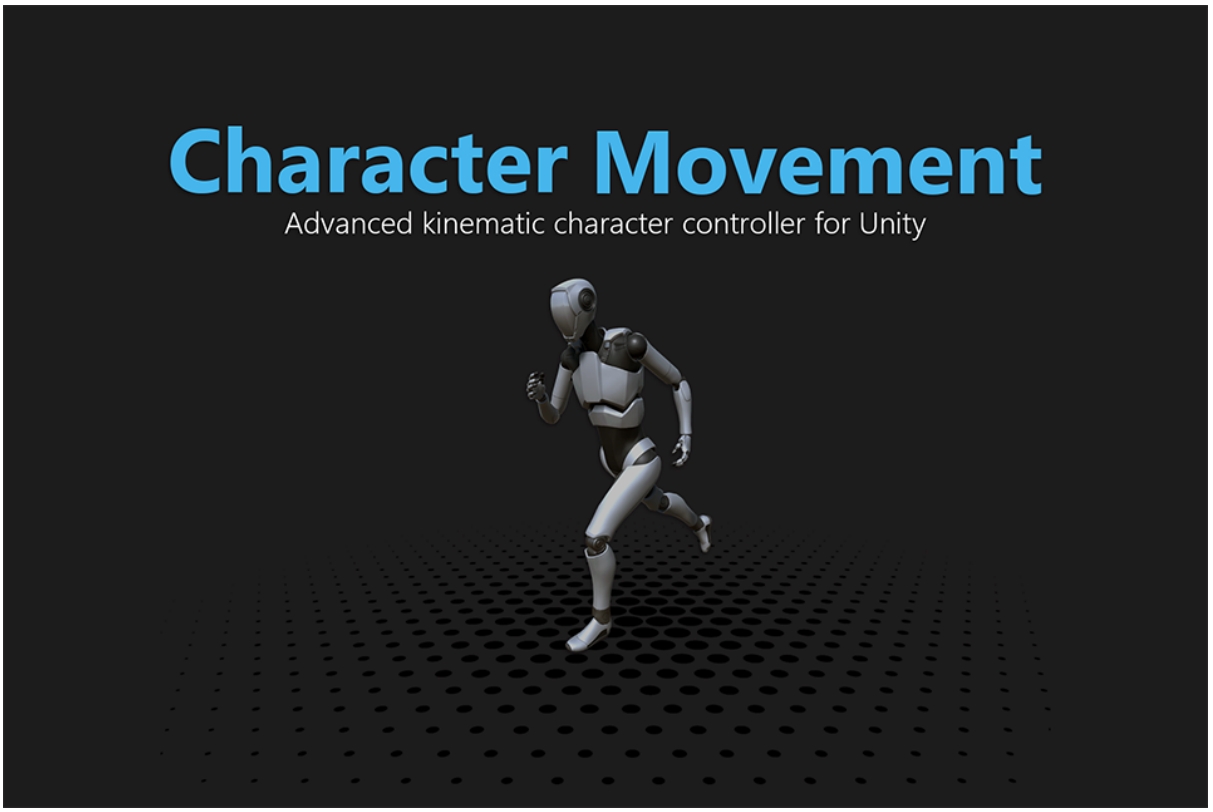
Character Movement

Advanced kinematic character controller for Unity

Getting Started

Version 1.0.1



Important note on package dependencies

The `Character Movement` component does not require any external package in order to operate, however its Demo and the First/Third Person examples require the **Cinemachine** package to work.

Please make sure to install the Cinemachine package into your project when importing the demo and examples.

Creating our Player

1. Create an empty *GameObject* and rename it as **Player**.

2. Drag and drop the **Capsule** *prefab* (*CharacterMovement\Samples\SharedAssets\Prefabs*) into your **Player** *GameObject* so it becomes a child of it. This will act as our character's visual representation.

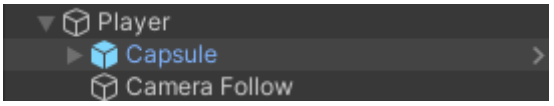


Fig. Player visual representation (our capsule).

3. Add an empty *GameObject* as a child of our **Player** *GameObject*. Call it **Camera Follow** and position it at 0, 1, 0 (capsule's center). This will be our *Camera* follow target.

Fig. Camera's follow target

4. Add the *CharacterMovement* component to our newly created **Player** *GameObject*. It will add its required components for proper operation.

5. Create a C# Script named **PlayerController**.

6.- Add this newly created **PlayerController** *script* to your **Player** *GameObject*. At this point we have a complete armature for our player character.

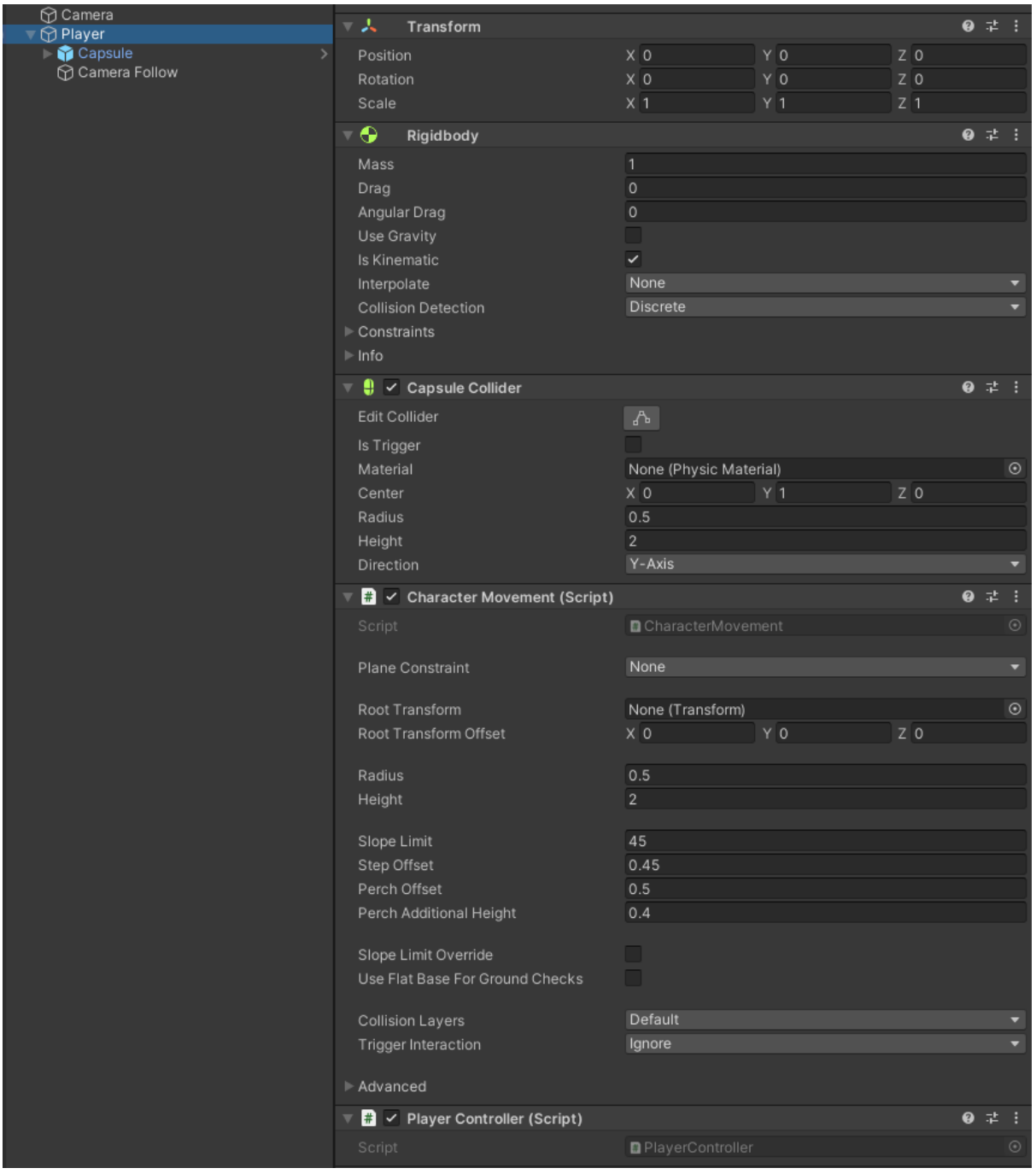


Fig. Player hierarchy and components.

7.- Lastly, drag and drop the *SimpleCameraController* (`CharacterMovement\Samples\Shared Assets\Scripts`) to the scene *Camera* and assign the **Player's Camera Follow** *GameObject* as the *SimpleCameraController Target*, so the camera follows our **Player** around.

Writing the `PlayerController` script

For this example, we'll use the `CharacterMovement SimpleMove` function, which implements a highly configurable friction based movement and will handle all possible movement modes our character will ever need with ease.

1. Add the following editor exposed fields to your `PlayerController` class, this will let us tweak the movement from the editor.

```
public float rotationRate = 540.0f;

public float maxSpeed = 5;

public float acceleration = 20.0f;
public float deceleration = 20.0f;

public float groundFriction = 8.0f;
public float airFriction = 0.5f;

public float jumpImpulse = 6.5f;

[Range(0.0f, 1.0f)]
public float airControl = 0.3f;

public Vector3 gravity = Vector3.down * 9.81f;
```

2. Add the following fields to the `PlayerController` class. This will hold the cached `CharacterMovement` component and the desired input movement direction (in world-space).

```
private CharacterMovement _characterMovement;

private Vector3 _movementDirection;
```

3. Add an `Awake` *MonoBehaviour* method to the **PlayerController** class and add the following code into it. This will cache the `CharacterMovement` component.

```
private void Awake()
{
    _characterMovement = GetComponent<CharacterMovement>();
}
```

4. Add an `Update` *MonoBehaviour* method to the `PlayerController` class and add the following code into it.. Will use this to implement our character movement.

5. In order to be able to move our player around, first we need to know the direction we want to move, for it we read the current *horizontal* and *vertical* input values and convert them into a movement direction vector in *world-space*.

```
// Read Input values

float horizontal = Input.GetAxisRaw($"Horizontal");
float vertical = Input.GetAxisRaw($"Vertical");

// Create a movement direction vector (in world space)

_movementDirection = Vector3.zero;
_movementDirection += Vector3.forward * vertical;
_movementDirection += Vector3.right * horizontal;

// Make Sure it won't move faster diagonally

_movementDirection = Vector3.ClampMagnitude(_movementDirection, 1.0f);
```

We use the *horizontal* and *vertical* axes to create a *horizontal* (*XZ* plane) movement direction vector. Make sure it won't move faster on diagonals clamping its magnitude to 1.

Worth noting the movement is in *world-space*, where the *horizontal* input axis is mapped to the world's *X-Axis* and the *vertical* input axis is mapped to the world's *Z-Axis*. Often it's desired to move the player relative to the camera's view direction (third person, first person, etc). Please refer to the *Walkthrough* guide for this and much more!

6. To implement our jump, we'll make use of the `CharacterMovement` `LaunchCharacter` function. This offers greater flexibility as it lets us add or override the character's velocity component wise.

Since `CharacterMovement` implements a ground-constraint in order to correctly keep the character *'glued'* to the ground, to jump we'll need to explicitly tell the character is allowed to leave the ground.

```
// Jump

if (_characterMovement.isGrounded && Input.GetButton($"Jump"))
{
   _characterMovement.PauseGroundConstraint();
   _characterMovement.LaunchCharacter(Vector3.up * jumpImpulse, true);
}
```

7. While you can freely rotate the character as your game needs, for this particular example we'll use the `CharacterMovement RotateTowards` function.

```
// Rotate towards movement direction

_characterMovement.RotateTowards(_movementDirection, rotationRate * Time.deltaTime);
```

8. Lastly, to move our character, call the `CharacterMovement SimpleMove` function with the **values for its current state** (*e.g:* grounded or not-grounded). This same approach applies for any other state your character needs.

```
// Perform movement

Vector3 desiredVelocity = _movementDirection * maxSpeed;

float actualAcceleration = _characterMovement.isGrounded ? acceleration : acceleration * airControl;
float actualDeceleration = _characterMovement.isGrounded ? deceleration : 0.0f;

float actualFriction = _characterMovement.isGrounded ? groundFriction : airFriction;

_characterMovement.SimpleMove(desiredVelocity, maxSpeed, actualAcceleration, actualDeceleration,
   actualFriction, actualFriction, gravity);
```

9. Done!