Static Scene

Just to let you guys know, I've shamelessly ripped apart Iarwain's Tutorial 4 and a little of his Tutorial 2 for this, so feel free to take a look at what he's done in order to get a different perspective on what, why and how!

Right, let's to it!

We'll need a few files, I've copied these out of Iarwain's tutorial data files, but have made just those I use available for download here here. Drop the 'data' folder from in this zip, into our project folder (For me, this is looks like “C:\MyProject\data”.)

Next we're going to create a new file, and call it 'StaticScene.ini', this will go in the same place as all the other ini files (“C:\MyProject\bin”.)

I'm going to describe each chunk of information as I go, and try to give a reason for it so that it (hopefully) makes a little more sense.

Firstly, we're going to create a 'scene-graph', meaning our entire scene is a nice little hierarchy of objects. We'll have one object called our 'scene', and this will “contain” every other object. That way, when we want to load this scene in code, we'll only have to manually 'create' that one object, and the entire thing will just pop into existence for us.

StaticScene.ini

```
[Scene] ;=====================================
ChildList           = Soldier
```

And here it is... our [Scene] object. When we create new things to put in the scene, we simply add them to this object's childlist. Now, as you can see we're going to create a soldier. Lets do that now:

```
[Soldier] ;===================================
ChildList           = SoldierGraphics#SoldierNameTag
```

This is another container object, essentially the same as our [Scene] above, however this one is specifically for our Soldier, we've already added two items to its childlist, which we'll create in a moment, and a position. We want to be able to move the “whole” soldier around (and his name tag) at once, so instead of having to move everything manually, we can 'just' move this container.

```
[SoldierPivot] ;=============================
Pivot               = (16.0, 16.0, 0.0)     ; Move to centre of image on X
axis, bottom of image on Y axis (Soldier's feet!)

[SoldierGraphics@SoldierPivot] ;=============
Graphic             = StoppedFrame

[StoppedFrame] ;=============================
Texture             = ../data/anim/soldier.png
```

```
TextureSize           = (32, 32, 0)
```

This is the graphics for our soldier (or at least, the beginnings of them), this defines the default image that we'll see when the soldier is un-animated. The Pivot is also defined so we have a logical point to move our guy around with. In this case, approximately the base of his feet. We use @SomethingElse in the name of our object, to 'include' everything from the other object.

Thus [SoldierGraphics@SoldierPivot] is the same as saying 'Make [SoldierGraphics] with everything we've added AND everything from [SoldierPivot].

```
[SoldierNameTag] ;============================
Graphic               = SoldierName
Position              = (0.0, -32.0, 0.0)

[SoldierName] ;==============================
Text                  = SoldierNameString
Color                 = (255, 255, 255)
Pivot                 = center

[SoldierNameString] ;========================
String                = Soldier
```

And this is our name-tag for the soldier.

Right. Let's add some new code to our project.

We need to make only one minor change:

StandAlone.cpp

```
orxSTATUS orxFASTCALL StandAlone::Init()
{
    // Our automatically loaded configuration files include the data to make our camera
    //   and our viewport, we need to initialise those here.
    // The viewport will in turn create the camera, so we only need the one line of code.
    orxViewport_CreateFromConfig( "Viewport" );

    // Our new static scene is loaded from the following file.
    orxConfig_Load( "StaticScene.ini" );

        // We create the 'root' object of our scene, which in turn creates it's children, and their children etc.
    orxObject_CreateFromConfig( "Scene" );

    return orxSTATUS_SUCCESS;
```

```
}
```

Simple aye? Let's compile this and see what happens!

Not very interesting so far... fine, lets get something happening!

First, we're going to do the large job of adding our animation frames to our StaticScene.ini file:

StaticScene.ini

```
[SoldierGraphics@SoldierPivot] ;=============
Graphic              = StoppedFrame
AnimationSet         = AnimSet
```

First we modify our SoldierGraphics object, giving it access to our animation set.

```
[AnimSet] ;==================================
AnimationList        = IdleRight#WalkRight#IdleLeft#WalkLeft
LinkList             =
IdleRightLoop#IdleRight2Left#IdleRight2WalkRight#WalkRightLoop#WalkRight2Idl
eRight#IdleLeftLoop#IdleLeft2Right#IdleLeft2WalkLeft#WalkLeftLoop#WalkLeft2I
dleLeft
```

This horrifying looking thing is actually very simple. The first item “AnimationList” contains all the animations that our soldier can do. (Idle and walking back and forward.) The second item, is the list of all the 'links' between animations, meaning we don't want our soldier to be able to switch from walking left, to walking right, without stopping and 'turning around' first. So we make lists of animations that are 'allowed' to connect together.

```
[IdleRightLoop] ;============================
Source               = IdleRight
Destination          = IdleRight             ; Loops back onto itself.
[IdleRight2Left] ;===========================
Source               = IdleRight
Destination          = IdleLeft
Property             = immediate             ; Immediately go to IdleLeft
[IdleRight2WalkRight] ;======================
Source               = IdleRight
Destination          = WalkRight
Property             = immediate             ; Immediately go to WalkRight
```

This is our first set of animation 'links'. Essentially this tells us that when we are in the 'idle and facing right' animation (IdleRight), we are allowed to -continue- playing IdleRight (the loop), we are allowed to switch to IdleLeft (essentially, turn around), and we are allowed to immediately start walking right.

```
[WalkRightLoop] ;============================
Source               = WalkRight
Destination          = WalkRight
```

```
[WalkRight2IdleRight] ;=======================
Source                = WalkRight
Destination           = IdleRight
Property              = immediate
Priority              = 9
```

This is our list of links for our WalkingRight animation. Again, we're allowed to loop (walk forever), or, we're allowed to drop back to the 'IdleRight' animation. We're -not- being given permission to change into any other animation here!

```
[IdleLeftLoop] ;==============================
Source                = IdleLeft
Destination           = IdleLeft
[IdleLeft2Right] ;============================
Source                = IdleLeft
Destination           = IdleRight
Property              = immediate
[IdleLeft2WalkLeft] ;=========================
Source                = IdleLeft
Destination           = WalkLeft
Property              = immediate

[WalkLeftLoop] ;==============================
Source                = WalkLeft
Destination           = WalkLeft
[WalkLeft2IdleLeft] ;=========================
Source                = WalkLeft
Destination           = IdleLeft
Property              = immediate
Priority              = 9
```

Here are the left-facing versions of those same links. While I'm at it, the Priority value; The default value for this is 8. This is used to make the engine automatically step into this link when nothing else is asked for. In this case, what that means is that unless we -specifically- ask the animation to continue playing WalkLeft, it will finish its current playthrough, then switch to IdleLeft immediately afterwards.

The engine now knows what it can and cannot link together, but it doesn't yet know -what- it's supposed to be showing so next thing, we'll actually put the animation frames in!

```
[IdleRight] ;=================================
KeyData1              = AnimRight6
KeyDuration1          = 0.1
```

One whole frame here, delicious. We could put a few here and have the soldier 'fidget' while standing still, but for now, doing nothing is good enough.

```
[IdleLeft] ;==================================
```

```
KeyData1              = AnimLeft6
KeyDuration1          = 0.1
```

Left version, bet you didn't guess that! :)

```
[WalkRight] ;==================================
DefaultKeyDuration    = 0.1
KeyData1              = AnimRight1
KeyData2              = AnimRight2
KeyData3              = AnimRight3
KeyData4              = AnimRight4
KeyData5              = AnimRight5
KeyData6              = AnimRight6

[WalkLeft] ;==================================
DefaultKeyDuration    = 0.1
KeyData1              = AnimLeft1
KeyData2              = AnimLeft2
KeyData3              = AnimLeft3
KeyData4              = AnimLeft4
KeyData5              = AnimLeft5
KeyData6              = AnimLeft6
```

Here's a longer example, with 6 frames ie: our walk-left and walk-right animations! Each 'KeyData' value refers to a single frame of an animation, just in case you were wondering.

```
[FullGraphic@SoldierPivot] ;=================
Texture               = ../data/anim/soldier_full.png
TextureSize           = (32, 32, 0)
```

This is how we define where we are getting our image data from. We can either use a single image per frame (in which case we'll have lots of these) or as in our case, one image with lots of frames inside it.

```
[AnimRight1@FullGraphic] ;===================
TextureCorner         = (0, 0, 0)

[AnimRight2@FullGraphic] ;===================
TextureCorner         = (0, 32, 0)

[AnimRight3@FullGraphic] ;===================
TextureCorner         = (0, 64, 0)

[AnimRight4@FullGraphic] ;===================
TextureCorner         = (32, 0, 0)

[AnimRight5@FullGraphic] ;===================
TextureCorner         = (32, 32, 0)

[AnimRight6@FullGraphic] ;===================
```

```
TextureCorner         = (32, 64, 0)
```

Because we're using a single image with lots of frames, we need to define where in that image we get each frame from. TextureCorner is the corner of each frame, combined with our earlier “FullGraphic”, (which contained a texture size!) we can tell the engine to grab a 32×32 pixel 'frame' anywhere inside that entire image.

```
[AnimLeft1@AnimRight1] ;=====================
Flip                  = x

[AnimLeft2@AnimRight2]
Flip                  = x

[AnimLeft3@AnimRight3]
Flip                  = x

[AnimLeft4@AnimRight4]
Flip                  = x

[AnimLeft5@AnimRight5]
Flip                  = x

[AnimLeft6@AnimRight6]
Flip                  = x
```

Here we made our left facing versions of the frames. Simple aye?

Anyway, all that work now means that we've got our soldier ready to animate. What next? Oh, yes; lets actually make the code use them so that we don't just get a repeat of our last test run! :)

StandAlone.h

```
#include "orx.h" // We want to interface with ORX, so including it is
helpful! :)
#include <string> // Our 'GetObjectByName' and 'Update' functions both use
strings for the comparison tests.

class StandAlone
{
public:
    // Instance deals with ensuring we only have one copy of our StandAlone
class.
    static StandAlone* Instance();

    // Init function sets up our default/first scene.
    static orxSTATUS orxFASTCALL Init();
```

```
    // Run and Exit are both empty for now.
    static orxSTATUS orxFASTCALL Run();
    static void orxFASTCALL Exit();

    // GetObjectByName compares the names of the objects loaded in memory,
with a string we pass in
    //   either returning the first matching object, or a null pointer.
    static orxOBJECT* orxFASTCALL GetObjectByName( std::string objectName );

    // Update is called on each 'tick' of a clock.
    static void orxFASTCALL Update( const orxCLOCK_INFO* clockInfo, void*
context );

    // We currently use EventHandler to simply log when some of events
happen, so we know they're working.
    static orxSTATUS orxFASTCALL EventHandler( const orxEVENT* currentEvent
);

protected:
    StandAlone();
    StandAlone(const StandAlone&);
    StandAlone& operator= (const StandAlone&);

private:
    static StandAlone* m_Instance;
};
```

Here we add the definitions of three new functions, GetObjectByName is a helper function so that we can search through the list of objects we have created earlier, and grab specific ones. (This is the function that will be using the string library.)

The second is function that will be called whenever a orxCLOCK updates. We'll be putting our animation logic in here.

The third is used to make something happen whenever an event goes off (Animations have 4 events, Start, Stop, Loop and Cut - Cut being a 'forced stop')

StandAlone.cpp

We're going to add three new functions to our StaticScene class now.

```
orxOBJECT* StandAlone::GetObjectByName( std::string objectName )
{
    // This goes through the linked-list of all objects that have been
loaded, and compares
    //   their name with whatever string we pass in, returning the first
match,
    //   or a NULL if no match is found.
    std::string TempName;
    orxOBJECT* TempObject;
```

```
    for( // Grab the first object in our linked list, which is an orxOBJECT,
        TempObject = orxOBJECT( orxStructure_GetFirst(
orxSTRUCTURE_ID_OBJECT ) );
        // Until we get one that's NULL,
        TempObject != orxNULL;
        // Continue onto the next object,
        TempObject = orxOBJECT( orxStructure_GetNext( TempObject ) ) )
    {
        // Get the current object's name,
        TempName = orxObject_GetName( TempObject );
        // Compare that to our input string (objectName),
        if( TempName.compare( objectName ) == 0 )
            // and return that object if they're the same.
            return TempObject;
    }

    // or return a NULL instead.
    return orxNULL;
}
```

GetObjectByName, simply goes through the list of all objects that have been loaded into memory, and compares their name, to the string we pass in, if it finds the correct item, it will return a pointer to that object for us, or a NULL value if it fails.

```
void orxFASTCALL StandAlone::Update( const orxCLOCK_INFO* clockInfo, void*
context )
{
    // This is our update function, we're assigning this to our 'Soldier
Graphics' object
    //   so that whenever the clock 'ticks' for our soldier, we can make him
do something.
    //   In our case we're going to make him use the 'walk right' animation.

    // We use the orxOBJECT helper to ensure we get back the correct type,
or a NULL pointer.
    orxOBJECT* Object = orxOBJECT( context );

    // Lets make sure we found the object before trying to do anything with
it.
    if( Object != orxNULL )
    {
        // We grab out the object name, to use the STD::String comparion
function in a moment.
        std::string ObjectName = orxObject_GetName( Object );

        // Next we handle different objects, essentially, check their name
against what we expect
```

```
        //   and if we've got the right one, we can go from there.
        if( ObjectName.compare( "SoldierGraphics" ) == 0 )
        {
            // For now, we only want our character to use his 'walk right'
animation indefinitely
            //   so we'll just ensure that the target animation is always
the same.
            orxObject_SetTargetAnim( Object, "WalkRight" );
                }
        }
}
```

Update is the function that will be called whenever our clock “ticks”.

```
orxSTATUS orxFASTCALL StandAlone::EventHandler( const orxEVENT* currentEvent
)
{
    // We could just use a direct cast, but in case we expand this function
later to deal
    //   with different event types, we're going to be careful and make sure
we're in the
    //   correct event type first.
    switch( currentEvent->eType )
    {
        // This is for animation events, the only ones we handle in tutorial
3.
        case orxEVENT_TYPE_ANIM:
        {
            // Okay so we know we're dealing with an animation, lets grab
the 'payload' out of the event.
            orxANIM_EVENT_PAYLOAD* EventPayload =
(orxANIM_EVENT_PAYLOAD*)currentEvent->pstPayload;

            // We want the name of the object who called this event.
            //   (In tutorial 3, this will always be 'SoldierGraphics'.)
            const orxCHAR* ObjectName = orxObject_GetName( orxOBJECT(
currentEvent->hRecipient ) );

            // And finally the action which is this event.
            const orxCHAR* EventAct;

            switch( currentEvent->eID )
            {
                case orxANIM_EVENT_START: EventAct = "started"; break;
                case orxANIM_EVENT_STOP: EventAct = "stopped"; break;
                case orxANIM_EVENT_CUT: EventAct = "been cut"; break;
                //case orxANIM_EVENT_LOOP: EventAct = "looped"; break; // We
comment this line out to avoid some hefty console spam.
                                default: return orxSTATUS_SUCCESS; // Here
we return early to avoid trying to access something we've not specifically
set or loaded.
```

```
            }

            // Now we'll output this nicely to the console, so we can see
exactly what animations
            //   are being called, on what objects, and what they're doing.
            orxLOG("Animation <%s>@<%s> has %s!", EventPayload->zAnimName,
ObjectName, EventAct );
            break;
        }

    return orxSTATUS_SUCCESS;
}
```

This is essentially a rewritten version of Iarwain's event handling code from his tutorial 2. Animations send out many events to alert us what is going on, so we are logging this. (This code helped me debug while I was writing this tutorial, so it's very useful to have! )

Next we modify StandAlone::Init() again.

```
orxSTATUS orxFASTCALL StandAlone::Init()
{
        <---- SNIP ---->

                Old Code Goes HERE!

        <---- SNIP ---->

    // We're creating a clock here, this will be used later to update our
animations.
    orxCLOCK* AnimationClock = orxClock_Create( 0.2f, orxCLOCK_TYPE_USER );

    // We grab our 'Soldier Graphics' object here.
    // It is important to remember, that the object we want to modify is the
-animation object-
    //   in our case, this is 'Soldier Graphics' because that is where we
have our
    //   AnimationSet variable in the ini file.
    orxOBJECT* Soldier = GetObjectByName( "SoldierGraphics" );

    if( Soldier != orxNULL )
    {
        // Seems we found the right object, so lets register the clock to
our Soldier, and make him call
        //   the Update function every 'tick' of the clock.
        orxClock_Register( AnimationClock, Update, Soldier,
orxMODULE_ID_MAIN, orxCLOCK_PRIORITY_NORMAL );
        orxObject_SetTargetAnim( Soldier, "WalkRight" );
```

```
    }

    return orxSTATUS_SUCCESS;
}
```

Okay, time to compile and test again. I won't screenshot this again this time, but by now your little soldier should be running like a madman.