Character Controller Pro 1.4.0 upgrade guide

Size interpolation (CharacterActor)

The built-in size interpolation functionality has been moved out from the *CharacterActor* component. This means that the user is now responsible for that task. In order to make this task easy, the actor now includes a few new methods dedicated to validate/interpolate size at the same time.

Here's an example taken from *NormalMovement.cs on* how to implement size lerping for the character:

```
void Crouch(float dt)
{
      // First, define the size reference (a.k.a the pivot)
      CharacterActor.SizeReferenceType sizeReferenceType = CharacterActor.IsGrounded ?
         CharacterActor.SizeReferenceType.Bottom :
crouchParameters.notGroundedReference;

      // Validate the new size and interpolate towards the goal
      CharacterActor.CheckAndInterpolateHeight(
         CharacterActor.DefaultBodySize.y * crouchParameters.heightRatio,
         crouchParameters.sizeLerpSpeed * dt, sizeReferenceType);

 }
```

Animator checks (CharacterStateController)

In previous versions, *CharacterStateController*'s *PreCharacterSimulation* and *PostCharacterSimulation* updates were supposed to be used for animation-related tasks (e.g. updating Animator parameters). This is the reason why a hidden set of Animator "checks" (if statements) was built-into the FSM. That being said, the user should be free to use these methods however he/she wants, especially because those methods can be extremely helpful when defining custom behaviors.

The user is now responsible to do the proper "checks" if needed. Fortunately, the actor now includes a method called *IsAnimatorValid* which does exactly that. Here's a simple example taken from *NormalMovement.cs*:

```
public override void PreCharacterSimulation(float dt)
{
     if (!CharacterActor.IsAnimatorValid())
         return;

     // ...
}

public override void PostCharacterSimulation(float dt)
{
     if (!CharacterActor.IsAnimatorValid())
         return;

     // ...
}
```

2D Rotation

In previous versions, a 2D character wasn't capable of doing yaw rotation (around its "up" direction). This was like this because of 2D Physics and the way the physics engine works internally. To compensate for this, a fake forward direction called "forward2D" was created for the actor. In addition, a *Rotator2D* component was added, whose only purpose was to follow this *Forward* vector and rotate the "graphics" accordingly.

In 1.4.0, the *Forward* direction matches with ± *transform.right*. In other words, the actor is now capable of doing yaw rotation (180 degrees) every time it needs to turn around.

Humanoids

2D humanoids characters don't need to rely on *Rotator2D* for yaw rotation anymore. In fact, this component does absolutely nothing for them. **You can safely remove it if you want**.

If smooth yaw interpolation (graphics rotation) is still required, you can add a *RotationLerper* (new component).

Sprites

When dealing with sprite-based characters, especially those built with more than one sprite, rotation is not something you'd want simply because the depth of the sprite will be completely messed up. Elements from the back will appear on the front, and vice versa.

A fix to this problem is to tweak the sign of the *localScale* property of the transform.

The Rotator2D component will do this for you just like in previous versions. **Even though there is not need to change anything, it might be best to replace *Rotator2D* with the new *SpriteRotator* component instead.**

Actor rotation

In previous versions, a 180 degrees rotation could be made by inverting any of the orthogonal direction (*Forward*, *Right* or *Up*):

```
CharacterActor.Forward *= -1f;
```

Even though this may look simple and effective, this was causing a lot of trouble for 2D. Since the entire rotation system has been improved, 3D rotation also ended up affected (in a good way).

From now on, the correct way to rotate an actor is by applying a quaternion 😱 … Wait, don't go away! The actor now includes many methods that will make this action super easy.

For example, this is how you make the character turn 180 degrees of **yaw rotation** (previous example):

```
// Turn 180 degrees
CharacterActor.TurnAround();
```

This is how you do yaw rotation by passing in a target forward direction:

```
// Look to the right
CharacterActor.SetYaw(CharacterActor.Right);

// Turn 180 degrees, same as TurnAround
CharacterActor.SetYaw(-CharacterActor.Forward);
```

Also you could specify pitch and roll rotations as well. There's even the option to pass in a pivot as an argument:

```
// Do pitch rotation using the actor center as a pivot
CharacterActor.RotatePitch(pitchValue, CharacterActor.Center);
```

In summary, now rotations are far more predictable.

Dynamic one way platforms (script)

The *OneWayPlatform* component **has been renamed** to *DynamicOneWayPlatform*. Since this component was only renamed, the file references (id) will match, assuming that the meta files were not modified. This means that you shouldn't get any missing references in the scene. In case you get a missing reference for whatever reason, please follow these simple steps:

1. Go to the missing component inspector.
2. Open the script field at the top
3. Select *DynamicOneWayPlatform.cs* as a replacement.