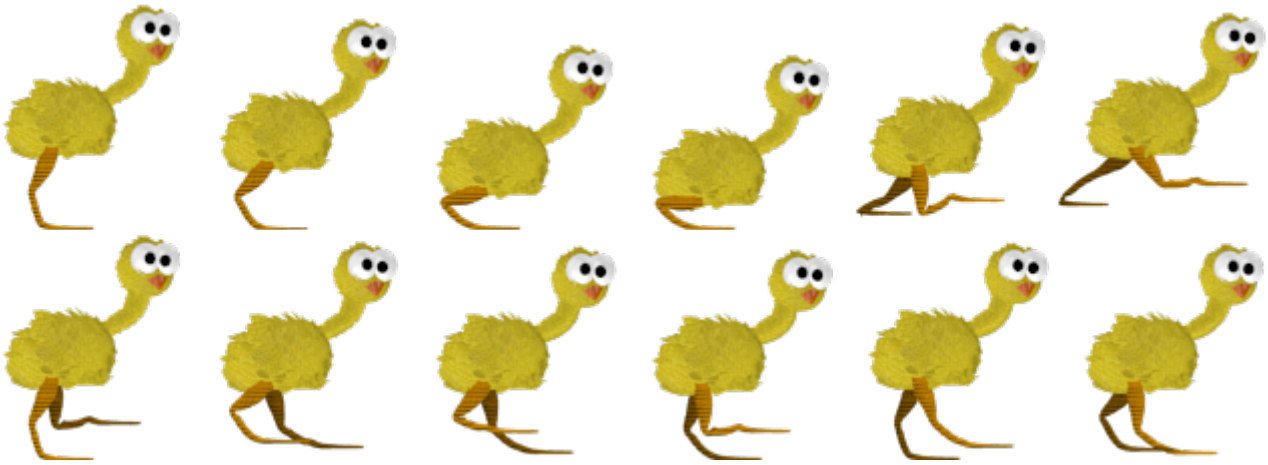
Animation Walk-through

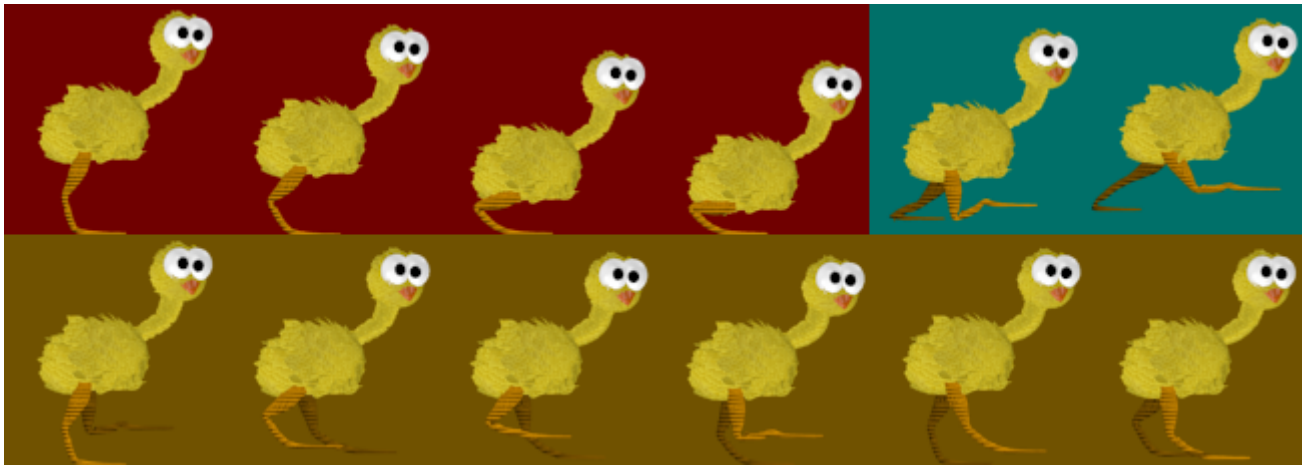
This walk-through is designed to guide you through the new Orx animation system. This new system is designed to be super easy to configure.

To follow along, ensure you have a working Orx project.

The sprite sheet that will be used throughout is below:



There will be five animations defined from this sprite sheet which are divided as follows:



The four frames in the red region will be used as the SitDown and StandUp animations. The two frames in the blue region will be the Jump animation. Finally, the six frames in the yellow region is the Run animation.

Lastly, when the chicken is doing nothing, he will sit idle with a single frame, the last one from the red region.

Basic Object Setup

Start with some basic object config:

```
[Chicken]
Graphic  = ChickenGraphic
Position = (400, 300, -0.1)

[ChickenGraphic]
Texture       = chicken-animation-sheet.png
TextureOrigin = (0, 0, 0)
TextureSize   = (108, 115, 0)
Pivot         = top left
```

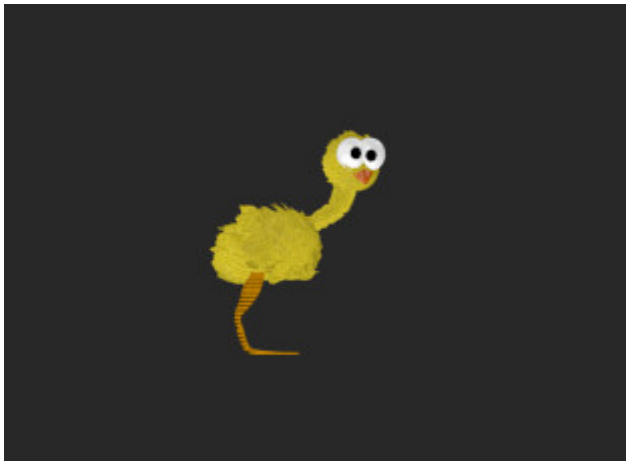
And create an instance from code:

```
chicken = orxObject_CreateFromConfig("Chicken");
```

Declare the chicken object variable at the top of your code:

```
orxOBJECT *chicken;
```

Compile and run, and we should have a basic chicken on the screen:



The Animation Set

This is the heart of Orx's animation system. Everything is connected through here. First, define an animation set on the object:

```
[Chicken]
Graphic      = ChickenGraphic
Position     = (400, 300, -0.1)
AnimationSet = ChickenAnimationSet
```



The optional `Graphic` property is only used to supply a default frame size for the animation frames. If you don't



define this, the first frame from your animation will supply the frame size.

Next, define the animation set, which is a very simple one:

```
[ChickenAnimationSet]
Texture     = chicken-animation-sheet.png
FrameSize   = (108, 115, 0)
StartAnim   = SitDownAnim
SitDownAnim = 4
```

The *Texture* is, of course, the sprite sheet. The *FrameSize* is the default size for each frame in the sprite sheet.

StartAnim is the name of the animation to start playing when the object is created. We only have a single animation so far which is called `SitDownAnim`.

Lastly, `SitDownAnim` (the name of our only animation) is set to 4. This means, capture the first four frames from the sprite sheet. And these four frames are the ones for sitting down.

The animation set now knows about `SitDownAnim`, simply because we created a property in the ChickenAnimationSet with that name, and we said how many frames to capture from the sheet.

However, to specify details about our particular animation, such as the KeyDuration, a config section for the animation name is required:

```
[SitDownAnim]
KeyDuration = 0.1
```

The only property in the animation is the duration between frames. But so much more can be specified. For now, we'll keep it simple to start with.

Re-run, and the chicken should be sitting down over and over:



Adding More Animations

Firstly, we will define the animation names, and how many frames need to be auto defined. Add the following to the `ChickenAnimationSet`:

```
[ChickenAnimationSet]
Texture     = chicken-animation-sheet.png
```

```
FrameSize   = (108, 115, 0)

StartAnim   = SitDownAnim

SitDownAnim = 4
StandUpAnim = 4
RunAnim     = 6
JumpAnim    = 2
IdleAnim    = 1
```

As already explained, `SitDownAnim` needs four frames from the sheet. In the same way, `StandUpAnim` will need four frames as well because it is going to use the same four frames, but in reverse.

`IdleAnim` will share the same group of frames as the two animations above, but only one frame from them.

`RunAnim` will need the bottom 6 frames of the sprite sheet. Jump will need 2 frames from the top right hand corner of the sprite sheet.

Of course, nothing we have defined yet tells Orx where to start sourcing the frames except for `SitDownAnim` which is starting by default from pixel position (0, 0, 0) in the sprite sheet.

Define the `StandUpAnim` as:

```
[StandUpAnim]
KeyDuration = 0.1
Direction   = left # up
TextureSize = (432, 115, 0)
```

For this animation the *Direction* property has been added. This changes the position where Orx gathers he frames from. We are telling Orx to gather the frames in the *left* direction. Effectively reversed.

But what about the *up* value? We'll come to that in a moment.

TextureSize is specified as well. Because of the *left # up* direction, the frame gathering will start at the bottom right hand of the sprite sheet, work backwards, then up.

This would incorrectly give us the chicken “running” frames, not the standing/sitting frames. TextureSize will reduce the gathering area to a single block of four frames in the top left hand of the sheet.

Just like reading the words in a book, we read from left to right, and down a line each time. In that case the value appropriate for a book would be: *right # down*.

In our case, TextureSize has restricted down to a single row of sprite frames, but Orx still expects two values. The value of *left # down* would have been fine as well.

The next animation to define is the `RunAnim`:

```
[RunAnim]
KeyDuration   = 0.1
TextureOrigin = (0, 115, 0)
```

This time there is a TextureOrigin property. As mentioned earlier, the ChickenAnimationSet keeps the amount of frames required for an animation from the sheet, but not the pixel coordinate to start from. (0, 0, 0) will be the default.

So supplying this coordinate means to gather the 6 frames, but start gathering them in the default Right direction starting at pixel position (0, 115, 0).

In the same way, the `JumpAnim` animation is defined as:

```
[JumpAnim]
KeyDuration   = 0.5
TextureOrigin = (432, 0, 0)
```

The *KeyDuration* is a little slower as he glides gracefully though the air.

Lastly our `IdleAnim` is defined as:

```
[IdleAnim]
KeyDuration   = 0.1
TextureOrigin = (324, 0, 0)
```



Please Note: Animation sections and Animation Frame sections (not demoed in this tutorial) are special and cannot use section inheritance like normal sections. AnimationSet sections however, can use section inheritance.



Although we are working with a small sheet of 12 frames, the Orx animation system scales beautifully, which is evident when using many animations with lots of frames.

Everything is correctly defined, but I'll show you a last trick with the auto frame defining before we move on. With `StandUpAnim` and `RunAnim` we told Orx to get 4 and 6 frames from the sprite sheet respectively:

```
StandUpAnim = 4
RunAnim     = 6
```

However, if you wish, you can change both of those to -1 like this:

```
StandUpAnim = -1
RunAnim     = -1
```

-1 means, get every frame that is available. Remember in the animation sections for StandUpAnim and RunAnim, we restricted the sprite sheet area using TextureOrigin and TextureSize. So -1 is a cleaner approach here, especially if you expand the sprite sheet size later.



- Greater than 0 = specific amount frames to be gathered
- -1 = as many frames as can fit inside the whole texture (or restricted size with TextureOrigin/TextureSize)
- 0 = as many frames as are defined in config (not demoed in this tutorial)

The animation definitions are done, but if you re-run your application, there is no change. We only have the *StartAnim* running.

Animation Links

It is time to plan how animations are to link from one to the other. There are three states for our chicken:

- When he is running
- When he is jumping
- When he is resting

When the chicken is resting, the IdleAnim will repeat itself over and over. But from this animation, he can also stand up.

When he is running, he can either continue to run, sit down, or jump.

When he is jumping, he will either end up running or sitting down.

When he is sitting down, he can only end up becoming idle.

That is a lot of detail. Fortunately, in the Animation Set section, we can plan all these transitions into lists. The first transition can be defined like this:

```
IdleAnim-> = IdleAnim # StandUpAnim
```

This syntax might seem a little odd at first.

IdleAnim→ means: “after IdleAnim completes, where do I go?”

= *IdleAnim* means: “set it to IdleAnim again as a priority, or StandUpAnim”

Based on the rule plan we set out a moment ago, the first draft of our linking will look like this:

```
IdleAnim->    = IdleAnim # StandUpAnim
StandUpAnim-> = RunAnim
RunAnim->     = RunAnim # SitDownAnim # JumpAnim
SitDownAnim-> = IdleAnim
JumpAnim->    = RunAnim # SitDownAnim
```

Notice how RunAnim has a list of possible animations to link off to? It could either be RunAnim again, SitDownAnim or JumpAnim. It could be any of them. But in this case, RunAnim would have priority to return back to itself.

Later we might have to make adjustments.

Re-run and the chicken will sit down and stay idle.

We can test the various animations by changing the *StartAnim* parameter. First let's test StandUpAnim:

```
StartAnim = StandUpAnim
```

Re-run, and the chicken will first stand up, and will proceed to start running. This is because in our link mapping, after standing up, he no other choice but to start running.

Now test the RunAnim:

```
StartAnim = RunAnim
```

The chicken will be running over and over. It would be nice to test idle to running. A little later we'll set up keyboard controls to test sitting, running and jumping.

Now test the JumpAnim:

```
StartAnim = JumpAnim
```

Re-run. The chicken will jump, then move to running over and over. Lastly, test the SitDownAnim:

```
StartAnim = SitDownAnim
```

The chicken will sit down, and then move to the idle animation over and over.

Great so everything appears to be working so far. If you would like to see the animation names display in the console (for debugging) when an animation starts, stops or loops, you can add an animation handler function like this one:

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

    if (_pstEvent->eType == orxEVENT_TYPE_ANIM){

        orxANIM_EVENT_PAYLOAD *pstPayload;
        pstPayload = (orxANIM_EVENT_PAYLOAD *)_pstEvent->pstPayload;
        const orxSTRING animName = pstPayload->zAnimName;
```

```
        if (_pstEvent->eID == orxANIM_EVENT_START ){
            orxLOG("orxANIM_EVENT_START %s", animName);
        }

        if (_pstEvent->eID == orxANIM_EVENT_LOOP ){
            orxLOG("orxANIM_EVENT_LOOP %s", animName);
        }

        if (_pstEvent->eID == orxANIM_EVENT_CUT ){
            orxLOG("orxANIM_EVENT_CUT %s", animName);
        }

        if (_pstEvent->eID == orxANIM_EVENT_STOP ){
            orxLOG("orxANIM_EVENT_STOP %s", animName);
        }
    }

    return orxSTATUS_SUCCESS;

}
```

And in your init() function:

```
orxEvent_AddHandler(orxEVENT_TYPE_ANIM, AnimationEventHandler);
```

A pretty handy routine to debug the animation transition names.

You might like to slow down all your animations to debug them, making it easier to see each frame, and the frames when they transition to another animation. You can do this using the *Frequency* property in the `ChickenAnimationSet`. To slow all animations down to a quarter of their normal speed add:

```
Frequency = 0.25
```

At any time, if you want to go back to normal animation speed, either comment out the *Frequency* property, let it to 1, or delete the property.

Before moving on, the animation that we should actually start with is the `IdleAnim`, so reset it back with:

```
StartAnim = IdleAnim
```

Testing Animations with Game Controls

Time to test some real-world controls to see if our links would work in a game environment. Our control rules will be:

- If a “run” key is pressed and held, the chicken will need to stand up and continue to run.
- If a “jump” key is pressed, the chicken will jump and then return to running if the “run” key is still held.
- If both keys are released, then the chicken will start sitting down, then become idle.

That will prove if our animation links are working. First, keys need to be defined:

```
[Input]
SetList = KeysForInput

[KeysForInput]
KEY_ESCAPE = Quit
KEY_RIGHT  = Run
KEY_LCTRL  = Jump
```

Add the following function to action the keys:

```
void CheckInputs(){
    if (orxInput_HasBeenActivated("Run")){
        orxObject_SetTargetAnim(chicken, "RunAnim");
    }
    else if (orxInput_HasBeenDeactivated("Run")){
        orxObject_SetTargetAnim(chicken, "IdleAnim");
    }
}
```

For the sake of ease, call the CheckInputs() function from the run() function:

```
orxSTATUS orxFASTCALL Run(){
  CheckInputs();

  ...
```

In the above, the “Run” key is checked to see if it is being pressed down. The second if statement checks if the “Run” key has been released.

In the first case, the target animation is set to `RunAnim` on the chicken object. Because the start animation is `IdleAnim`, the animation system needs to find a way to `RunAnim`.

If you consult the list of links in the `ChickenAnimationSet` you'll see that the three possible paths to `RunAnim` are:

1. StandUpAnim
2. RunAnim
3. JumpAnim

And the only link that can make it to `StandUpAnim` is:

- IdleAnim

Therefore, the link path to get from idle to run is:

`IdleAnim` → `StandUpAnim` → `RunAnim`

So re-compile and run again. This is the animation path that the chicken is expected to follow. Hold the Right Arrow key down and the chicken should stand up, and continue to run.

When releasing the “Run” key, the path the animation system will take is:

`RunAnim` → `SitDownAnim` → `IdleAnim`

So release the Right Arrow key and that's the path the chicken animations will take. After all that running, he'll sit down and stay seated.

So it's looking pretty good!

Next, to implement the jump. Add the following condition to the CheckInputs() function:

```
    if (orxInput_HasBeenActivated("Jump") && orxInput_IsActive("Run")){
        orxObject_SetTargetAnim(chicken, "JumpAnim");
    }
```

This new code checks if the “Jump” key was pressed, but only if the “Run” key is already held down.

Compile and run. Hold down the “Run” key to get the chicken running, then press and release the “Jump” key. Yay, our chicken jumps, and then continues to run.

But wait a second… he jumps again, then runs, then jumps, then runs. That's not right.

Clearing a Target Animation

We've come to our first problem in our link set up. Let's a take a close look at what could be going on here with the links:

```
IdleAnim->     = IdleAnim # StandUpAnim
StandUpAnim-> = RunAnim
RunAnim->      = RunAnim # SitDownAnim # JumpAnim
SitDownAnim-> = IdleAnim
JumpAnim->     = RunAnim # SitDownAnim
```

In code we set the target animation to `JumpAnim`. When releasing the jump key, the animation system used the links above to leave `JumpAnim`, but has to choose between `RunAnim` and `SitDownAnim`. Even though the `JumpAnim` has finished, it still remains the target!

Therefore the animation system will try and find a path from `JumpAnim` to `JumpAnim`, which is:

`JumpAnim` → `RunAnim` → `JumpAnim`

It is good know what is going on, but this is clearly not what we want. After jumping has completed, we want him just to go back to running again. Thankfully, it is simple to fix. We need to clear the

target when going to the JumpAnim like this:

```
RunAnim-> = RunAnim # SitDownAnim # !JumpAnim
```

The ! mark means: “when going from RunAnim, to JumpAnim, clear the target so we don't find the path back there again”.



Reminder: ! mark means to cleartarget

Change your config to the above and re-run.

Hold down the “Run” key to get the chicken running, then press and release the “Jump” key. The chicken jumps, and then continues to run over and over until the “Run” key is released.

Excellent so that works!

You may have noticed (especially when running your animations at a slower frequency) that the chicken will only jump after the run animation has completed. The chicken should, however, jump immediately.

Immediate Animations

We need to correct this. Fortunately Orx caters for this, and the change is ridiculously simple:

```
RunAnim-> = RunAnim # SitDownAnim # .!JumpAnim
```

Notice the full-stop “.” added to the JumpAnim? The “.” mark means: “when JumpAnim is set as the target, and we are currently in RunAnim, leave RunAnim and perform JumpAnim immediately”.



Reminder: . mark means to run this animation immediately - don't wait

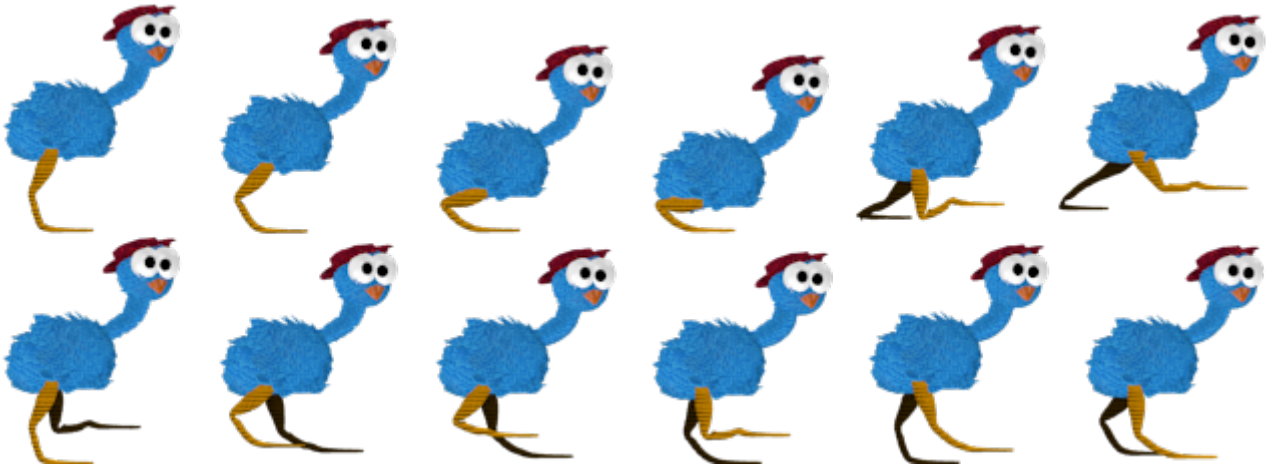
Change your config to the above and re-run. The chicken will run and jump immediately when told to.

Excellent. We are almost done. The chicken animations are all working correctly, and he behaves perfectly under control.

Another Sprite sheet for another version

One of the great advantages of Orx's animation system is the ability to create variations of a game character using the same animations and links, but supplying a different sprite sheet and using some simple section inheritance.

Here is another chicken, this one is blue with a nice red hat:



Save this to your data folder as: chicken-animation-sheet-blue.png

The first step is to create a new chicken object, and giving it a new *AnimationSet* property of `BlueChickenAnimationSet`:

```
[BlueChicken]
Graphic      = ChickenGraphic
Position     = (200, 300, -0.1)
AnimationSet = BlueChickenAnimationSet
```

Then create the `BlueChickenAnimationSet` section by inheriting the `ChickenAnimationSet`:

```
[BlueChickenAnimationSet@ChickenAnimationSet]
Texture = chicken-animation-sheet-blue.png
```

In code, change the name of the created object to:

```
chicken = orxObject_CreateFromConfig("BlueChicken");
```

Compile and re-run. The yellow chicken is replaced with a blue one. With some simple inheritance and a few extra sprite sheets, you can have ships or enemies that all vary on a theme, and use the same animation actions.

There are two things that were not covered in this tutorial section:

- The Prefix property
- Animation aliases

These two properties come in very handy when all the sprites are on the same sprite sheet. But having two separate sprite sheets avoided needing to use that. We'll cover these in another tutorial.

And now we're really done! I hope you enjoy using the animation system. Ask any questions on the forum, or come chat over on discord.

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://www.orx-project.org/wiki/en/tutorials/animation/animation_walkthrough?rev=1597924648

Last update: **2025/09/30 17:26 (14 months ago)**