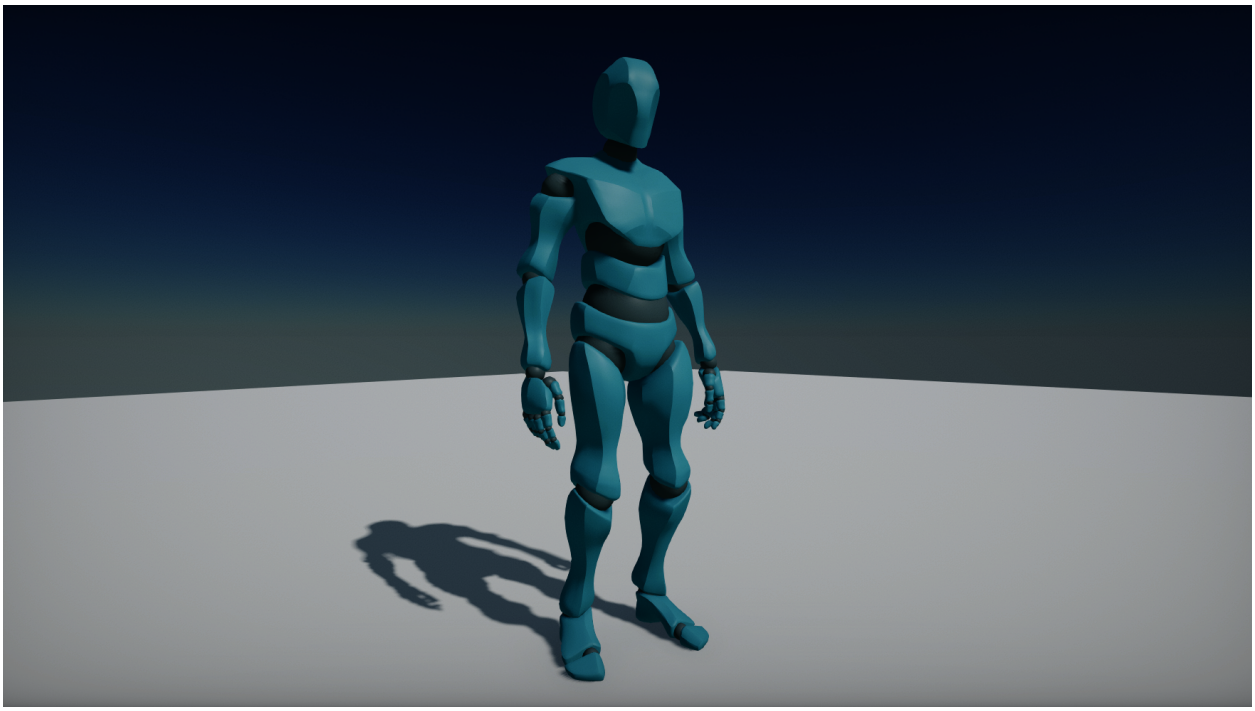
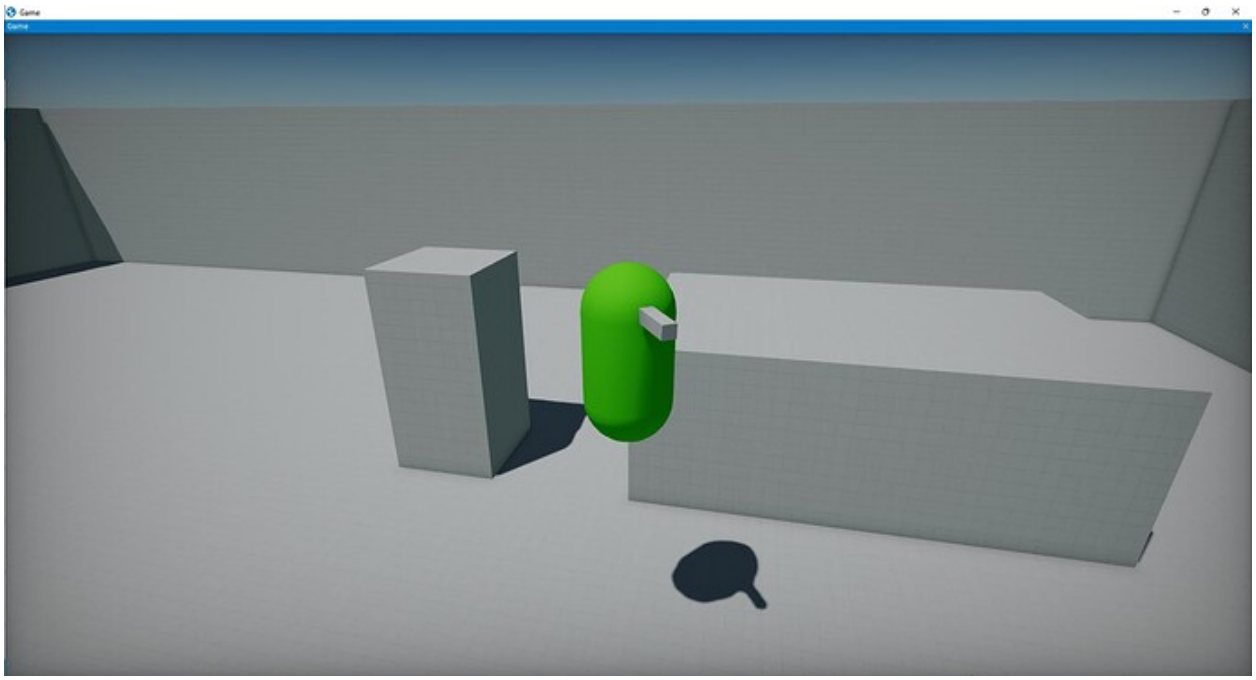


CharacterControllerPro User Manual

Overview

CharacterControllerPro is an open-source implementation of Flax Engine's *CharcaterController* class that adds complete movement functionality to it. The CharcaterController class exists as a modular base for us to build upon and create our own movement code. *CharacterControllerPro* takes this base and adds complete humanoid movement to it.

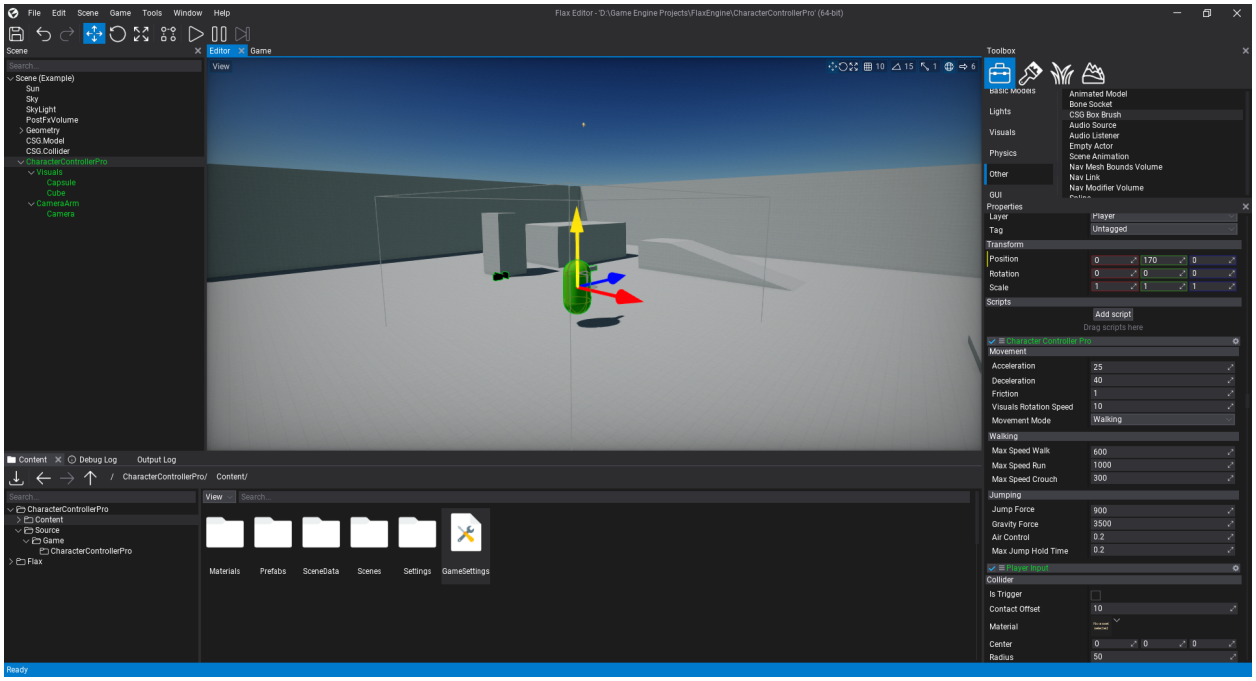
CharacterControllerPro implements acceleration, friction, gravity, multi-height jumping, sprinting, camera collision, camera rotation, input, and more features!



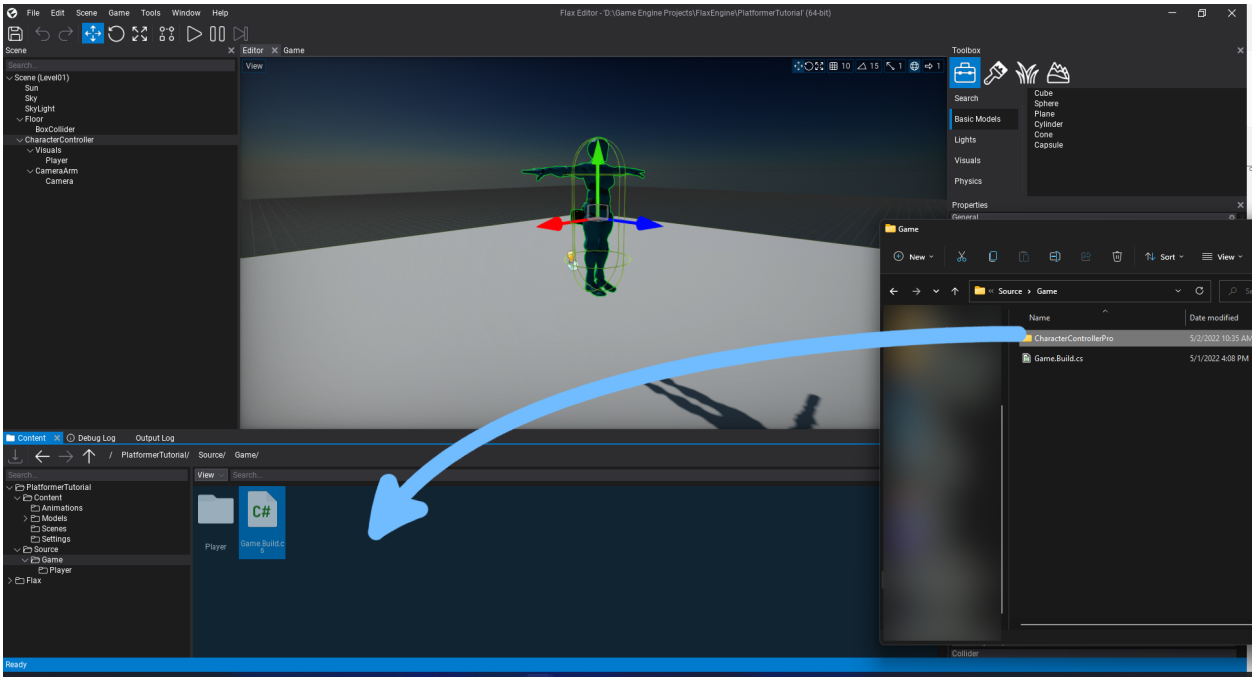
License

Setup

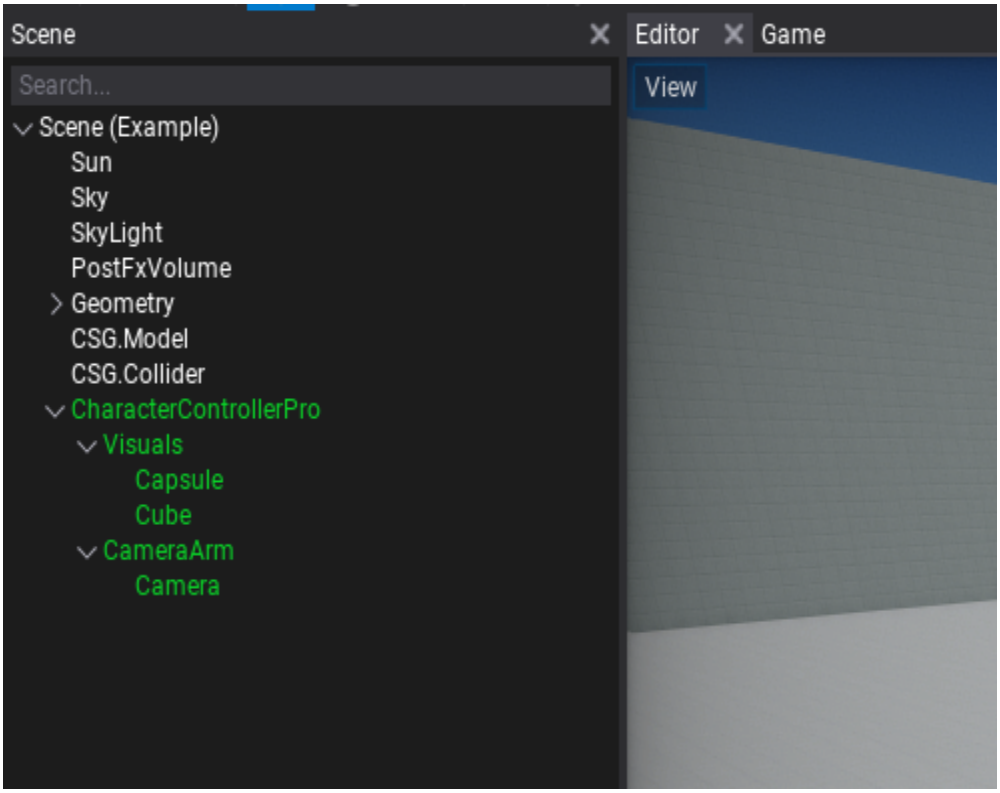
You can get the complete *CharacterControllerPro* project from its GitHub repository. The project contains the *CharacterControllerPro* source code, along with a sample scene that demonstrates how to construct a *CharacterControllerPro* player.



To use *CharacterControllerPro* in one of your projects, navigate to the *CharacterControllerPro* project's Source folder. Then, you can drag the CharcaterControllerPro folder into your project's Source folder.



Actor Hierarchy

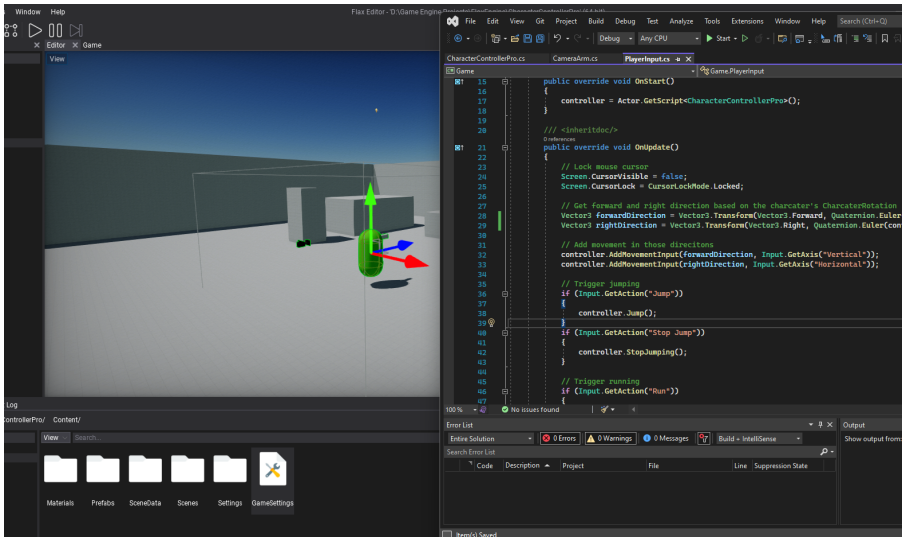


CharacterControllerPro is designed around a specific Actor hierarchy. To set up a new *CharacterControllerPro*, first add a new CharcaterController to the scene. Then, drag the *CharacterControllerPro* script onto it. So that the *CharacterControllerPro* can rotate the camera and character visuals separately, the visuals must be contained inside an empty Actor, named “Visuals” (Spelling is important), which has to be added to the *CharacterControllerPro*. All models, particles, etc. should be children of the “Visuals” Actor.

The camera can be placed as a direct child of the *CharacterControllerPro*, for a first person perspective, for example. However, CharcaterControllerPro also comes with a third person camera arm, which handles camera collision, rotation input, and more. The *CharacterControllerPro* will also rotate and move relative to the camera view. To add one to a *CharacterControllerPro*, add a new empty Actor to the *CharacterControllerPro*, and attach the CameraArm script to it. You will also need to add a Camera as a child of the camera arm Actor.

Using CharacterControllerPro

CharacterControllerPro is built around a design that allows for many different control schemes, whilst also being easy to use. It uses a custom design inspired in part by Unreal Engine's Character system, but it is different enough to have its own basic breakdown.



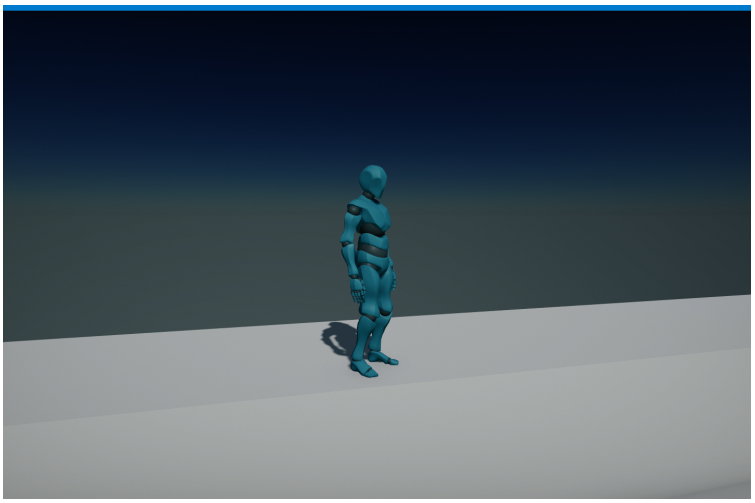
Character Rotation

Instead of rotating the entire Actor, *CharacterControllerPro* has a virtual rotation value, called CharacterRotation, that can be used to determine its rotation. This is done so that rotation of the camera and the model can be done separately. *CharacterControllerPro* never uses the CharacterRotation directly. Instead, its intended use is for using the camera's view direction as the axis to move the player along.

```
// Add character rotation
playerController.AddCharacterRotation(new Vector3(0, lookInput.X, 0));
```

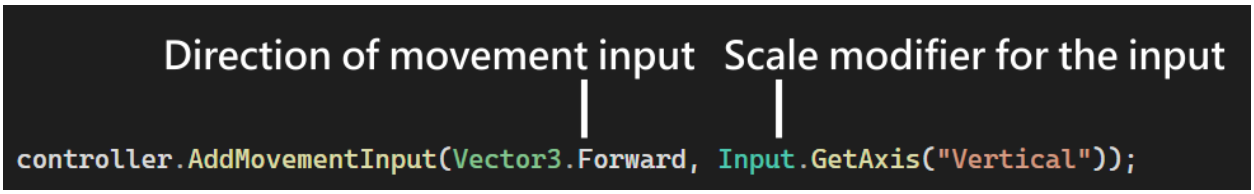
The included CameraArm script uses the mouse X input to rotate the CharcaterRotation's Y axis, or yaw. This rotation can then be used to determine the forward direction for input. A similar method can be used for a custom camera controller.

The *CharacterControllerPro* script rotates the "Visuals" Actor based on the input direction supplied, making its rotation independent of the camera. Notice how the character model can freely rotate towards the direction of the input, and how the camera can rotate around the character.

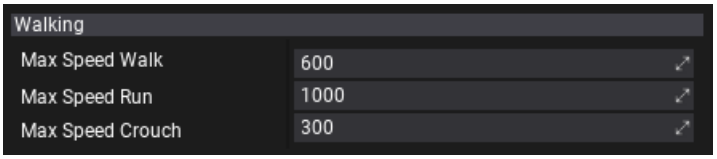


Moving Along

Unlike the default Flax Engine CharacterController, a direct velocity input is not supplied to *CharacterControllerPro*. Instead, a max speed is set, and then an input script supplies the *CharacterControllerPro* script a direction to move along, such as forward. The input script can also supply a scale for the input, to modify the magnitude of the direction. For example, setting the scale to 0.5 would make the player move at half speed and setting it to -1 would cause the player to move backwards.



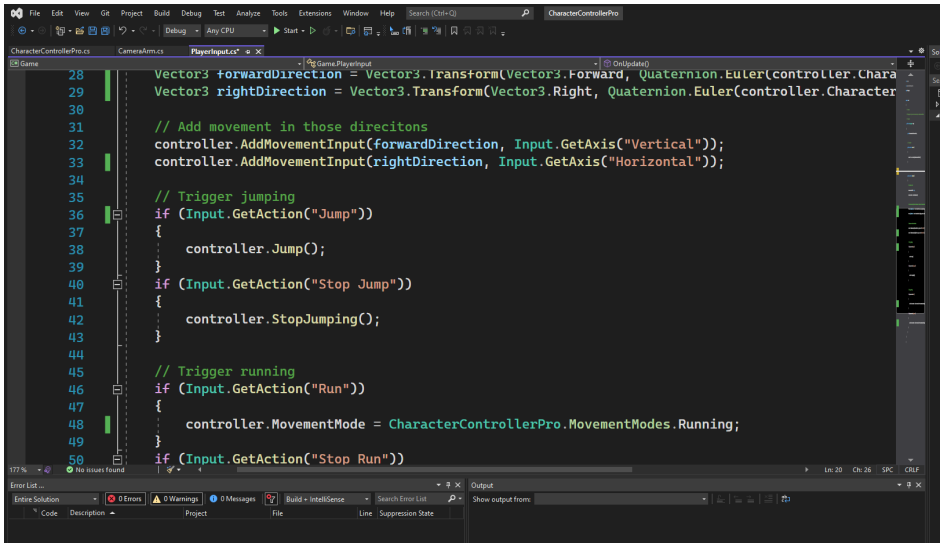
The movement speed is based on the character's MovementMode. There are 4 MovementModes, each with a different speed. By default, the MovementMode is set to the Walking mode. The only MovementMode that cannot be changed is the Stopped mode, which prevents the character from moving. The rest have speed values that can be modified.



Using MovementMode and the input scale, the *CharacterControllerPro* can determine the speed it should move at. For example, if the code in the example above is used, MaxWalkSpeed is set to 600, and the player holds the joystick halfway back, the character would move at a speed of -300 backwards. Similarly, if the player is pressing the W key, the character would move forward at a speed of 600.

The character's velocity cannot be set directly. Instead, the LaunchCharacter method can be used in cases where the character needs a single jolt of speed, such as being shot from a cannon or performing a dash. Similarly, in cases where the player should stop moving without having to worry about friction, deceleration, or input, the StopMovementImmdiately method can be used. MovementMode can also be set to MovementModes.Stopped in addition to this method to stop the player and prevent them from moving. This is useful in many situations, such as cutscenes or dialogue.

Giving Input



```
28      Vector3 forwardDirection = Vector3.Transform(Vector3.Forward, Quaternion.Euler(controller.Chara
29      Vector3 rightDirection = Vector3.Transform(Vector3.Right, Quaternion.Euler(controller.Character
30
31      // Add movement in those direcitons
32      controller.AddMovementInput(forwardDirection, Input.GetAxis("Vertical"));
33      controller.AddMovementInput(rightDirection, Input.GetAxis("Horizontal"));
34
35      // Trigger jumping
36      if (Input.GetAction("Jump"))
37      {
38          controller.Jump();
39      }
40      if (Input.GetAction("Stop Jump"))
41      {
42          controller.StopJumping();
43      }
44
45      // Trigger running
46      if (Input.GetAction("Run"))
47      {
48          controller.MovementMode = CharacterControllerPro.MovementModes.Running;
49      }
50      if (Input.GetAction("Stop Run"))
```

CharacterControllerPro offers an adaptable input system, which allows the developer to use it to make anything from third-person to a side-scroller or a top-down game. This section will explain how to set up input for a third-person game, and hopefully give you the knowledge to confidently use the input system to make any type of game.

```
// Get forward and right direction based on the charcater's CharcaterRotation
Vector3 forwardDirection = Vector3.Transform(Vector3.Forward, Quaternion.Euler(controller.CharacterRotation));
Vector3 rightDirection = Vector3.Transform(Vector3.Right, Quaternion.Euler(controller.CharacterRotation));
```

First, we can get the character's forward and right vectors by using the CharacterRotation variable. We can't use Transform.Forward or Transform.Right, since we never rotate the actual character. So this is where CharacterRotation comes into play.

```
// Add movement in those directions
controller.AddMovementInput(forwardDirection, Input.GetAxis("Vertical"));
controller.AddMovementInput(rightDirection, Input.GetAxis("Horizontal"));
```

Next, we can use those directions to add movement input to the *CharacterControllerPro*. Adding input multiple times is okay, since the values will be normalized. In this case, we are adding input along the forward direction and the right direction we got from the CharacterRotation. We are also using the Vertical and Horizontal Input axes as the scales for the input.

Next, we can call the Jump method when the Jump action is triggered. The values set in JumpForce and GravityForce control the height and duration of the jump. If MaxJumpHoldTime is larger than 0, another action for when the player lets go of the jump key is also required, which should call the StopJumping method. This is done so the player can control the duration of the jump.

```
// Trigger jumping
if (Input.GetAction("Jump"))
{
    controller.Jump();
}
if (Input.GetAction("Stop Jump"))
{
    controller.StopJumping();
}
```

```
// Trigger running
if (Input.GetAction("Run"))
{
    controller.MovementMode = CharacterControllerPro.MovementModes.Running;
}
if (Input.GetAction("Stop Run"))
{
    controller.MovementMode = CharacterControllerPro.MovementModes.Walking;
}
```

Then, we can implement a sprinting system. When the player presses the run, or sprint action, the *CharacterControllerPro*'s MovementMode can be set to MovementModes.Running, which has a higher speed than MovementModes.Walking. When the player releases the run key, another input action, called "Stop Run" should be fired, which should set the MovementMode back to MovementModes.Running.

That's it! You've just implemented a complete third-person input system for *CharacterControllerPro*! The *CharacterControllerPro* project also comes with an input script using the code above, so you can also use that as a reference, or as a base for a more complex input system.

API Reference

CharacterControllerPro contains a robust, yet easy to use API, to give developers the ability to control and customize the feeling and movement style of their character, give input, and more. Most of the movement settings can be tweaked straight from the Properties window, for rapid prototyping and easy customization.

Properties

float Acceleration

Gets or sets the speed that the character will reach its maximum speed. Set to higher value for more snappy movement.

float AirControl

Gets or sets the multiplier for how much control the character will have in the air.

Vector3 CharacterRotaton

Gets a value returning a rotation vector which can be used to determine the direction this character should move in when calculating input.

float Deceleration

Gets or sets how quickly the character will slow down. Set to higher value for more snappy movement.

float Friction

Gets or sets the multiplier for how much control the character will have in the air. Can be used to simulate slippery surfaces.

float GravityForce

Gets or sets how strong gravity will be. Higher values result in stronger gravity.

bool IsJumping

Gets a value indicating if the character is currently in a jump.

bool IsOnGround

Gets a value indicating if the character is on the ground. This exists simply for convenience, and is the same as CharacterController.IsGrounded.

float JumpForce

Gets or sets how powerful jumps will be. Higher values result in stronger jumps.

float MaxJumpHoldTime

Gets or sets how long the character can jump. Non-zero values require calling StopJumping() when the player lets go of the jump button to allow for multi-height jumps.

float MaxSpeedCrouch

Gets or sets the speed the character will move when MovementMode is set to MovementModes.Crouching.

float MaxSpeedRun

Gets or sets the speed the character will move when MovementMode is set to MovementModes.Running.

float MaxSpeedWalk

Gets or sets the speed the character will move when MovementMode is set to MovementModes.Walking.

MovementModes MovementMode

Gets or sets the character's MovementMode, which affects the speed of the character. See MovementModes for possible options.

Vector3 Velocity

Gets the speed of the character, in units per second (u/s). Can be used to detect how fast the character is currently moving. This exists simply for convenience, and is the same as CharacterController.Velocity.

float VisualsRotationSpeed

Gets or sets how quickly the character visuals rotate to match the input direction.

Methods

void AddCharacterRotation(**Vector3** rotation)

Adds a value to CharacterRotation, which can be used to determine the direction of the character.

Vector3 rotation	The value added to the character's CharacterRotation.

void AddMovementInput(**Vector3** direction, **float** scale)

Adds input to the character to be used in movement, such as walking.

Vector3 direction	The world-aligned vector of which the input will be applied. This value should be normalized, and will be if it is not.
float scale	Scale to apply to input. Can be used in cases where there is analog input (e.g. joystick).

void Jump()

Causes the character to jump. If MaxJumpHoldTime is a non-zero value, the character will continue jumping until the jump has lasted as long as MaxJumpHoldTime or when StopJumping() is called.

void LaunchCharacter(**Vector3** newVelocity, **bool** isAdditive)

Effectively launches the character by directly modifying its velocity. It is not recommended to use this function every frame, rather in one-off uses or sequences for custom movement situations, e.g. warping, dahsing.

Vector3 newVelocity	The velocity that will be applied to this character.
bool isAdditive	If true, newVelocity will be added to the character's current velocity. The character's velocity will be set to newVelocity if false.

void StopJumping()

If the character is jumping and MaxJumpHoldTime is larger than 0, calling this will end the jump. This method is useful for cases where holding the jump button will result in a longer jump.

void StopMovementImmediately()

Immediately resets the character's velocity, setting it to 0 on all axes.

Enums

MovementModes

A list of modes that control the motion of the character (e.g. walking, sprinting, standing still)

Stopped	Any movement input supplied to the character will be multiplied by 0, effectively preventing it from moving.
Walking	The character's speed multiplier will be MaxSpeedWalk.
Running	The character's speed multiplier will be MaxSpeedRun.
Crouching	The character's speed multiplier will be MaxSpeedCrouch.