the ufo, which is not right. We'll need to ignore the rotation from the parent:

```
[UfoShadowObject]
Graphic              = UfoShadowGraphic
Position             = (20, 20, 0.05)
IgnoreFromParent     = position.rotation rotation
```

Not only do we need to ignore the rotation of ufo, we also need to ignore the rotation position of the ufo.

Re-run and the shadow sits nice and stable to the bottom right of the ufo.

Now to do the same with the pickup shadow:

```
[PickupShadowGraphic]
Texture = pickup-shadow.png
Alpha   = 0.3
Pivot   = center

[PickupShadowObject]
Graphic          = PickupShadowGraphic
Position         = (20, 20, 0.05)
IgnoreFromParent = position.rotation
```

The only difference between this object and the ufo shadow, is that we want the pickup shadow to take the rotation value from the parent. But we do not want to take the position rotation.

That way, the pickup shadow will remain in the bottom right of the pickup, but will rotate nicely in place.

Now attach as a child to the pickup object:

```
[PickupObject]
Graphic   = PickupGraphic
FXList    = RotateFX
Smoothing = true
Body      = PickupBody
ChildList = PickupShadowObject
```

Re-run, and the shadows should all be working correctly.