# Tracks

Tracks are incredibly useful. But what can you use Tracks for?

Here's a couple of typical cases where you could use them.

1. The player has died and lost all three lives. The game is over. But the very split second that the player dies, you would not flash the “Game Over” banner across the screen. Normally the player is left with a few moments to suffer with their mistake, lives are 0, then after a second or two the Game Banner appears. Yep it's game over.

2. The player pressed go and the race is about to begin. A panel flashes up. It says “Get Ready”. Then 3, 2, 1, and Go. Each text fades to reveal the next. Then the final panel with Go disappears and the cars take off.

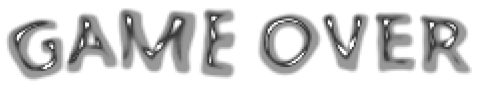
Imagine having to code that using orxEvents or orxClocks and managing lots of variables.

To build tracks, you should be familiar with the Command Console Syntax.

## Let's build scenario 1

Our objective will be: as soon as the game is over, start the track which will create the object, and fade it in. But the fade in will not occur until there is a initial delay of two seconds. There are two slightly different way to do this and we'll cover both.

You'll need a sample graphic for our game over banner:



And then the config for the banner object, and a fade in effect.

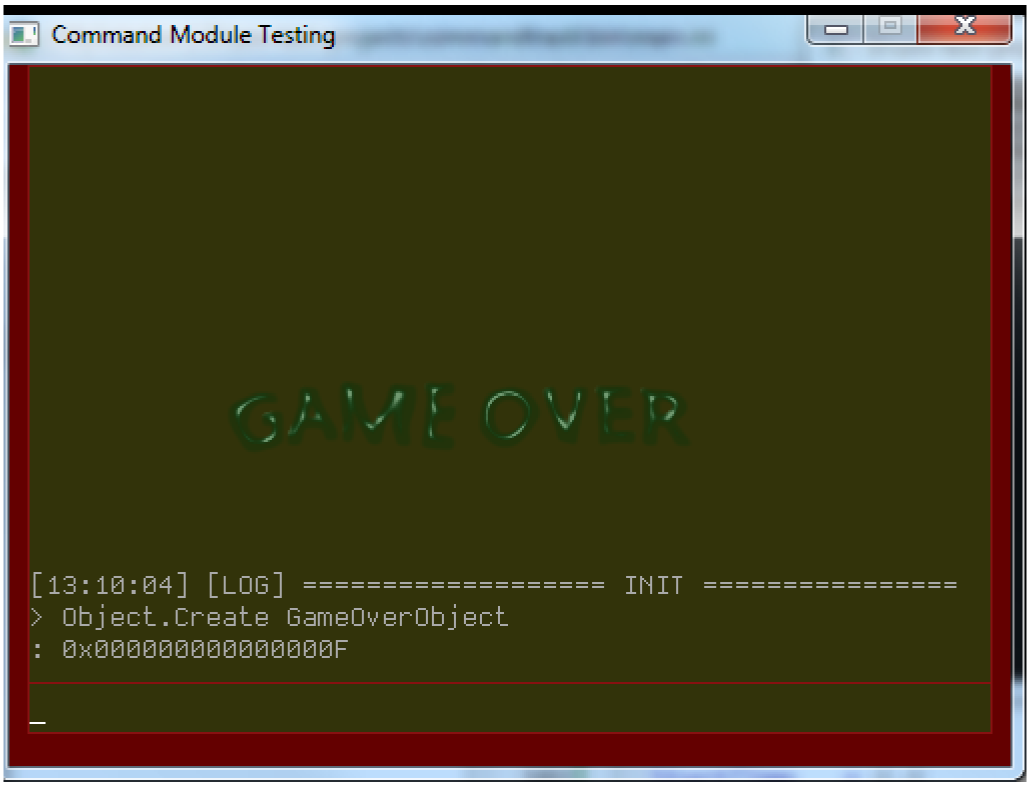
```
[GameOverObject]
Graphic = GameOverGraphic
Position = (200, 150, 0)
Alpha = 0 ;start faded out

[GameOverGraphic]
Texture = gameover.png


[FadeInFXSlot]
SlotList = FadeInFX

[FadeInFX]
Type        = alpha
StartTime   = 0.0
EndTime     = 0.5
```

```
Curve       = linear
Absolute    = true
Period        = 0.5
EndValue      = 1;
StartValue    = 0;
```

Compile and Run your project, and bring up the Console with the ` key.

Enter these commands:

```
>Object.Create GameOverObject
 Object.AddFX < FadeInFXSlot
```



```
[13:10:04] [LOG] ================== INIT ================
> Object.Create GameOverObject
: 0x000000000000000F
_
```

The first command creates the GameOverObject and places in on the screen in the position defined in the config file. Then the object is pushed onto the stack using the '>' symbol.

The second command pops the GameOver object off the stack and adds to the fade-in effect to it.

So that looks good. Let's take these commands and make an actual Track with it:

```
[GameOverTrack]
2 = > Object.Create GameOverObject # Object.AddFX < FadeInFXSlot
```

Looks a little different. In the track config above, both commands will occur after two seconds. After a two second delay, the GameOverObject will be created, pushed onto the stack, then popped off the stack so that the fade in FX can be applied.

We'll now make an empty object that will use the track:

```
[EmptyGameOverObject]
TrackList = GameOverTrack
```

So in code, if you create an instance of EmptyGameOverObject, after two seconds the GameOverObject is created and faded in:

```
orxObject_CreateFromConfig("EmptyGameOverObject");
```

Compile and Run. After 5 seconds, “Game Over” will fade in. Great!

The other method is to make the actual GameOverObject the one that owns the Track. So that when the GameOverObject is created, itself will fade in after two seconds.

```
[GameOverObject]
Graphic = GameOverGraphic
Position = (100, 150, 0)
Alpha = 0 ;start faded out
TrackList = GameOverTrack
```

For this to work, you need to know about the ^ symbol. This symbol means, “The Object That Owns This Track”.

And your track would become this instead:

```
[GameOverTrack]
2 = Object.AddFX ^ FadeInFXSlot
```

So the only differences between this and the last version of GameOverTrack is that the creation of the GameOverObject is removed, and the last command adds FX to ^, which is the object that owns this track, the GameOverObject.

Remove the EmptyGameOverObject from your config.

Change the created object line in your code to become:

```
orxObject_CreateFromConfig("GameOverObject");
```

Compile and Run again. Very nice.

## Let's build scenario 2

Here are some sample graphics to use for the example. Included is a graphic panel to display notices to the player:

Then the texts to use as individual graphics:

**GET READY**

**3**

**2**

**1**

**GO**

> Note: Using actual text as graphics is not recommended.
> Instead, look to using Orx's Text and Font to render bitmaps fonts
> from mapped characters: orxText.

Next, the config entries to cover the images and a fadeout effect:

```
[PanelObject]
Graphic = PanelGraphic
Position = (100, 150, 0)

[PanelGraphic]
Texture = readypanel.png


[GetReadyObject]
Graphic = GetReadyGraphic
Position = (190, 245, -0.1)

[GetReadyGraphic]
Texture = getready.png
```

```
[3Object]
Graphic = 3Graphic
Position = (255, 245, -0.1)

[3Graphic]
Texture = 3.png


[2Object]
Graphic = 2Graphic
Position = (255, 245, -0.1)

[2Graphic]
Texture = 2.png


[1Object]
Graphic = 1Graphic
Position = (255, 245, -0.1)

[1Graphic]
Texture = 1.png


[GoObject]
Graphic = GoGraphic
Position = (245, 245, -0.1)

[GoGraphic]
Texture = go.png


[FadeOutFXSlot]
SlotList = FadeOutFX

[FadeOutFX]
Type        = alpha
StartTime   = 0.0
EndTime     = 0.5
Curve       = linear
Absolute    = true
Period      = 0.5
EndValue    = 0;
StartValue  = 1;
```

Compile and run your project and bring up the Console with the ` key.

Enter the first two commands:

```
>Object.Create PanelObject
```

```
>Object.Create GetReadyObject
```

This will create the Panel object then the GetReady object and place them in the positions as per defined in the config file. Both objects will be pushed onto the stack using the '>' symbol. The last object pushed onto the stack was the GetReady object so the next command will affect it:

```
Object.AddFX < FadeOutFXSlot
```

The GetReady object is popped off the stack with the '<' symbol, then set to fade out by adding the FadeOutFXSlot effect.

We didn't push it back onto the stack with this command, so we cannot do anything further with it using '<' on subsequent commands. However, you can still press tab instead of '<' to recall the last ID.

```
>Object.Create 3Object
Object.AddFX < FadeOutFXSlot
```

Next, the 3 object is created, then faded out.

```
>Object.Create 2Object
Object.AddFX < FadeOutFXSlot
>Object.Create 1Object
Object.AddFX < FadeOutFXSlot
>Object.Create GoObject
Object.AddFX < FadeOutFXSlot
```

The same for the 2, 1, and Go objects.

```
Object.AddFX < FadeOutFXSlot
```

Finally, adding FX to the next object popped off the stack is the very first object pushed on there, which was, of course, the Panel object.

Right, so that was cool, but we should now look at taking our commands above and creating a track in the config file. It'll look something like this:

```
[ReadyGoTrack]
0 = >Object.Create GetReadyObject
1 = Object.AddFX < FadeOutFXSlot
3 = >Object.Create 3Object
4 = Object.AddFX < FadeOutFXSlot
6 = >Object.Create 2Object
```

```
7 = Object.AddFX < FadeOutFXSlot
9 = >Object.Create 1Object
10 = Object.AddFX < FadeOutFXSlot
12 = >Object.Create GoObject
13 = Object.AddFX < FadeOutFXSlot # Object.AddFX ^ FadeOutFXSlot
15 = Object.Delete ^
```

In the same vein as demonstrated in scenario 1 above, we won't create the PanelObject using the track, we'll make the PanelObject the owner of the track instead.

Attach the TrackList to the PanelObject and only create the GetReady, 3, 2, 1, and Go objects in the track.

That way, whenever an instance of PanelObject is created, all the magic happens automatically.

Now to make the PanelObject own the track:

```
[PanelObject]
Graphic = PanelGraphic
Position = (100, 150, 0)
TrackList = ReadyGoTrack
```

In your code, create an instance of the PanelObject and watch it go.

```
orxObject_CreateFromConfig("PanelObject");
```

Cool, so now when your game starts up, the PanelObject is created on screen and all the magic defined in the ReadyDoTrack track happens.

That's it!

But hang on. What we have created here is a sequence that relies heavily on the stack and is very error prone. It can be frustrating to debug.

A better option is to redo the above, but decentralise everything. This means: no longer using the stack, no push or popping, and moving the responsibilities to other objects and tracks.

All the config can be re-written as:

```
[ReadyGoTrack]
0 = Object.Create GetReadyObject
3 = Object.Create 3Object
6 = Object.Create 2Object
9 = Object.Create 1Object
12 = Object.Create GoObject
13 = Object.AddFX ^ FadeOutFXSlot
15 = Object.Delete ^
```

This new version of the ReadyGoTrack doesn't use the stack. No '>'s or '<'s. It will create the GetReady, 3, 2, 1, and Go objects as before. But how do we fade these objects? Notice, only the panel will be faded out at the end. The answer is the creation of a second Track:

```
[FadeOutTrack]
1 = Object.AddFX ^ FadeOutFXSlot
3 = Object.Delete ^
```

This is a generic track that can be used by any object. An object using it will be faded out after a second, and then deleted on the third second.

```
[FadeOutObject]
TrackList = FadeOutTrack
```

A generic object config containing the generic track. And then inheriting that in our objects:

```
[GetReadyObject@FadeOutObject]
Graphic = GetReadyGraphic
Position = (190, 245, -0.1)

[GoObject@FadeOutObject]
Graphic = GoGraphic
Position = (245, 245, -0.1)

[3Object@FadeOutObject]
Graphic = 3Graphic
Position = (255, 245, -0.1)

[2Object@FadeOutObject]
Graphic = 2Graphic
Position = (255, 245, -0.1)

[1Object@FadeOutObject]
Graphic = 1Graphic
Position = (255, 245, -0.1)
```

Compile and run and you'll see the same result as the previous version.

To best explain how this works:

The GetReady, 3, 2 objects etc all inherit the FadeOutObject object. This ensures they are all using TrackList = FadeOutTrack.

This means, if you were to create any of those object manually in code, or in the Console, they would fade out after a second and be deleted after three seconds.

So, in the ReadyGoTrack, every object it creates will automatically fade and die away.

This version may seem to do your head in, while the other one was more straight forward. However, our example used six different objects. What if we had 20? What if they all varied what effects were performed and the timings? In the old sequenced version, it would be very messy, hard to read and to debug.

This version scales better. Scaling is a fancy way of saying: as the amount of objects we add increases, and track gets larger with more adventurous effects added, the complexity does not increase. This is excellent for returning later to make changes or additions.

There is another problem with using the stack in Tracks. Because there is only a single stack, if you have more than one track pushing to/popping from the stack at once, you might get unexpected results.

Enjoy using tracks to simplify sequences of events in your game without having to code.

More details and examples of the Command Module syntax and examples can be found at:

- https://groups.google.com/forum/?hl=en#!topic/orx-dev/IK5cLrNVMPg
- https://github.com/iarwain/resource/blob/master/asset/data.ini?at=default
- commandnotes