Interaction, Controls and Physics!

For this tutorial, we'll first add some keyboard controls, then later some physics.

These, are added to our *data* folder, along with everything from tutorial 3.

As usual, I'll be tearing apart Iarwain's tutorials and reshuffling them to my own thought patterns. Take a look at those for more “whys and hows”!

Interaction.ini (This is a new file!)

This is our newest file, here we're going to put in any new bits and pieces specific to this tutorial.

```
[Input] ;==================================
SetList             = SoldierInput

[SoldierInput] ;==============================
KEY_LEFT             = GoLeft
KEY_RIGHT            = GoRight
```

We'll be adding more to this later, but for now we're going to simply have left and right controls for our Soldier.

StandAlone.cpp

Let's add a few lines to our Init call.

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

                This is where the code from the tutorial 3 version of this
function will be; do not just copy & paste!! :)

        <---- SNIP ---->

    // Lets load up our tutorial 4: Interaction, file.
    orxConfig_Load( "Interaction.ini" );

    // Reload our input functionality, now that we've got some!
    orxInput_Load( orxSTRING_EMPTY );

    // Next we're going to ensure our input events get logged too!
    // We can use the same function we wrote earlier, because we were
careful to write
```

```
    //   it so that we could make easy changes.
    orxEvent_AddHandler( orxEVENT_TYPE_INPUT, StandAlone::EventHandler );

    return orxSTATUS_SUCCESS;
}
```

We load our new ini file, reset our input (orxInput_Load(orxSTRING_EMPTY) will force the engine to reevaluate all the loaded input values, including any new ones we've added.) and then set up our event handler to deal with Input events as well as Animation events.

Next is adding to our Update call.

```
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 'idle 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 )
        {
            /* -- Tutorial 3 : Static Scene and Animation --
            // 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" );
            */

            // Here we test if certain keys are active for tutorial 4.
            if( orxInput_IsActive( "GoRight" ) )
            {
                // As 'GoRight' is active, we need to change the animation
```

```
for our soldier, so he's
                //   playing the correct one. In this case 'WalkRight'!
                orxObject_SetTargetAnim( Object, "WalkRight" );
            }
            else if( orxInput_IsActive( "GoLeft" ) )
            {
                // This is essentially the same as above.
                orxObject_SetTargetAnim( Object, "WalkLeft" );
            }
            else
            {
                // This case is what happens when we have no active inputs.
                // For us, we want the animation to finish it's current
cycle, then return to an
                //   'idle' animation, facing the same direction.
                // Here is where all our hard work with the animation data
(remember the LinkList?)
                //   in the previous tutorial pays off, as this is all done
automatically!
                orxObject_SetTargetAnim( Object, orxNULL );
            }
        }
    }
}
```

Here we've removed our old 'always walk right' code, and updated it to choose it's animation based upon what input the player has active.

As I said in the comments, all that work we did in the previous tutorial, to set up a bunch of animations we weren't using, this pays off now with the final case (no input active). We can let the soldier automatically process his own animation system to decide what animations to play next. In our case, we set him up to go idle after walking (remember, we set his 'walk→idle' link to priority 9!) if we didn't specifically ask him to keep walking.

Alright, lets add input events to our EventHandler next.

```
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 )
    {
        <---- SNIP ---->

                This is where the code from the other tutorial versions of
this function will be; do not just copy & paste!! :)
```

```
        <---- SNIP ---->

        // Here we add event handling for tutorial 4, Input!
        case orxEVENT_TYPE_INPUT:
        {
            orxINPUT_EVENT_PAYLOAD* EventPayload =
(orxINPUT_EVENT_PAYLOAD*)currentEvent->pstPayload;

            const orxCHAR* EventAct;

            switch( currentEvent->eID )
            {
                case orxINPUT_EVENT_ON: EventAct = "pressed down"; break;
                case orxINPUT_EVENT_OFF: EventAct = "released"; break;
                default: return orxSTATUS_SUCCESS; // Remember to 'early
return' here, as there are more than just the two events we've specifically
handled right now.
            }

            orxLOG("Input <%s> has been %s!", EventPayload->zInputName,
EventAct );
            break;
        }
    }

    return orxSTATUS_SUCCESS;
}
```

Fairly straight forward what we've done here again. This will make sure we write a simple line to the console whenever one of our defined inputs is pressed, or released.

Time to compile and test again. You should see something lovely! (And no screenshots this time, don't want to spoil the surprise, right?)

Alright, we've now got some basic controls in place, so you can see what we did and why (hopefully!) Next, lets create our physics 'play pen'.

Interaction.ini

```
[Walls] ;===================================
ChildList           = LeftWall#RightWall#Ceiling#Floor

[WallGraphic] ;==============================
Texture             = ../data/phys/wall.png
Pivot               = center
```

This part you should recognise easily enough, so we'll just move on.

```
[VerticalWallGraphic@WallGraphic] ;==========
```

```
Repeat                = (1, 13, 1)            ; This tiles the graphic 13
times on the Y axis.

[HorizontalWallGraphic@WallGraphic] ;========
Repeat                = (22, 1, 1)            ; This tiles the graphic 22
times on the X axis.
```

This however is new. As you can see in the comments, the Repeat variable, allows us to tile the graphic in the same amount of space. Below is a small image to demonstrate the meaning.

The top texture is tiled twice on the X axis, the bottom image is the default texture. Notice the top box has been scaled down, to fit two times across the axis.

This would be written as:

```
Repeat                = (2, 1, 1)
```

For now we'll continue, but we will cover this again shortly.

```
[FullBoxPart] ;================================
Type                  = box                   ; This defines the bounding
type of the object
Restitution           = 0.0                   ; This defines the 'bounciness'
of the object
Friction              = 1.0
SelfFlags             = 0x0001                ; This is essentially the ID of
the collection of physics objects.
CheckMask             = 0xFFFF                ; This defines what OTHER ID
Flags this will collide with.
Solid                 = true
Density               = 1.0
; TopLeft             = (0.0, 0.0, 0.0)       ; Because we don't specify
these values, they will be automatically created
; BottomRight         = (1.0, 1.0, 1.0)       ;    based upon the object that
references it.
```

Here's our first physics defining object! We will use this for all our physics objects in this tutorial.

```
[WallBody] ;===================================
PartList              = FullBoxPart
```

This essentially sets us up an object which can have multiple physics parts. We could make a human with physics arms and legs for example. No object in this list will ever collide with any other in this list.

```
[WallTemplate] ;===============================
Body                  = WallBody
```

This simply saves us from having to specify the Body value in multiple other wall objects.

```
[VerticalWall@WallTemplate] ;================
Graphic             = VerticalWallGraphic
Scale               = @VerticalWallGraphic.Repeat ; The @<something>.Repeat
causes the extra graphics we specified earlier to spawn nearby based upon
the repeat value.
                                                  ; Essentially, you get a
line of boxes instead of many spawned on the same spot.

[HorizontalWall@WallTemplate] ;==============
Graphic             = HorizontalWallGraphic
Scale               = @HorizontalWallGraphic.Repeat ; Don't forget the @
symbol, you'll get strange errors if you do! :)
```

Okay, we're back to those strange 'Repeat' values. This bit of ini, as the comments say, allows us to actually move those extra graphics we spawned earlier, into a nice line, instead of just all spawning on top of one another. (When we test in a little while, I would recommend commenting out the 'scale' variable to see what this does exactly!)

```
[LeftWall@VerticalWall] ;====================
Position            = (-304.0, 0.0, 0.0)    ; -320 (half the screen width)
+ 16 (half the wall graphic size) = -304

[RightWall@VerticalWall] ;===================
Position            = (304.0, 0.0, 0.0)     ; 320 - 16 = 304... I think you
get the idea

[Ceiling@HorizontalWall] ;===================
Position            = (0.0, -224.0, 0.0)

[Floor@HorizontalWall] ;=====================
Position            = (0.0, 224.0, 0.0)
```

Okay, now we've done all the basic setup, we can create our actual walls. We're just going to put them around the outside of our screen for now.

project_d.ini & project.ini

For the first time in a -very- long time, we have to modify our BASE project config. As we're changing an initialisation value.

```
[Physics] ;==================================
AllowSleep          = false                 ; Normally our boxes would
settle and fall 'asleep', for now we'll disable this.
Gravity             = (0.0, 1000.0, 0.0)    ; A value of 1000 is roughly
Earth-like.
WorldUpperBound     = (300.0, 300.0, 1.0)   ; One thing you'll want to
remember is this Upper and Lower refers to the numbers you put here
WorldLowerBound     = (-300.0, -300.0, -1.0);  NOT the "upper" and "lower"
```

```
parts of the scene. Lower numbers go in the lower bounds variable!
```

Here, we turn on physics. :) The 'AllowSleep' value would, as the comment says, let the physics objects 'go to sleep' when they don't do anything for a little while. When they are 'asleep' the physics engine will not process them as much, so to avoid that we're just going to tell the engine that we need everything awake and processing. In a later tutorial I will show you how to handle this a much better way.

StandAlone.cpp

Okay, only a few additions this time.

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

                This is where the code from the other tutorial versions of
this function will be; do not just copy & paste!! :)

        <---- SNIP ---->

    // Now we're going to load up our new physics 'walls'
    orxObject_CreateFromConfig( "Walls" );

    orxEvent_AddHandler( orxEVENT_TYPE_PHYSICS, StandAlone::EventHandler );

    return orxSTATUS_SUCCESS;
}
```

As you can see, we've added a loader for our 'Walls', and a new event type to our event handler.

```
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 )
    {
        <---- SNIP ---->

                This is where the code from the other tutorial versions of
this function will be; do not just copy & paste!! :)

        <---- SNIP ---->

        // And physics events too!
```

```
        case orxEVENT_TYPE_PHYSICS:
        {
            orxPHYSICS_EVENT_PAYLOAD* EventPayload =
(orxPHYSICS_EVENT_PAYLOAD*)currentEvent->pstPayload;

            const orxCHAR* EventAct;

            switch( currentEvent->eID )
            {
                case orxPHYSICS_EVENT_CONTACT_ADD: EventAct = "touched";
break;
                case orxPHYSICS_EVENT_CONTACT_REMOVE: EventAct = "stopped
touching"; break;
                default: return orxSTATUS_SUCCESS;
            }

            orxLOG( "Physics; <%s> %s <%s>!", EventPayload->zSenderPartName,
EventAct, EventPayload->zRecipientPartName );
            break;
        }
    }

    return orxSTATUS_SUCCESS;
}
```

This is fairly cut and dry. We're doing the same thing as we've done the last couple of times with our Event handler. If there are any questions (as always!) Ask them on the forums and I'll be sure to update this tutorial to make those answers clear. (and answer on the forums too! )

Okay, if you compile and test now, you should see this:

Interaction.ini

Okay, lets add some physics toys!

```
[Input] ;=====================================
;SetList            = SoldierInput
SetList             = PhysicsInput

[SoldierInput] ;==============================
KEY_LEFT            = GoLeft
KEY_RIGHT           = GoRight
```

We comment out our old Input list, and add a new one here, so that we can add some new keys without replacing the old.

```
[PhysicsInput@SoldierInput] ;================
MOUSE_LEFT            = SpawnBox
```

We still want SoldierInput, so we've just included it. :)

```
[BoxBody] ;==================================
PartList              = FullBoxPart
Dynamic               = true                  ; (Default False) Setting this
to true causes the object to move according to the physics simulation.

[BoxGraphic] ;===============================
Texture               = ../data/phys/box.png
Pivot                 = center
```

Our 'Walls' didn't move, as you might have noticed. The 'Dynamic' variable makes all the difference here. If we make an object Dynamic, it will fall and bounce and move as the physics simulation asks it to, otherwise it will be set as Static, and act as a part of the “world”. Things will bounce off Static items, but static items do not move on their own.

```
[DynamicBox] ;=======================================
Graphic               = BoxGraphic
Position              = (-200.0, -200.0, 0.0) ~ (200.0, -200.0, 0.0) ; The
tilde (~) symbol is used to specify a value randomly chosen between the two
options given.
Body                  = BoxBody
Scale                 = 2.0
```

And here's the meat of our new Box object. The Scale value will resize the physics variable as well as the graphics variable. Something to be aware of; changing scale during runtime is more expensive when using Box2D physics (the default orx physics plugin) because it must destroy and recreate the physics object, as it does not currently support scaling directly.

StandAlone.cpp

Okay, we're going to change our EventHandler call now. I won't show the whole function, as we're only changing one piece.

```
// Here we add event handling for tutorial 4, Input!
case orxEVENT_TYPE_INPUT:
{
    orxINPUT_EVENT_PAYLOAD* EventPayload =
(orxINPUT_EVENT_PAYLOAD*)currentEvent->pstPayload;
    const orxCHAR* EventAct;

    switch( currentEvent->eID )
    {
        // We don't want this any more, lets comment it out
        //case orxINPUT_EVENT_ON: EventAct = "pressed down"; break;
```

```
        // NEW STUFF BELOW
        case orxINPUT_EVENT_ON:
        {
            if( orxInput_IsActive( "SpawnBox" ) )
                orxObject_CreateFromConfig( "DynamicBox" );

            EventAct = "pressed down";
            break;
        }
        // NEW STUFF ABOVE

        case orxINPUT_EVENT_OFF: EventAct = "released"; break;
        default: return orxSTATUS_SUCCESS; // Remember to 'early return'
here, as there are more than just the two events we've specifically handled
right now.
    }

    orxLOG("Input <%s> has been %s!", EventPayload->zInputName, EventAct );
    break;
}
```

Let's see what we get when we compile and run...

Interaction.ini

```
[PhysicsInput@SoldierInput] ;================
MOUSE_LEFT              = SpawnBox
MOUSE_RIGHT             = DeleteBox
```

Quick extra button, so we can clean up after ourselves. :)

StandAlone.h

```
class StandAlone
{
public:
        <---- SNIP ---->

                This is where the code from the other tutorial versions of
this constructor will be; do not just copy & paste!! :)

        <---- SNIP ---->

    // GetMouseWorldPosition gets the mouse 'screen' position, converts that
```

```
to 'world'
    //   coordinates, and returns the vector.
    static orxVECTOR orxFASTCALL GetMouseWorldPosition();

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

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

We've a new function declaration now, so let's fill it in!

StandAlone.cpp

```
orxVECTOR orxFASTCALL StandAlone::GetMouseWorldPosition()
{
    orxVECTOR MouseWorldPosition = { 0.0, 0.0, 0.0 };
    orxVECTOR MouseScreenPosition = { 0.0, 0.0, 0.0 };

    orxRender_GetWorldPosition( orxMouse_GetPosition( &MouseScreenPosition
), &MouseWorldPosition );

    return MouseWorldPosition;
}
```

Fairly simple function this one, just grabs the Screen Position and converts that to World Position.

'World' refers to the coordinates all our objects live in, 'Screen' refers to the coordinates inside the window we're clicking on.

WARNING: There is a bug in the Visual Studio 2008 compiler where this function is mis-optimized and your game may crash at a random place.

Next we'll make a little change to our EventHandler.

```
case orxINPUT_EVENT_ON:
{
    /* We no longer want to 'randomly' spawn boxes.
    if( orxInput_IsActive( "SpawnBox" ) )
        orxObject_CreateFromConfig( "DynamicBox" );
    */

    if( orxInput_IsActive( "SpawnBox" ) )
    {
        // Here we grab the mouse position,
        orxVECTOR MouseWorldPosition = GetMouseWorldPosition();
```

```
        // then create a new box.
        orxOBJECT* NewObject = orxObject_CreateFromConfig( "DynamicBox" );
        // Next we make sure the box is in the correct Z plane,
        MouseWorldPosition.fZ = 0.0f;
        // and finally, move the new box to our mouse pointer.
        orxObject_SetPosition( NewObject, &MouseWorldPosition );
    }

    if( orxInput_IsActive( "DeleteBox" ) )
    {

        // Let's grab the object under the mouse pointer right now,
        orxVECTOR MouseWorldPosition = GetMouseWorldPosition();
        orxOBJECT* PickedObject = orxObject_Pick( &MouseWorldPosition );
        if( PickedObject != orxNULL )
        {
            // then it's name,
            std::string ObjectName = orxObject_GetName( PickedObject );

            // and if it's a DynamicBox, we'll delete it!
            if( ObjectName.compare( "DynamicBox" ) == 0 )
                orxObject_Delete( PickedObject );
        }
    }

    EventAct = "pressed down";
    break;
}
```

Now, with all of that, you should be able to left-click to spawn boxes, and right-click to delete them.

Compile, and try it out, and hopefully get distracted for the next 5 minutes like I was, so you don't notice that this tutorial is over! :D