Kinematic Character Controller

Walkthrough

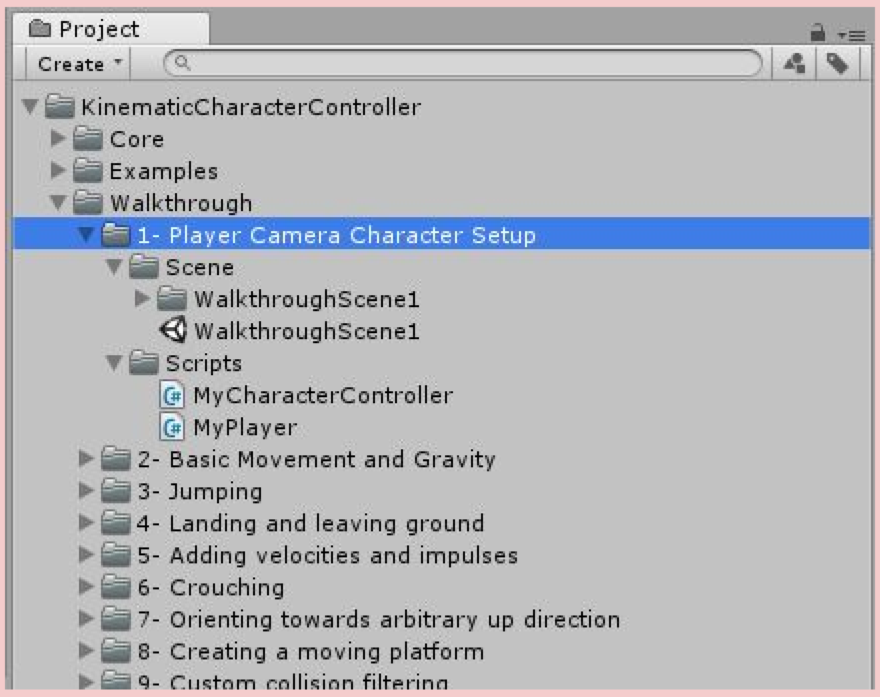
Support email: **store.pstamand@gmail.com**

Table of Contents

Character Controller Creation Walkthrough

This walkthrough will present a step-by-step guide to implementing a complete character controller from scratch using the Kinematic Character Controller system. It can either be followed sequentially or be used as a reference for a specific feature you wish to implement.

How to use this walkthrough:
To follow along, open the “Walkthrough” folder in the project. In each section of this walkthrough, follow along with the scene and the code in the corresponding folder. **The scene in each folder represents the completed version of each section of the walkthrough**.



```
KinematicCharacterController
  Core
  Examples
  Walkthrough
    1- Player Camera Character Setup
      Scene
        WalkthroughScene1
        WalkthroughScene1
      Scripts
        MyCharacterController
        MyPlayer
    2- Basic Movement and Gravity
    3- Jumping
    4- Landing and leaving ground
    5- Adding velocities and impulses
    6- Crouching
    7- Orienting towards arbitrary up direction
    8- Creating a moving platform
    9- Custom collision filtering
```

At every major step of the tutorial, you will be invited to look at specific areas of the walkthrough section code with a highlighted comment such as this one. The comments in the code will explain the rest

Note: The character controller we will be creating during this walkthrough isn’t necessarily meant to be a final game-ready character. It is mostly created for learning purposes. For instance, the swimming mode and the ladder-climbing implementations are very rudimentary and they are mostly just meant to give you a general idea of how such things could be implemented.

Player, Character & Camera setup

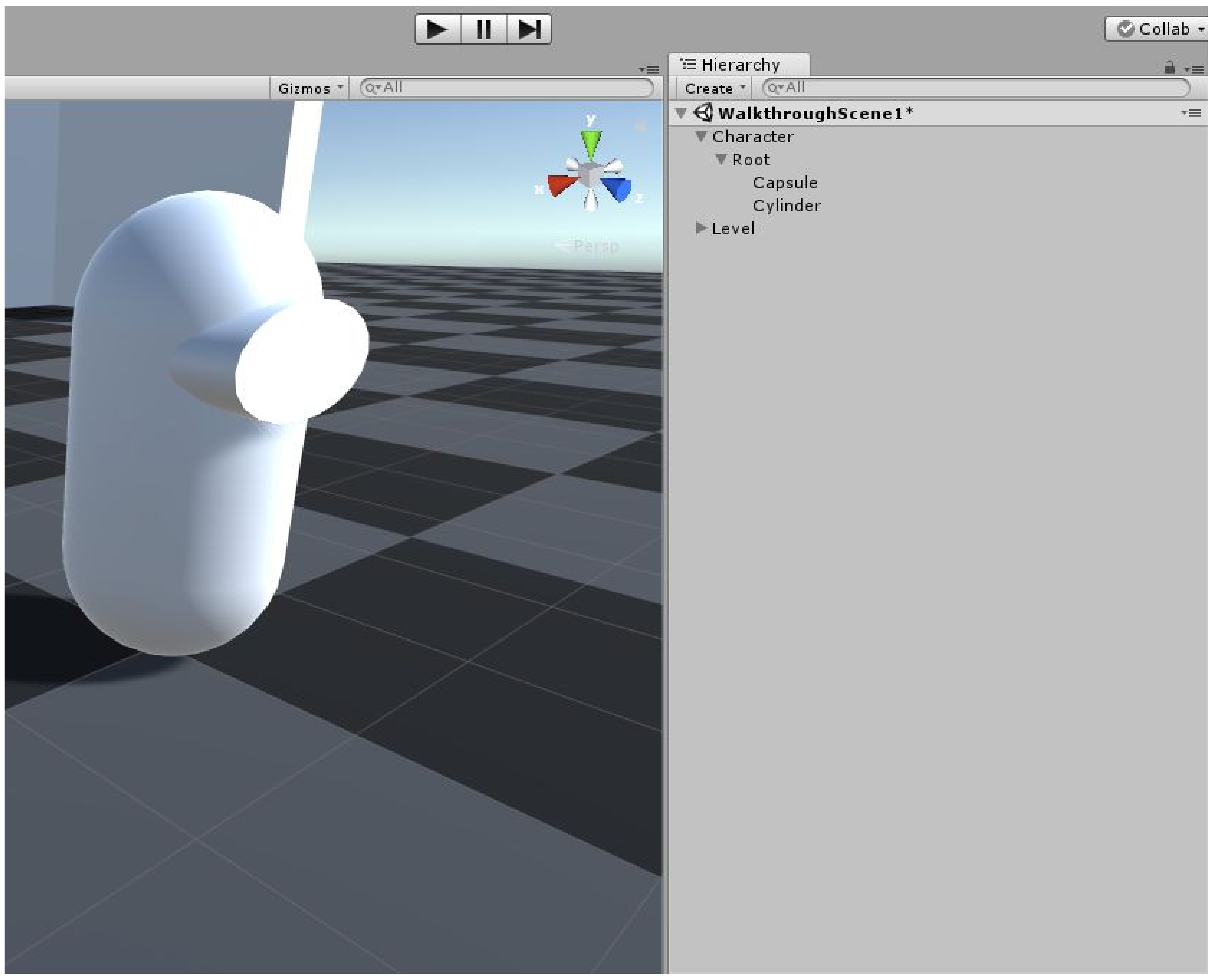
We'll start by creating the general structure for making inputs, characters and cameras work together. Instead of putting input handling and camera control directly into the character controller, we will isolate it in a "MyPlayer" class. The reasoning behind this is that not all character controllers will necessarily be human-controlled. Some will be AI, and it wouldn't make sense for AI characters to handle input and cameras. With this sort of structure, we can easily make certain characters controlled by a human "Player", and others controller by AIs. All of this without requiring two different character controllers.

Creating the Character Controller GameObject

- In a scene, create an empty GameObject
- Add a KinematicCharacterMotor component (this will automatically create a readonly capsule collider and rigidbody)
- Create a script and call it "MyCharacterController". Make it inherit from "BaseCharacterController" and implement all of its abstract methods.
- Add your "MyCharacterController" to the character GameObject
- Assign your "MyCharacterController" component to the "CharacterController" field of the KinematicCharacterMotor component
- Add an empty GameObject as child of your character GameObject, and call it "Root". This will be the container for all the meshes of the character. Having this setup will come in handy later when we will implement crouching.
- Add a mesh as a child of your "Root" GameObject. You can use a capsule primitive for now, **but don't forget to remove ALL colliders that are under the Character GameObject! Otherwise your character will fly off into the sunset because it'll keep trying to de-collide from them!**
- Set your character's capsule dimensions and physics material (can remain empty) under the "Component Settings" section of the KinematicCharacterMotor inspector.
- In the "Motor Parameters" section, you can leave the defaults for now. See the tooltips for more information on what each variable does.

Look at the MyCharacterController class to see what it should look like at this point.

And your Character GameObject should look like this:



```
▼ WalkthroughScene1*
  ▼ Character
    ▼ Root
        Capsule
        Cylinder
  ▶ Level
```

Creating the Player

- In a scene, create an empty GameObject and call it "Player"
- Create a script and call it "MyPlayer".
- Add the "MyPlayer" component to your "Player" GameObject

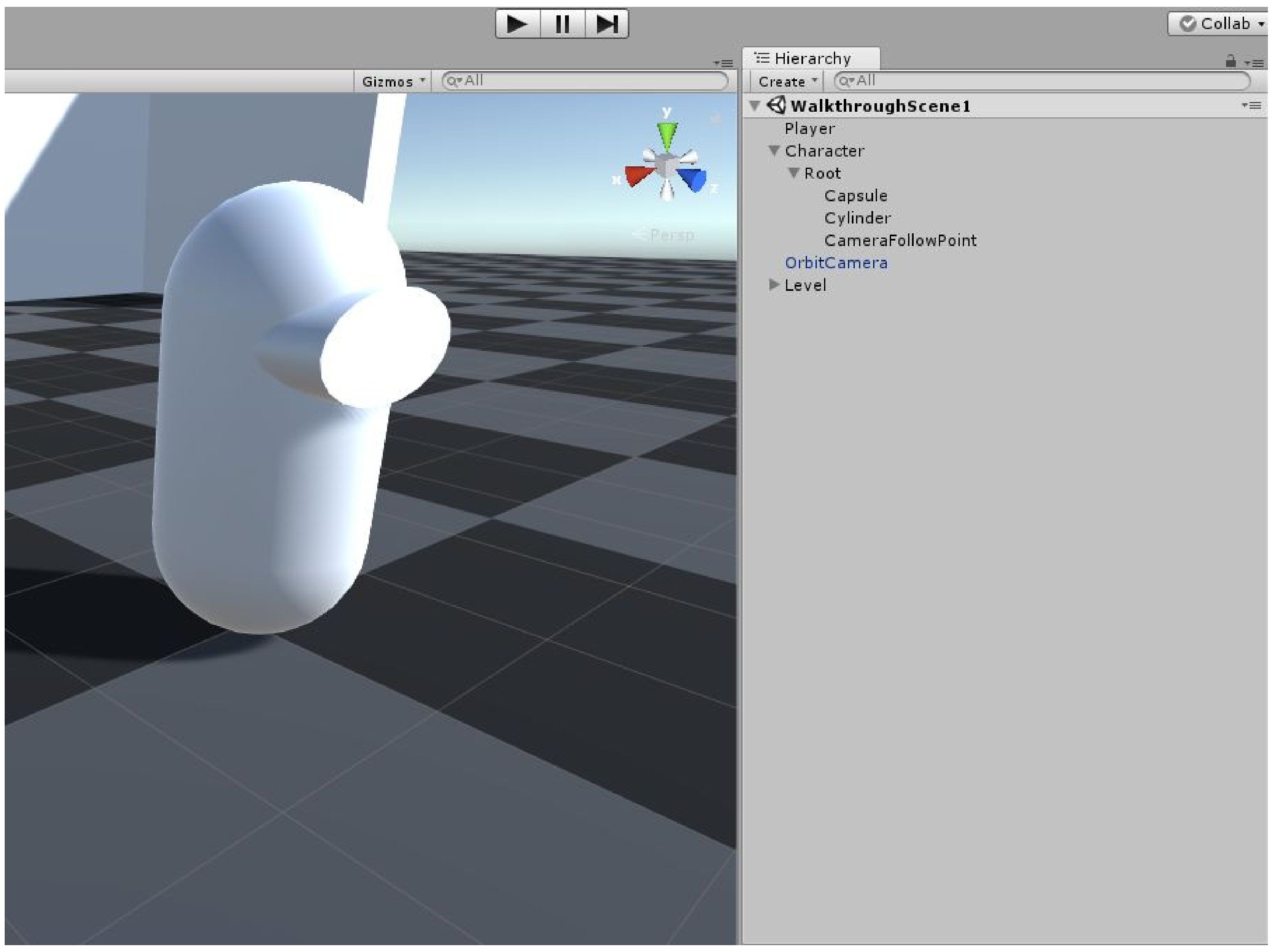
MyPlayer will be where we handle all character and camera input, and make the connections between the camera and the character. For now, let’s just take care of camera handling. On Start(), MyPlayer will setup the camera to follow the character, and on Update(), it will calculate input from mouse movement and mouse scroll wheel, and send these inputs to its assigned OrbitCamera. This will result in a controllable camera that follows the character at all times.

Look at the MyPlayer class, and see how we handle sending input to the camera.

Additionally, look at the OrbitCamera class to see how inputs are transformed into movement. Camera handling will not be the focus of this walkthrough, so we will not go into details about this. (the comments in the code should suffice)

Linking everything together

- Drag the “OrbitCamera” prefab from *KinematicCharacterController\Examples\Prefabs* into the scene
- Assign the OrbitCamera to the corresponding field in MyPlayer’s inspector
- Assign the MyCharacterController to the corresponding field in MyPlayer’s inspector
- Under your Character’s GameObject, add an empty GameObject that will serve as the camera’s orbit point. Add this transform to the “Camera Follow Point” field in MyPlayer’s inspector
- Add a floor to the scene (flat cube primitive with (100, 0, 100) scale, for example)
- The final setup should look like this:



```
WalkthroughScene1
  Player
  Character
    Root
      Capsule
      Cylinder
      CameraFollowPoint
  OrbitCamera
  Level
```

Now you can press play and notice you have control of the camera. The initial setup is complete! But you can’t move the character, because we haven’t yet implemented any movement code….

Basic movement and Gravity

Now let’s start writing some movement code!

Movement input handling in MyPlayer and MyCharacterController

We need to start by making MyPlayer tell MyCharacterController where it needs look and move to.

First of all, we will add a “HandleCharacterInput” method in MyPlayer, which will be called on every Update. This method will evaluate movement input axis, and turn them into a 3D vector oriented relatively to the camera. This means that pressing the W key will create a vector that points along the camera’s forward, and pressing the D key will result in a vector that points along the camera’s right direction. The opposite applies to the S and A keys. This will take care of telling the character where it needs to move to, but we also need to tell it where it needs to orient itself towards. For this, we will use “OrbitCamera.PlanarDirection” which represents the camera’s forward direction projected on the plane represented by its followed transform’s up vector. This means we want the character to look in our camera’s direction at all times, but only on its XZ plane.

For sending those inputs to the character, we will create a “SetInputs” method in MyCharacterController, which is called by MyPlayer at the end of its “HandleCharacterInput” method. “SetInputs” simply stores the inputs for later.

See how this was implemented in the HandleCharacterInput method in MyPlayer, and the SetInputs method in MyCharacterController.

With this, we have MyPlayer sending movement inputs to MyCharacterController, but we have yet to process those inputs.

Character controller movement code

Now it’s time to make our character controller move with those inputs from the Player.

First of all, we will handle translation movement, which will be done in the “UpdateVelocity” override method of MyCharacterController. **This method is called by KinematicCharacterMotor on every character update and it’s basically where you tell your character controller what its velocity should be right now. It is very important to always handle character velocity in this method, because it is called precisely at the right time in the character update loop in order for everything to work well.** You must modify the “currentVelocity” parameter passed as reference to this method in order to set what the velocity should be.

Look at the UpdateVelocity method in MyCharacterController where we implement basic ground/air movement with gravity. See the comments for a more detailed explanation

Next, we can handle character orientation. We will do this in the “UpdateRotation” override method of MyCharacterController. **This method is called by KinematicCharacterMotor on every character update and it’s basically where you tell your character controller what its rotation should be right now. It is very important to always handle character rotations in this method, because it is called precisely at the right time in the character update loop in order for everything to work well.** Modify the “currentRotation” parameter passed as reference to this method in order to accomplish this.

Look at the UpdateRotation method in MyCharacterController where we implement smoothly orienting towards the camera look direction.

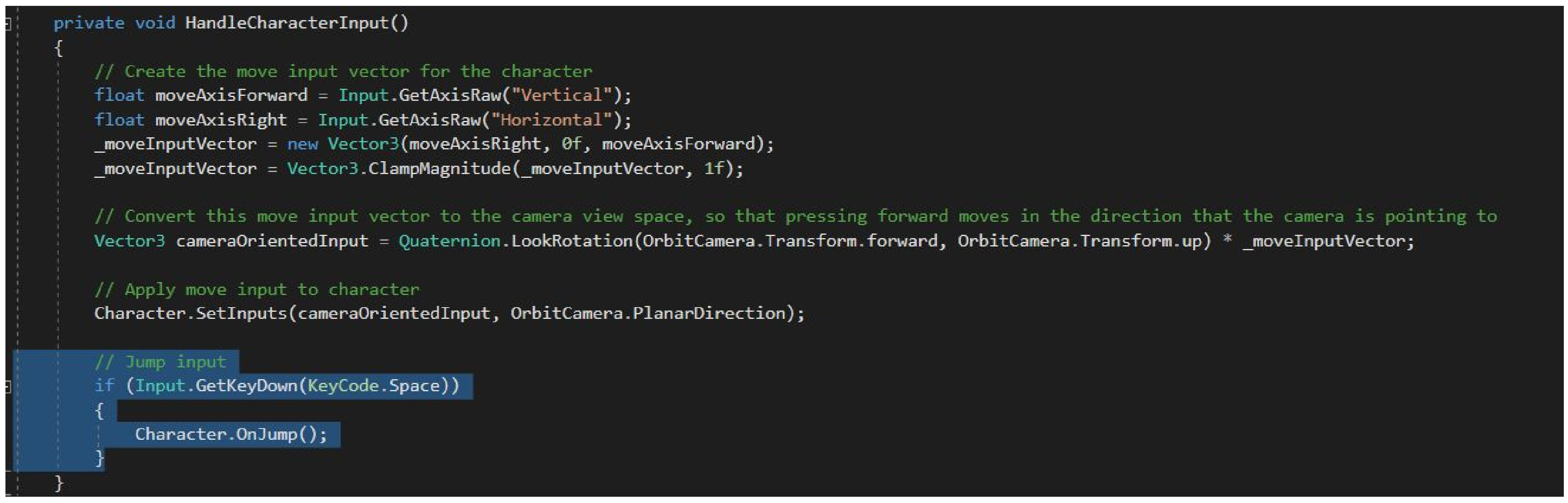
Press play and try out the implemented movement. Feel free to add more geometry to your scene at this point.

Jumping

Simple jumping

To implement jumping, we will first need to handle jump input in MyPlayer.

See the jumping code added to the HandleCharacterInput method



```
private void HandleCharacterInput()
{
    // Create the move input vector for the character
    float moveAxisForward = Input.GetAxisRaw("Vertical");
    float moveAxisRight = Input.GetAxisRaw("Horizontal");
    _moveInputVector = new Vector3(moveAxisRight, 0f, moveAxisForward);
    _moveInputVector = Vector3.ClampMagnitude(_moveInputVector, 1f);

    // Convert this move input vector to the camera view space, so that pressing forward moves in the direction that the camera is pointing to
    Vector3 cameraOrientedInput = Quaternion.LookRotation(OrbitCamera.Transform.forward, OrbitCamera.Transform.up) * _moveInputVector;

    // Apply move input to character
    Character.SetInputs(cameraOrientedInput, OrbitCamera.PlanarDirection);

    // Jump input
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Character.OnJump();
    }
}
```

...and see the corresponding “OnJump” method added to MyCharacterController.

OnJump does not actually apply the jump right away, because we absolutely need to handle all velocity in UpdateVelocity for all movement/collision code to work well. So instead, is simply remembers that we want to jump, and resets the “timeSinceJumpRequested” timer.

The next part of the jump implementation is in the UpdateVelocity and AfterCharacterUpdate methods of MyCharacterController.

See the jump-handling code that adds velocity at the end of the UpdateVelocity method. This is where the jump velocity is actually applied. **Pay special attention to the call to “KinematicCharacterMotor.ForceUnground()”**. This is required whenever we want our character to leave the ground, otherwise it would keep snapping back on.

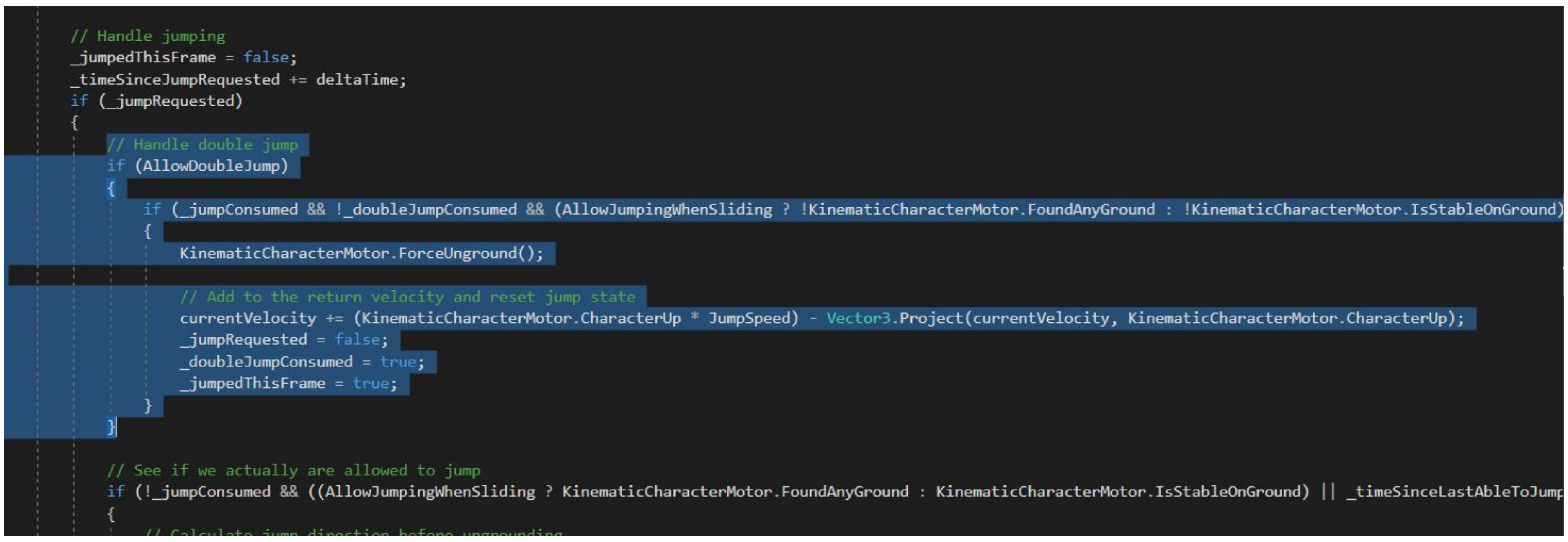
Also see the additional jump logic handling in the “AfterCharacterUpdate” method of MyCharacterController.

Note: The JumpPreGroundingGraceTime and JumpPostGroundingGraceTime respectively represent the extra time before landing where you can press jump and it’ll still jump once you land, and the extra time after leaving stable ground where you’ll still be allowed to jump.

Double Jumping

In order to add double-jumping, we simply have to add a condition when jumping is requested where if we have consumed our first jump and we aren't on ground, we can jump again.

See the implementation of the double-jump in MyCharacterController.UpdateVelocity, under the "// Handle double jump" comment.

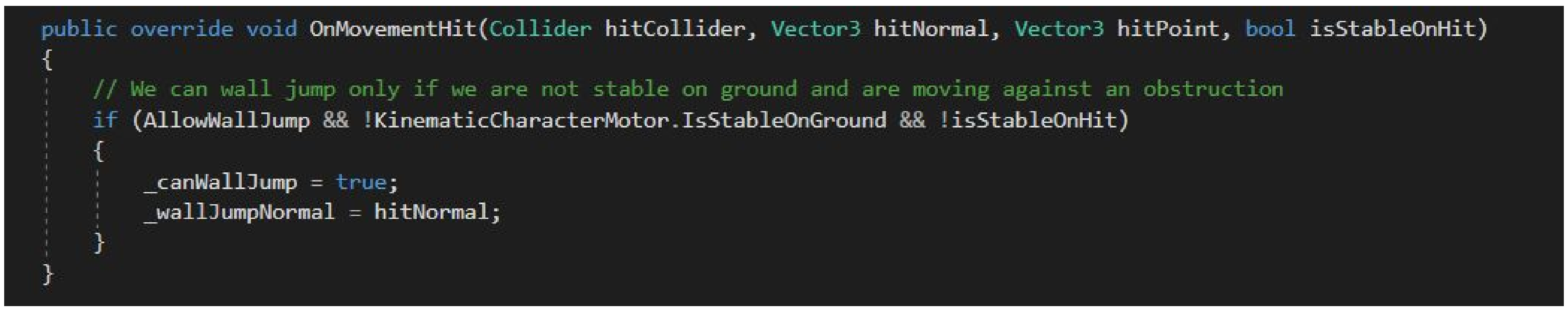


```
// Handle jumping
_jumpedThisFrame = false;
_timeSinceJumpRequested += deltaTime;
if (_jumpRequested)
{
    // Handle double jump
    if (AllowDoubleJump)
    {
        if (_jumpConsumed && !_doubleJumpConsumed && (AllowJumpingWhenSliding ? !KinematicCharacterMotor.FoundAnyGround : !KinematicCharacterMotor.IsStableOnGround)
        {
            KinematicCharacterMotor.ForceUnground();

            // Add to the return velocity and reset jump state
            currentVelocity += (KinematicCharacterMotor.CharacterUp * JumpSpeed) - Vector3.Project(currentVelocity, KinematicCharacterMotor.CharacterUp);
            _jumpRequested = false;
            _doubleJumpConsumed = true;
            _jumpedThisFrame = true;
        }
    }

    // See if we actually are allowed to jump
    if (!_jumpConsumed && ((AllowJumpingWhenSliding ? KinematicCharacterMotor.FoundAnyGround : KinematicCharacterMotor.IsStableOnGround) || _timeSinceLastAbleToJump
    {
        // Calculate jump direction before ungrounding
```

Wall Jumping

In order to implement wall-jumping, we will do something very similar to regular jumping, but only if we are currently moving against a wall. In order to accomplish this, we will need to add code in the "OnMovementHit" method of MyCharacterController that will keep track of if we are allowed to wall-jump, and then we will use that variable in UpdateVelocity in order to perform the actual jump.

See the implementation of the wall-jump in MyCharacterController.OnMovementHit and MyCharacterController.UpdateVelocity. (Look for all the places where the _canWallJump variable is used.)

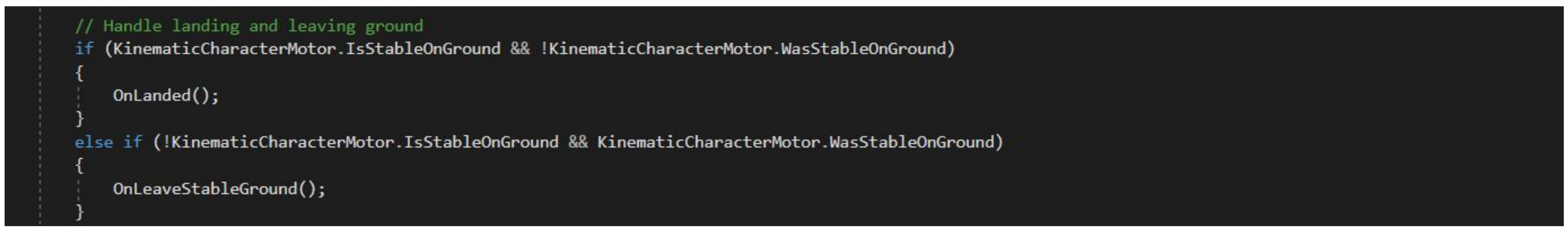


```
public override void OnMovementHit(Collider hitCollider, Vector3 hitNormal, Vector3 hitPoint, bool isStableOnHit)
{
    // We can wall jump only if we are not stable on ground and are moving against an obstruction
    if (AllowWallJump && !KinematicCharacterMotor.IsStableOnGround && !isStableOnHit)
    {
        _canWallJump = true;
        _wallJumpNormal = hitNormal;
    }
}
```

Detecting landing and leaving ground

Most character controllers will need a way to detect when it has landed, or when it has left ground (for animation, sound effects, etc….). This is very easy to accomplish. All we need to do is to compare the “IsStableOnGround” and “WasStableOnGround” of the KinematicCharacterMotor during the character update.

See the implementation of this in MyCharacterController.AfterCharacterUpdate



```
// Handle landing and leaving ground
if (KinematicCharacterMotor.IsStableOnGround && !KinematicCharacterMotor.WasStableOnGround)
{
    OnLanded();
}
else if (!KinematicCharacterMotor.IsStableOnGround && KinematicCharacterMotor.WasStableOnGround)
{
    OnLeaveStableGround();
}
```

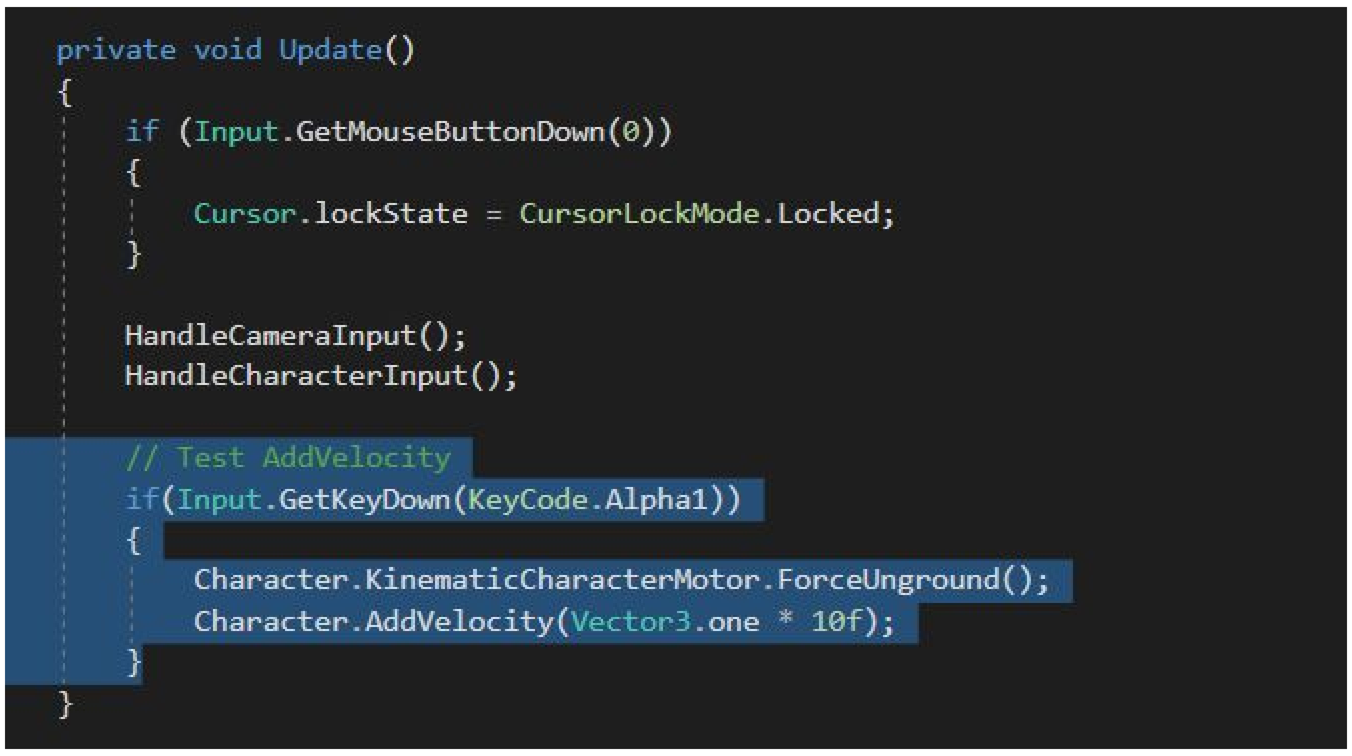
Enter Play mode and try jumping around to see the debug log messages for when you land and leave ground.

Adding velocities and impulses

It is often desirable to have a quick and easy way to add forces and impulses to the character, whether it's for explosion forces, hit impacts, wind zones, etc…. In order to accomplish this, we will create an "AddVelocity" method in MyCharacterController, which will maintain an internal velocity vector to add to the final velocity in UpdateVelocity.

Look for the "AddVelocity" method and all the places where we use _internalVelocityAdd in MyCharacterController

In order to test this, we can simply add some input in MyPlayer that will add a velocity to the character.



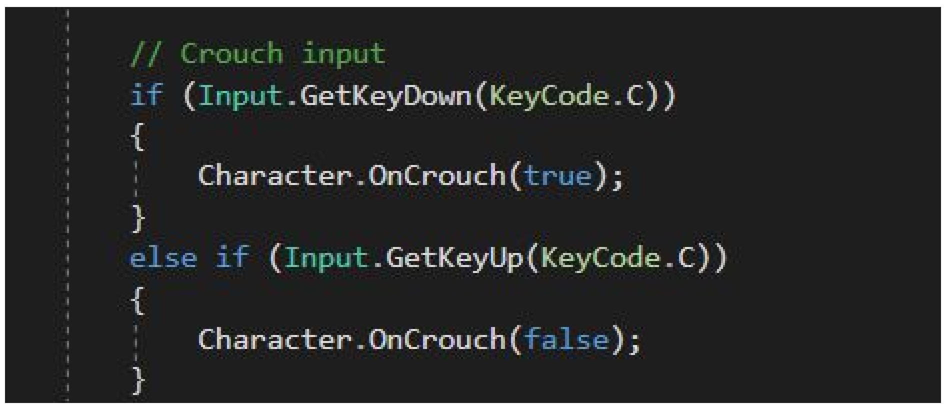
```
private void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Cursor.lockState = CursorLockMode.Locked;
    }

    HandleCameraInput();
    HandleCharacterInput();

    // Test AddVelocity
    if(Input.GetKeyDown(KeyCode.Alpha1))
    {
        Character.KinematicCharacterMotor.ForceUnground();
        Character.AddVelocity(Vector3.one * 10f);
    }
}
```

Notice that we call "ForceUnground" just before adding the velocity. That's because we want the force to launch the character into the air. Without this, the character would always remain snapped to the ground!

Crouching

To implement crouching, we will first need to handle crouch input in MyPlayer.

See the crouching input code added to the HandleCharacterInput method



```
// Crouch input
if (Input.GetKeyDown(KeyCode.C))
{
    Character.OnCrouch(true);
}
else if (Input.GetKeyUp(KeyCode.C))
{
    Character.OnCrouch(false);
}
```

...and see the corresponding “OnCrouch” method added to MyCharacterController.

OnCrouch tracks whether or not we should be crouching right now (based on player input) with the “_shouldBeCrouching” variable. If we want to crouch, we resize the capsule and scale down the character’s meshes to give temporary visual feedback.

But un-crouching is not handled in OnCrouch. That’s because it is possible that the character is in a situation where it doesn’t have enough space to uncrouch. The handling for this is done in MyCharacterController.AfterCharacterUpdate

Look for the “// Handle uncrouching” part of MyCharacterController.AfterCharacterUpdate

This code first tries to determine if we should be uncrouching, and if yes, it’ll temporarily resize the capsule to match the character’s standing height, and do an overlap test with KinematicCharacterMotor.CharacterOverlap. The reason why we do this as opposed to a simple OverlapCapsule is that CharacterOverlap takes all of the character’s ignored colliders and specific collision filtering into account. If it detects that we can’t stand, it resets the capsule dimensions to its crouching size. But if we can stand, it resets the scale of the mesh and assigns IsCrouching to false.

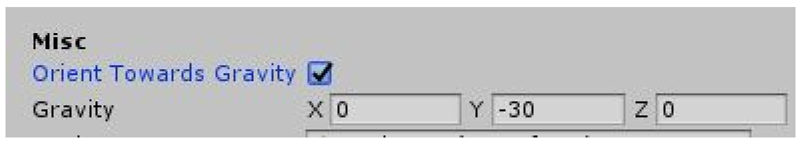
Now try passing under the red module in the scene by crouching with the C key.

Orienting towards arbitrary up direction

In order to demonstrate how you could orient the character towards any direction, we will now implement an option that allows the character to always orient its up direction in the opposite direction of the gravity.

Look for the usage of the “OrientTowardsGravity” variable in MyCharacterController.UpdateRotation

In “UpdateRotation”, which is the method in which you specify what rotation you want your character to have right now, we simply tell the character to rotate from its current up to the inver-gravity direction. This does the trick. Now try to activate “Orient Towards Gravity” in the character’s inspector and play with the character’s “Gravity” vector to see the re-orienting in action.



Misc

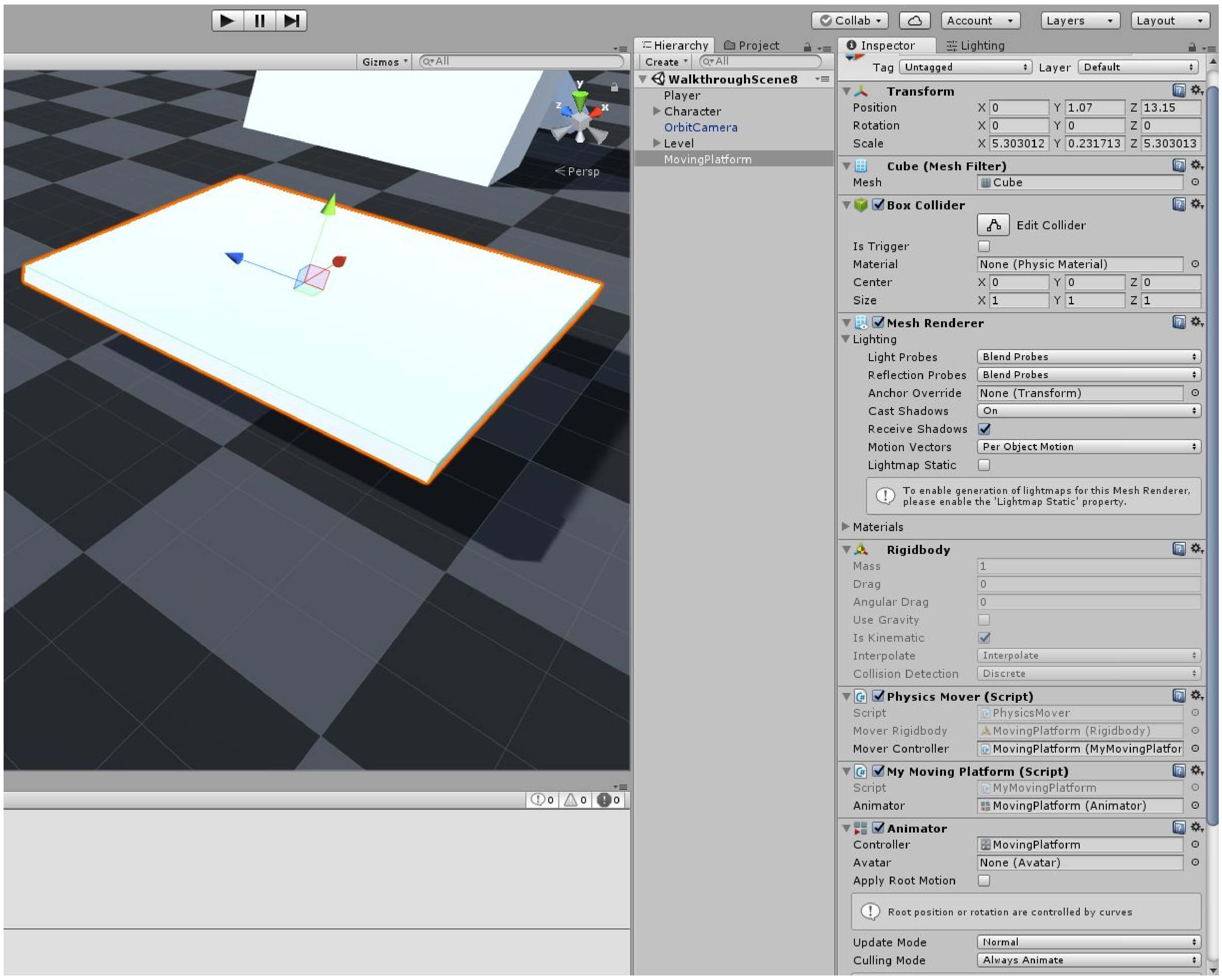
Orient Towards Gravity ✓

Gravity X 0 Y -30 Z 0

Creating a moving platform

We will now create a moving platform that you can move through animations. First, let's set up the platform object:

- Create a flat cube in the scene, and call it "MovingPlatform".
- Add a "PhysicsMover" component to it. You should use this component to create any sort of kinematic moving object that the character can stand on, or be pushed by.
- Create a "MyMovingPlatform" script (or take the one in the walkthrough sources). This class must inherit from "BaseMoverController" and implement the "UpdateMovement" abstract method.
- Add the "MyMovingPlatform" component to the "MovingPlatform" GameObject
- Assign the MyMovingPlatform component to the "MoverController" field of PhysicsMover
- Create an animation that moves and rotate that platform



Tag Untagged Layer Default

Transform

Position	X 0	Y 1.07	Z 13.15
Rotation	X 0	Y 0	Z 0
Scale	X 5.303012	Y 0.231713	Z 5.303013

Cube (Mesh Filter)
Mesh: Cube

Box Collider
Edit Collider
Is Trigger: ☐
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Size: X 1, Y 1, Z 1

Mesh Renderer
Lighting
Light Probes: Blend Probes
Reflection Probes: Blend Probes
Anchor Override: None (Transform)
Cast Shadows: On
Receive Shadows: ✓
Motion Vectors: Per Object Motion
Lightmap Static: ☐
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Materials

Rigidbody
Mass: 1
Drag: 0
Angular Drag: 0
Use Gravity: ☐
Is Kinematic: ✓
Interpolate: Interpolate
Collision Detection: Discrete

Physics Mover (Script)
Script: PhysicsMover
Mover Rigidbody: MovingPlatform (Rigidbody)
Mover Controller: MovingPlatform (MyMovingPlatfor

My Moving Platform (Script)
Script: MyMovingPlatform
Animator: MovingPlatform (Animator)

Animator
Controller: MovingPlatform
Avatar: None (Avatar)
Apply Root Motion: ☐
Root position or rotation are controlled by curves
Update Mode: Normal
Culling Mode: Always Animate

The way to use PhysicsMovers is that you tell it exactly what their position and rotations should be in BaseMoverController's "UpdateMovement" callback. When you handle movement through this callback, all of the physics will be handled properly with the character controllers, so it is crucial not to move those PhysicsMovers with anything else. However, this could be a problem if we want to move them with animation….

We will solve this problem by taking total control over the animation evaluation and only update it along with our PhysicsMover's updates (FixedUpdate). To do this, we will first cache the PlayableGraph of the animator, and call Stop on it. This will make the animation not run automatically.

Look at the stopping of the animator's PlayableGraph in MyMovingPlatform.Start

Next we will implement the UpdateMovement callback.

- First we will cache the pose we were in before evaluating the animation,
- then we will evaluate the animation (which will update the transform's pose instantly),
- then we will tell our PhysicsMover where it needs to go over the next FixedUpdate,

- And finally we will reset the transform’s pose to where it was before evaluating the animation. This is so that rigidbody interpolation can work properly, and so that the animation doesn’t interfere with the physics

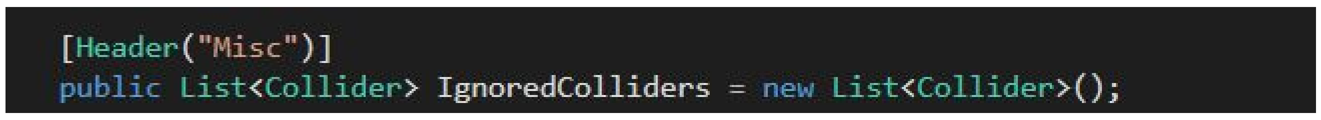
Look at the implementation of MyMovingPlatform.UpdateMovement

With this, we have moving platforms that support animations. Try entering Play mode and jumping on the platform.

Custom collision filtering

In games, you will often need to filter out specific collisions, and it can't always be done elegantly through physics layers. For example, you might want to pass through teammates in an online game, but be blocked by enemies. Here we will create a simple collision filtering example that will give you an idea of how to accomplish this.

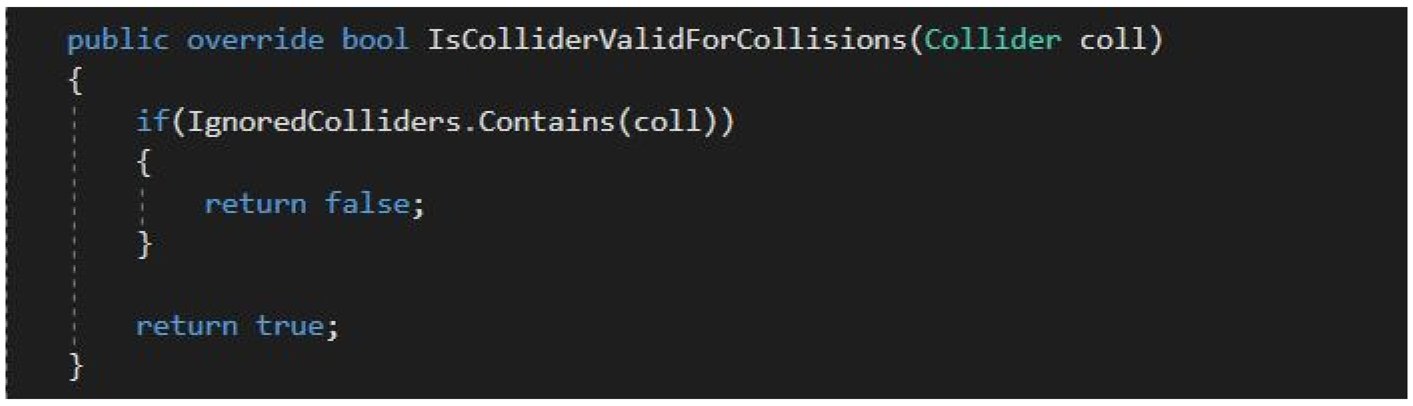
First of all, add a public list of colliders to your MyCharacterController that will represent the colliders you wish to ignore.:



```
[Header("Misc")]
public List<Collider> IgnoredColliders = new List<Collider>();
```

The actual filtering will be done in "IsColliderValidForCollisions". The method asks you to return true if you can collide with this collider, or false otherwise.

Look at the implementation of MyCharacterController.IsColliderValidForCollisions



```
public override bool IsColliderValidForCollisions(Collider coll)
{
    if(IgnoredColliders.Contains(coll))
    {
        return false;
    }

    return true;
}
```

You can try this in play mode by moving against the transparent red cube, which has a collider, but is in our IgnoredCollider list.

You could use this method in any way you want. For example, you could do a GetComponentInParent of any kind of component type on the collider, and filter out all objects that have this specific component.

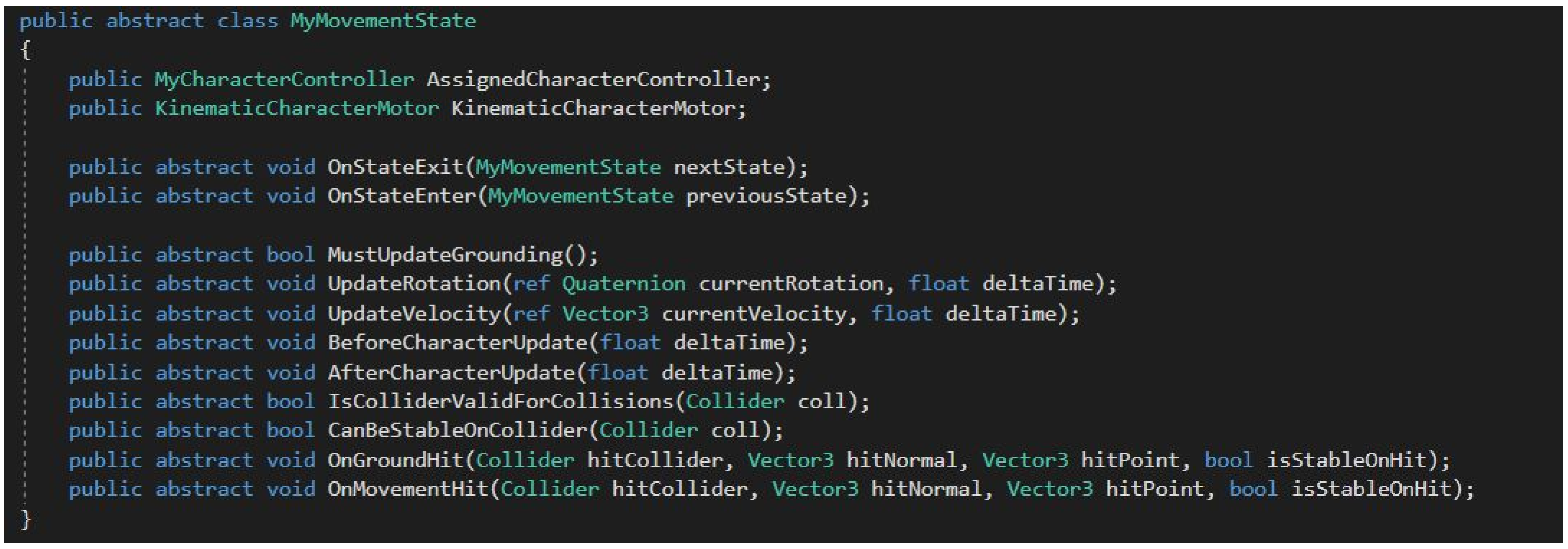
(Note that this does not make the camera ignore those colliders too. The setup you choose to make both the character and the camera ignore the same colliders in your game will be up to you. But as an example, you could put the IgnoredColliders list in MyPlayer, and the make MyPlayer tell both the character and the camera to ignore these.)

Multiple movement states setup

We will soon reach a point where we'll want multiple movement states for our character controller. To prevent the code from getting too messy, we will take a bit of time right now to re-arrange everything into something much easier to work with. It'll pay off later!

Here's what we're going to do:

1. We will create a "MyMovementState" abstract class that has the same callbacks as our MyCharacterController, and some more callbacks for entering or exiting the state:

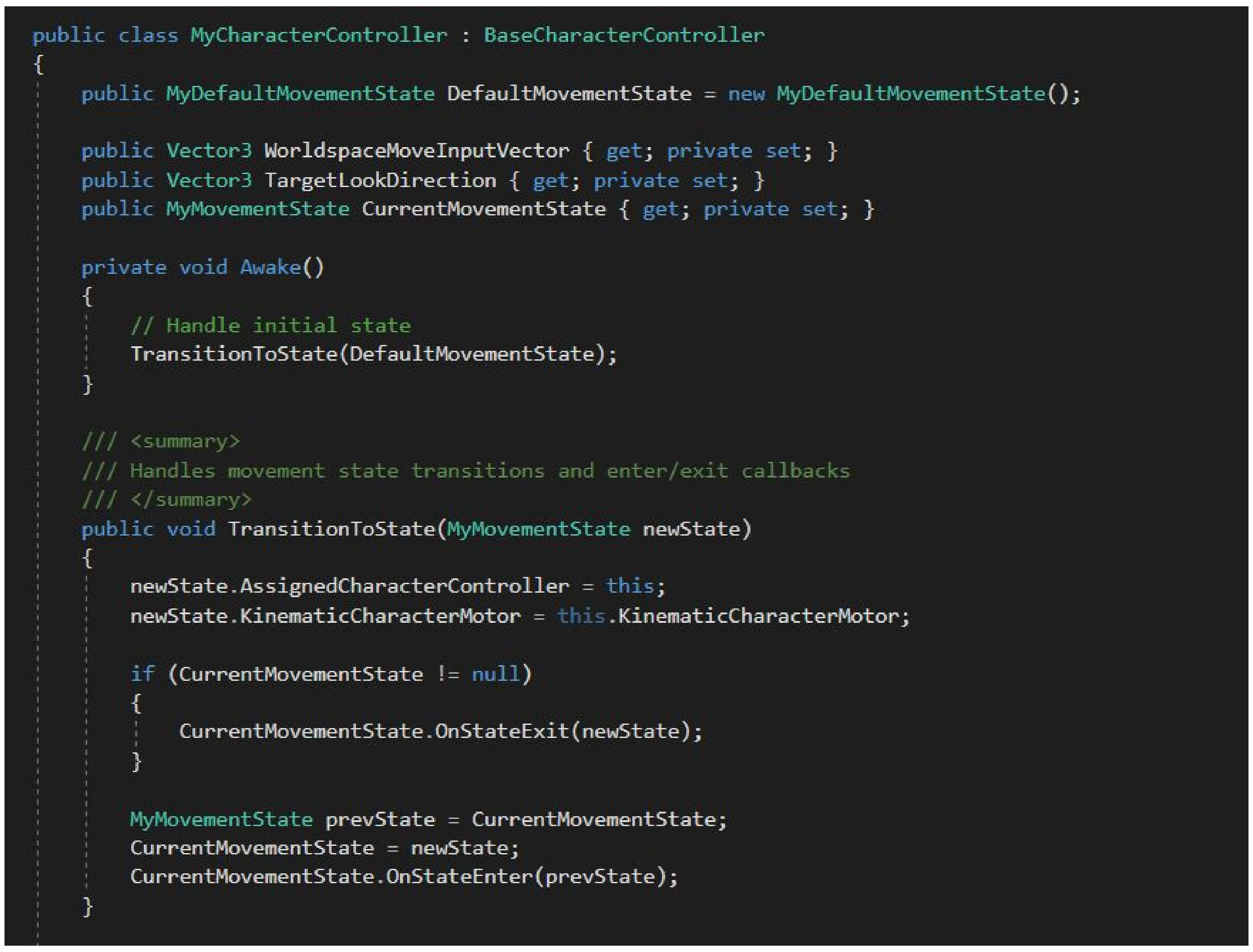


```
public abstract class MyMovementState
{
    public MyCharacterController AssignedCharacterController;
    public KinematicCharacterMotor KinematicCharacterMotor;

    public abstract void OnStateExit(MyMovementState nextState);
    public abstract void OnStateEnter(MyMovementState previousState);

    public abstract bool MustUpdateGrounding();
    public abstract void UpdateRotation(ref Quaternion currentRotation, float deltaTime);
    public abstract void UpdateVelocity(ref Vector3 currentVelocity, float deltaTime);
    public abstract void BeforeCharacterUpdate(float deltaTime);
    public abstract void AfterCharacterUpdate(float deltaTime);
    public abstract bool IsColliderValidForCollisions(Collider coll);
    public abstract bool CanBeStableOnCollider(Collider coll);
    public abstract void OnGroundHit(Collider hitCollider, Vector3 hitNormal, Vector3 hitPoint, bool isStableOnHit);
    public abstract void OnMovementHit(Collider hitCollider, Vector3 hitNormal, Vector3 hitPoint, bool isStableOnHit);
}
```

2. We will then create a child class of MyMovementState, call it "MyDefaultMovementState", and implement all abstract methods. The goal of MyDefaultMovementState will be to represent the default locomotion state of the character.
3. We will move the contents of all character controller methods from "MyCharacterController" to their "MyDefaultMovementState" equivalent

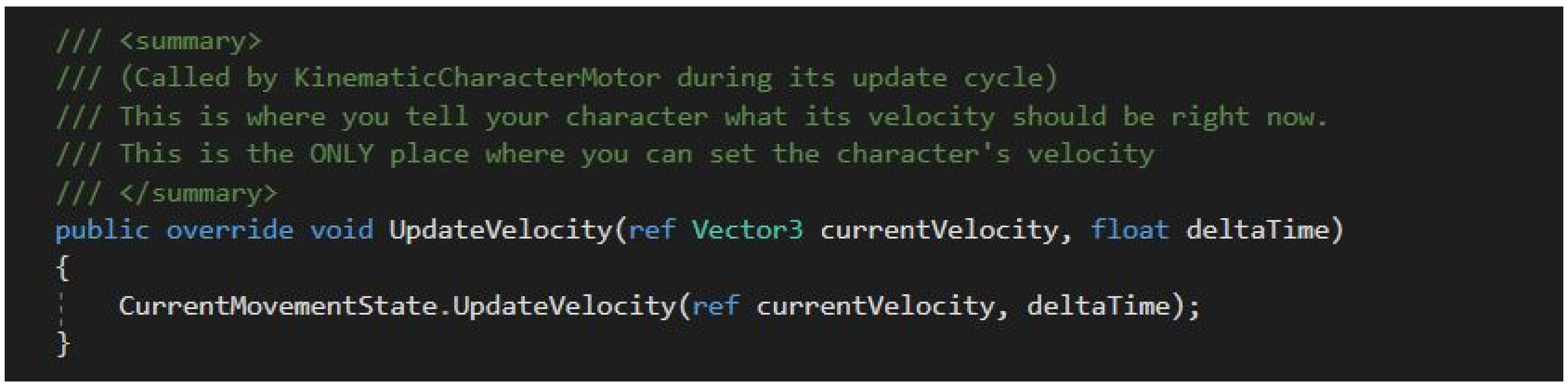
Look at the implementation of MyDefaultMovementState. Notice it contains almost everything that was in MyCharacterController previously, except SetInputs()

4. Back in MyCharacterController, we will implement a few things to create a rudimentary state machine: a "CurrentMovementState", a "TransitionToState" method, a definition for the DefaultMovementState, and a call to "TransitionToState" on Awake in order to initialize the state machine.



```
public class MyCharacterController : BaseCharacterController
{
    public MyDefaultMovementState DefaultMovementState = new MyDefaultMovementState();

    public Vector3 WorldspaceMoveInputVector { get; private set; }
    public Vector3 TargetLookDirection { get; private set; }
    public MyMovementState CurrentMovementState { get; private set; }

    private void Awake()
    {
        // Handle initial state
        TransitionToState(DefaultMovementState);
    }

    /// <summary>
    /// Handles movement state transitions and enter/exit callbacks
    /// </summary>
    public void TransitionToState(MyMovementState newState)
    {
        newState.AssignedCharacterController = this;
        newState.KinematicCharacterMotor = this.KinematicCharacterMotor;

        if (CurrentMovementState != null)
        {
            CurrentMovementState.OnStateExit(newState);
        }

        MyMovementState prevState = CurrentMovementState;
        CurrentMovementState = newState;
        CurrentMovementState.OnStateEnter(prevState);
    }
```

5. For each now-empty character controller method in MyCharacterController (such as UpdateVelocity, OnMovementHit, etc...), we will simply call the equivalent method in the current movement state. Like this:



```
/// <summary>
/// (Called by KinematicCharacterMotor during its update cycle)
/// This is where you tell your character what its velocity should be right now.
/// This is the ONLY place where you can set the character's velocity
/// </summary>
public override void UpdateVelocity(ref Vector3 currentVelocity, float deltaTime)
{
    CurrentMovementState.UpdateVelocity(ref currentVelocity, deltaTime);
}
```

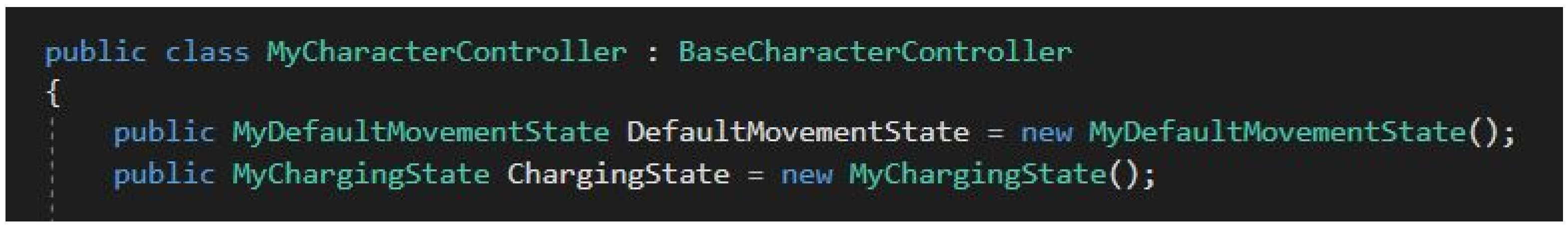
Take some time to look at all the scripts for this section in order to really understand how all of this works together.

We only have one movement state for now, but with this setup, we are ready to add as many movement states as we want!

Charging state

We will now start writing a new and simple movement state that we will call “Charging”. In this mode, the character will keep moving forward at a constant velocity until it hits a wall, or until X seconds have elapsed. It will then pause for some time and go back to the default movement mode.

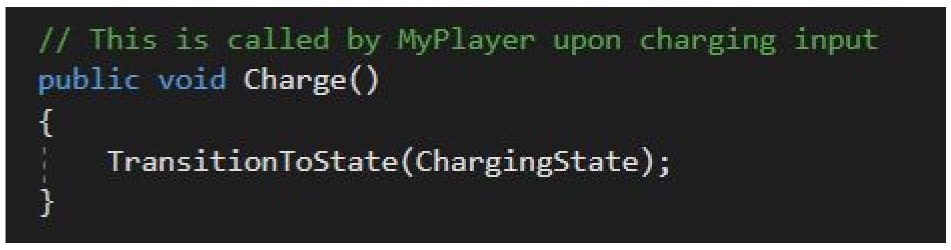
First, we will create a “MyChargingState” class that inherits from MyMovementState and implements all of its abstract methods. Then, in MyCharacterController, we will add a public field for this state:



```
public class MyCharacterController : BaseCharacterController
{
    public MyDefaultMovementState DefaultMovementState = new MyDefaultMovementState();
    public MyChargingState ChargingState = new MyChargingState();
```

Take a look at the MyChargingState class and the corresponding variable added to MyCharacterController.

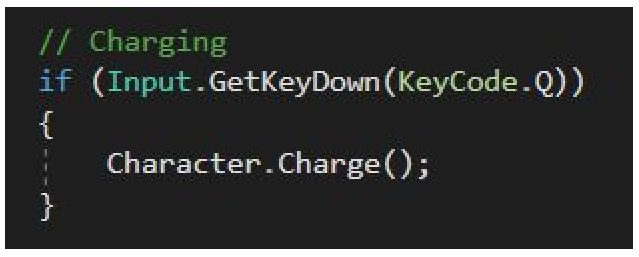
Next, we will create a “Charge” method in MyCharacterController which will transition to the charging state when called:



```
// This is called by MyPlayer upon charging input
public void Charge()
{
    TransitionToState(ChargingState);
}
```

Take a look at the MyCharacterController.Charge method.

Next, we will write the necessary code for switching to that charging state with the “Q” key in MyPlayer.HandleCharacterInput:



```
// Charging
if (Input.GetKeyDown(KeyCode.Q))
{
    Character.Charge();
}
```

Take a look at the charging input handling in the MyPlayer.HandleCharacterInput method.

We are now ready to implement the charging state itself. Here’s an overview of what we want to do:

- In OnStateEnter, cache a charging velocity based on the character’s forward direction
- In UpdateVelocity, keep applying the charging velocity every update unless we are in the pause at the end of the charge
- In OnMovementHit, look for collisions with walls in order to trigger the end of the charge
- In AfterCharacterUpdate,
    - determine if the charging time has elapsed and if we need to end the charge
    - determine if it’s time to retransition back to the default movement state

See how each of these points were implemented in MyChargingState

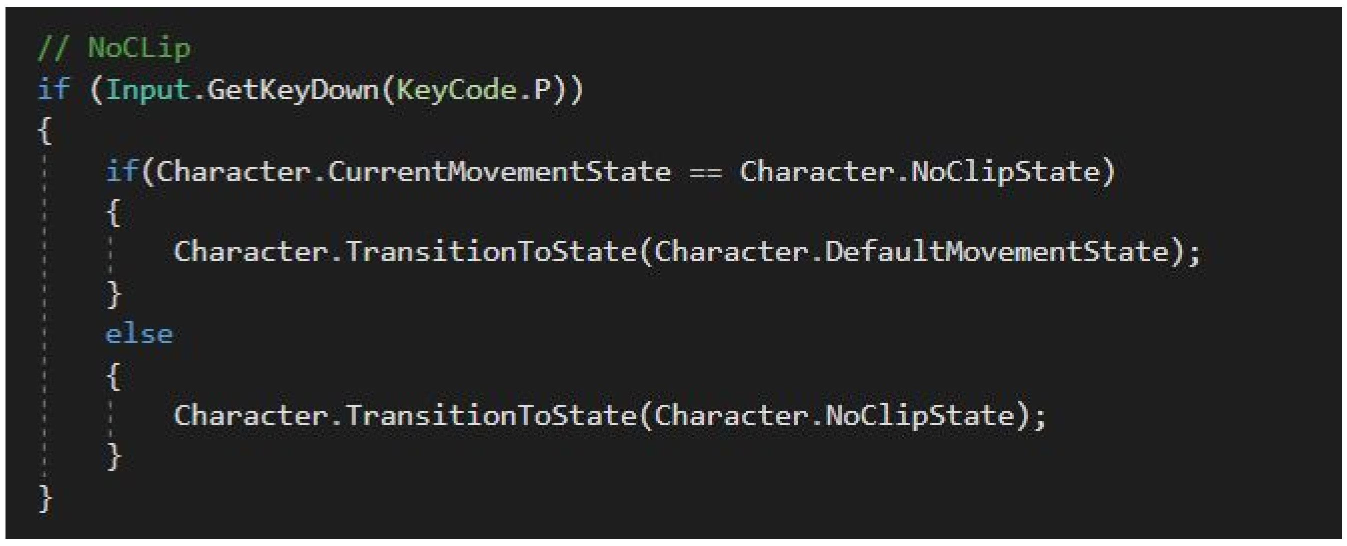
In Play mode, we can now press Q and our character will enter a charge that can be stopped by walls or by reaching a certain time. Notice that it does not get interrupted by ramps or steps, and that gravity is only applied when stopped. This is just to show you the kind of versatility you can have with this system.

NoClip state

We will now demonstrate a simple implementation of a “NoClip” mode in order to teach you about the “HandlePhysics” method of KinematicCharacterMotor. A NoClip mode is when your character can fly around and pass through all collisions. It’s like a “spectator” mode of some sort.

To do this, follow the same procedure as in last section in order to add a new movement state to MyCharacterController. We will call this one “MyNoClipState”.

Take a look at MyNoClipState, and the input we added to MyPlayer that makes you transition to this state



```
// NoCLip
if (Input.GetKeyDown(KeyCode.P))
{
    if(Character.CurrentMovementState == Character.NoClipState)
    {
        Character.TransitionToState(Character.DefaultMovementState);
    }
    else
    {
        Character.TransitionToState(Character.NoClipState);
    }
}
```

There is very little going on in MyNoClipState. The most important thing is what happens in OnStateEnter and OnStateExit. Here we control whether or not the character’s custom collision detection code will be bypassed or not with the KinematicCharacterMotor.HandlePhysics method. When the first parameter of HandlePhysics is set to false, no collision solving is done whatsoever, and the character can go through anything. The second parameter controls the activation of the kinematic capsule collider itself. Always use HandlePhysics for deactivating character collisions instead of trying to disable the capsule collider directly, otherwise the character controller’s capsule casts and overlap tests will still be running, and it’ll give you unexpected results.

The rest of the implementation of MyNoClipState is just some very basic velocity handling in UpdateVelocity. In order to add a way of moving up and down vertically in NoClip mode, we also added input handling in MyCharacterController to keep track of a “VerticalInput”.

Look at how we now have separated input in three different methods for clarity in MyCharacterController:

- SetMoveVectorInput
- SetLookDirectionInput
- SetVerticalInput

And see how these methods are used in MyPlayer. The VerticalInput is controlled with the SPACE and C keys.

Try entering Play mode and pressing “P” to enter NoClip mode. Move around with W ,S, A, D, SPACE, and C keys

Swimming state

Now we will implement a swimming state. Start by creating a new movement state class just like in the previous sections, and call it “MySwimmingState”.

First we will handle transitions between the default movement state and the swimming state. This will be done in MyCharacterController. We will add a “SwimmingReferencePoint” transform variable to that class, which will represent the point that will trigger the transition to the swimming state when submerged in water. The detection of “being submerged” will be done in the “BeforeCharacterUpdate” callback. We will start with a “CharacterOverlap” test to detect if we are overlapping with any water trigger on the specified water layer. If we’ve found an overlapping trigger, we will use “Physics.ClosestPoint” to determine if our “SwimmingReferencePoint” is inside the trigger collider or not. If it is inside the trigger, we can transition to swimming state.

Look at the implementation of the swimming state transitions in MyCharacterController.BeforeCharacterUpdate

Next, we will implement the swimming state itself. The first thing to look at is the UpdateVelocity method. In it, we first have a smooth velocity interpolation that is very similar to the one we had in the NoClip mode. But after that, we do a test to see if this velocity would take our SwimmingReferencePoint out of the water on the next frame. If so, we find out what the water surface normal would be using Physics.ClosestPoint, and project our velocity on that plane. This will take care of sticking to the surface of the water even if you’re trying to move upwards out of it.

Look at the implementation of this in MySwimmingState.UpdateVelocity

Finally, we need to return false in the “MustUpdateGrounding” and “CanBeStableOnCollider” methods. If you try swimming in Play mode with both methods returning true, you’ll see that the character snaps to the ground underwater, which is not how swimming should work. We don’t want any ground detection running when in swimming mode, so we will return false. As for “CanBeStableOnCollider”, if we left it to true even though “MustUpdateGrounding” was returning false, the velocity would not be projected correctly when swimming against ramps that you’d normally be stable on. This is because if you are in a state where you’re not stable on ground, but the movement algorithm in KinematicCharacterMotor detects that you’re moving against something you could be stable on, it’ll do a special velocity projection that is meant for when your character “lands” on stable ground. We bypass this entirely by telling the motor that we can never be stable on any collider in this mode.

Notice that “MustUpdateGrounding” and “CanBeStableOnCollider” are returning false in MySwimmingState

Now enter Play mode and go for a swim!

Climbing a ladder

Now we will implement the ability to climb ladders. The end result we'll be looking for is as follows:

- We have ladders in the world that have a start point and an end point, which we can define manually. These points will form a "ladder segment" (displayed in cyan color in scene view).
- When pressing a key (E), if the character is in range of a ladder, the character will "snap" to the ladder on the ladder segment. This will be our ladder climbing state.
- When in ladder climbing state, pressing W and S makes you move up and down the ladder segment
- Once you reach one of the extremities of the ladder segment, your character will automatically snap off of the ladder and return to its default movement state.

The ladder script

We will start by implementing the ladder script, which will serve two purposes: defining the ladder segment, and containing a method that calculates the closest point on a segment from another point. The character will later use the aforementioned method to know where it has to move to when it wants to snap to a ladder.

Take a look at the MyLadder script:

- LadderSegmentBottom and LadderSegmentLength are used to define the ladder segment
- BottomReleasePoint and TopReleasePoint are used to tell the character where to move to when it reaches an extremity of the ladder and needs to return to default movement mode
- ClosestPointOnLadderSegment is the method that calculates where the character must move to when it wants to snap to the ladder segment

Additionally, in the scene, you can find the ladder GameObjects and see that they have a trigger on them. We will use those triggers later for detecting ladders.

Implementing state transitions in MyCharacterController

After creating a MyLadderClimbingState and defining it in MyCharacterController, we will implement the transition to that state. We will create an "Interact" method in MyCharacterController, which would be a general-purpose environment interaction method. This method will be called by MyPlayer when the E key is pressed.

Notice the call to "Character.Interact" in MyPlayer

When called, the Interact method will first do an overlap test. If anything was found, and if the overlapped collider had a "MyLadder" component, we pass a reference to that ladder to our LadderClimbingState, and we will transition to that state. Additionally, if we were already in the climbing state, we will go back to our default movement mode upon pressing E.

Look at the implementation of the "Interact" method in MyCharacterController

The ladder climbing state

MyLadderClimbingState itself has three sub-states: Anchoring, DeAnchoring, and Climbing. Anchoring and DeAnchoring are when the character is transitioning in and out of "snapping" to the ladder segment. In this example, this is done through a simple interpolation of the character's position and rotation, but in a real game this would normally be done with specific animations.

Let's take a look at OnStateEnter/OnstateExit first. Here we use KinematicCharacterMotor.HandlePhysics to disable the character's movement and collision solving code, but keep the actual kinematic capsule collider active. This is because we don't want anything potentially making our character de-collide from the ladder or from the walls while it is climbing, but we do want things to collide with it (imagine if a box fell on the character as it was climbing). The rest of OnStateEnter is simply caching the position and rotation that we want to snap to. We use the ClosestPointOnLadderSegment method of MyLadder for this.

Look at the implementation OnStateEnter and OnStateExit in MyLadderClimbingState

Next, let's look at UpdateVelocity. If we are climbing, we will set our velocity to a certain speed along the ladder's up direction, depending on if we press up (W), or down (S). When Anchoring or DeAnchoring, we will set a velocity that has the effect of interpolating our character's position from where it was originally to where it needs to go. We use KinematicCharacterMotor.GetVelocityForMovePosition here to make things easier for ourselves. This method returns the velocity required to move to the target position over the next character update frame.

Look at the implementation UpdateVelocity in MyLadderClimbingState

Next, we will look at UpdateRotation. If we are climbing, we will set our rotation directly to the ladder's. If we are anchoring or de-anchoring, we interpolate our rotation from original to target.

Look at the implementation UpdateRotation in MyLadderClimbingState

Finally, let's look at what we do in AfterCharacterUpdate. If we are climbing, we will keep checking if we have reached one of the extremities of the ladder, so that we can de-anchor from it (and therefore transition to the DeAnchoring state). We do this by using the second parameter of the "ClosestPointOnLadderSegment" method, which always returns 0 if we are within the bounds of the segment, and returns the distance from the closest extremity if we are out of bounds. If we are anchoring, we detect if the anchoring phase is finished so that we can transition to the Climbing state. If we are de-anchoring, we detect if the de-anchoring phase is finished so that we can transition back to the default movement state.

Look at the implementation AfterCharacterUpdate in MyLadderClimbingState

Now enter Play mode and try out some of the ladders in the scene.

Root motion example

Now we will demonstrate how to use animation root motion with this character controller. For the sake of simplicity and clarity, we've made new MyPlayer and MyCharacterController classes specifically for this section, which only handle basic root motion movement.

MyPlayer now only handles two input types for the character: moveAxisForward (W and S keys), and moveAxisRight (A and D keys).

Look at the new input handling in MyPlayer.HandleCharacterInput

MyCharacterController now has only one movement state: MyRootMotionMovementState. But we'll get to that later. First, let's see how we can gather information about root motion. There is an Animator component on the same GameObject as MyCharacterController, which means we can use the "OnAnimatorMove" callback of Monobehaviours. In OnAnimatorMove, we accumulate root motion position/rotation deltas every frame while we wait for the character update to process that motion. We need to do this because since the character update runs on a FixedUpdate, it's entirely possible that we'll get multiple OnAnimatorMove callbacks between two character updates.

Look at the implementation of OnAnimatorMove in MyCharacterController,
And see how they are only reset in AfterCharacterUpdate

The next important thing to notice in MyCharacterController is the handling of animation parameters in Update(). Here we smooth out input values and apply them to the "forward" and "turn" parameters of the Animator, which makes the character run.

Look at the animation handling in MyCharacterController.Update

At this point, we've implemented the requirements for a character that is animated with input and tracks root motion deltas. All we need to do now is to apply that root motion as a velocity. This is handled in MyRootMotionMovementState. In UpdateVelocity, if we are grounded, we calculate the velocity from the root motion position delta, reorient it on the ground slope, and set "currentVelocity" to that. This takes care of moving with root motion. If we're not grounded, the movement handling is the same as in the "MyDefaultMovementState" of previous sections. Note that it is extremely important that we translate the root motion to a velocity and apply it in UpdateVelocity. Otherwise, if we simply let the root motion move the transform directly, the character's movement solving code would not be taken into account.

Look at MyRootMotionMovementState.UpdateVelocity to see how root motion position is applied as a velocity.

Next, we handle root motion rotation in UpdateRotation. Here we simply rotate our current rotation by the root motion rotation delta.

Look at MyRootMotionMovementState.UpdateRotation to see how root motion rotation is applied

Now you can try out root motion movement in Play mode.

Navmesh usage example

Now we will demonstrate how to use navmeshes with this character controller. We will do this by creating an AI bot that follows you around in the level.

Note: Since the assets for this section contain the NavMeshComponents package, which may already be present in many projects, they are all packaged in the “Walkthrough16_Package” file. Import the contents of this unityPackage to follow along.

First, we will create the AI bot character. For this, we will simply duplicate our own character and give it a different material. Note that we will not put any sort of “NavMeshAgent” on this character. Instead, we will create a “MyAI” class that will fulfill a similar role as “MyPlayer”, but will handle navigation for AI characters. We will discuss its implementation later.

Next, we will use Unity’s new navmesh components (https://github.com/Unity-Technologies/NavMeshComponents) to build a navmesh in the example scene. A “NavMeshSurface” component was placed on the “Level” GameObject on the scene, and we baked a navmesh with it. Note that we baked the navmesh for an agent type that corresponds to our AI bot character’s dimensions and slope limit. For more information on using the new navmesh components, see the github page.

Finally, we will implement the MyAI class. MyAI will have a reference to its controlled character, as well as its destination transform (our player character, in this case). On every Update, MyAI will try to calculate a path to its destination, and if it finds one, it’ll set inputs on its controlled character that will make it move towards the next path node.

Look at the implementation of the “HandleCharacterNavigation” method in MyAI

The result is an AI bot that follows our character around in the scene. Try it out in Play mode. Notice how we didn’t have to make two separate character classes for human players and AIs in order to accomplish this. We only had to make two “controllers” that used the same character controller script to send input to. Of course, if you prefer making two separate character controllers, it is also perfectly doable with this system.