

Kinematic Character Controller



Kinematic Character Controller *Walkthrough*

Support email: store.pstamand@gmail.com

Table of Contents

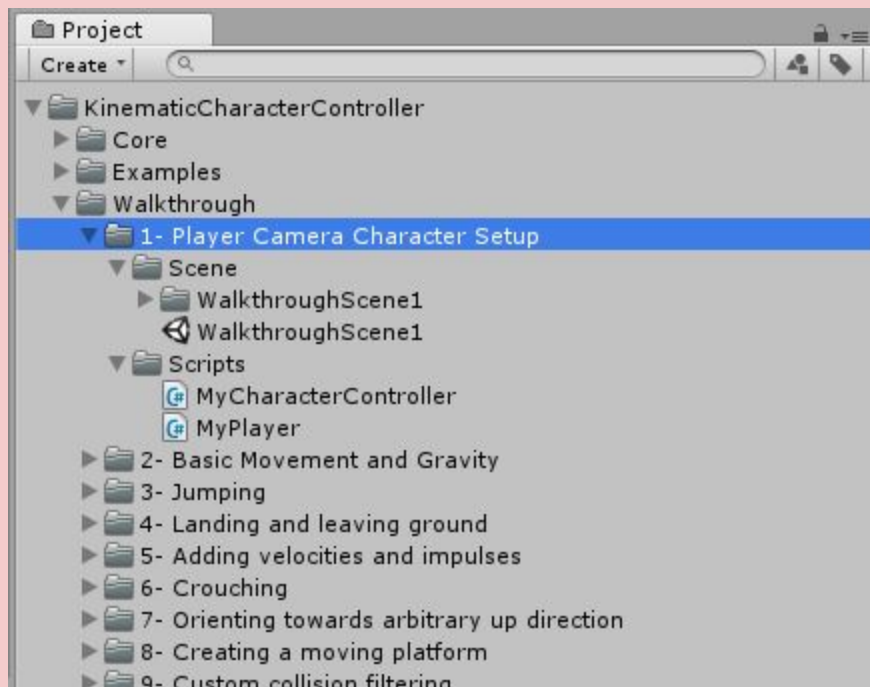
Character Controller Creation Walkthrough	3
Player, Character & Camera setup	4
Creating the Character Controller GameObject	4
Creating the Player	4
Linking everything together	5
Basic movement and Gravity	6
Movement input handling in MyPlayer and MyCharacterController	6
Character controller movement code	6
Jumping	7
Simple jumping	7
Double Jumping	7
Wall Jumping	7
Detecting landing and leaving ground	8
Adding velocities and impulses	9
Crouching	10
Orienting towards arbitrary up direction	11
Creating a moving platform	12
Custom collision filtering	14
Multiple movement states setup	15
Charging state	16
NoClip state	17
Swimming state	18
Climbing a ladder	19
The ladder script	19
Implementing state transitions in MyCharacterController	19
The ladder climbing state	19
Root motion example	21
Navmesh usage example	22
Frame Perfect rotation	23
Networking example	24
Introduction	24
How to try out the example	24
Mandatory reading	24
Overview of the system	25
How to “Tick” the character movement simulation	25
How to save the character state	25
How to restore a character state	26
What has to be networked?	26
How interpolation is handled	26

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.**



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

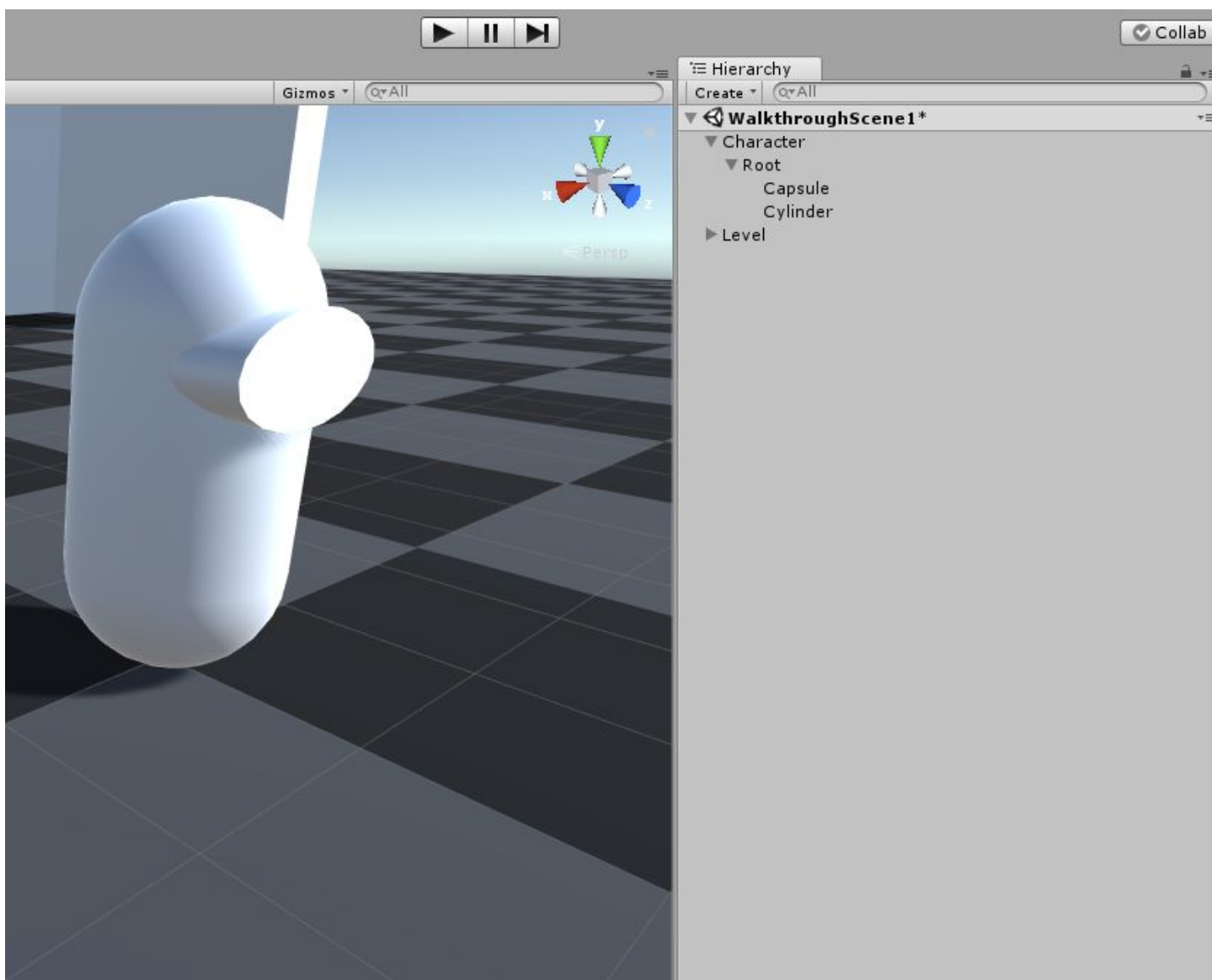
We'll start by creating the general structure for making inputs, characters and cameras work together. Instead of putting input handling and camera control directly into the character controller, we will isolate it in a "MyPlayer" class. The reasoning behind this is that not all character controllers 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 "Capsule 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:



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

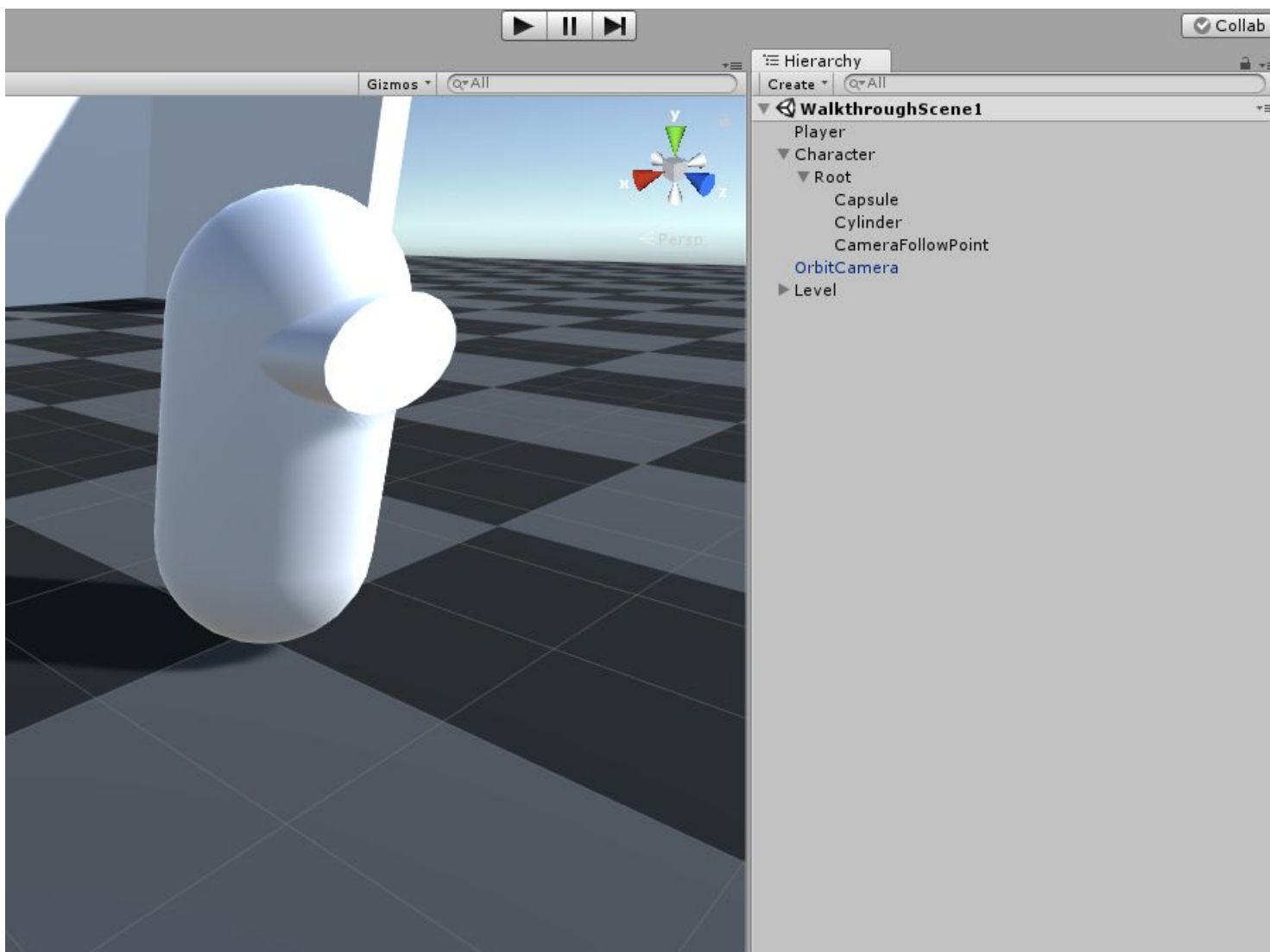
MyPlayer will be where we handle all character and camera input, and make the connections between the camera and the character. For now, let's just take care of camera handling. On Start(), MyPlayer will setup the camera to follow the character, and on Update(), it will calculate input from mouse movement 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:



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'll create a struct that represents the inputs that a player can give to its character.

Notice the "PlayerCharacterInputs" struct we define in MyCharacterController.cs

We will then add a "HandleCharacterInput" method in MyPlayer, which will be called on every Update. This method will simply pass on the inputs from the player to the character.

Look at the implementation of MyPlayer.HandleCharacterInputs(), which in turn calls MyCharacterController.SetInputs()

SetInputs()'s goal is to transform these inputs into information that the character can use for its movement and rotation. Here we build a movement vector and a look direction vector based on the camera orientation and the character's up direction (we want to move in the camera's direction, but only on the character's plane).

With this, we have MyPlayer sending movement inputs to MyCharacterController, but we have yet to turn this into actual character movement.

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.

See the “JumpDown” field we’ve added to the PlayerCharacterInputs struct, and see how we set it in `MyPlayer.HandleCharacterInput()`

And finally, see how we process this jump input in `MyCharacterController.SetInputs()`. Here we remember that we want to jump, and we start keeping track of the time since jump was requested.

Remember that all velocity needs to be processed in `UpdateVelocity`, so this is why we keep track of the fact that that we want to jump and don’t apply the movement right now in some way.

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`, for handling timers and jump states.

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.

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.)

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 current ground status with the previous ground status of the KinematicCharacterMotor during the “PostGroundingUpdate”, which is called right after the character has evaluated its new grounding status.

See the implementation of this in `MyCharacterController.PostGroundingUpdate` (“// Handle landing and leaving ground”)

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. This is done with the "Q" key in this example

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.

See the “CrouchDown” and “CrouchUp” fields we’ve added to the PlayerCharacterInputs struct, and see how we set it in `MyPlayer.HandleCharacterInput()`

And finally, see how we process this input in `MyCharacterController.SetInputs()`. Here we remember our desired crouching state, and apply the capsule rescale if we do crouch

But un-crouching is not handled in `SetInputs`. 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.

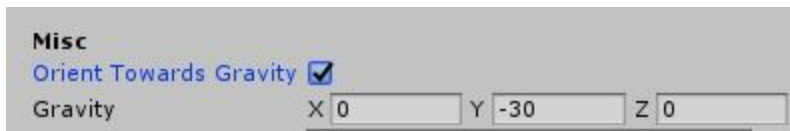
Now try passing under the red module in the scene by crouching with the C key.

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`

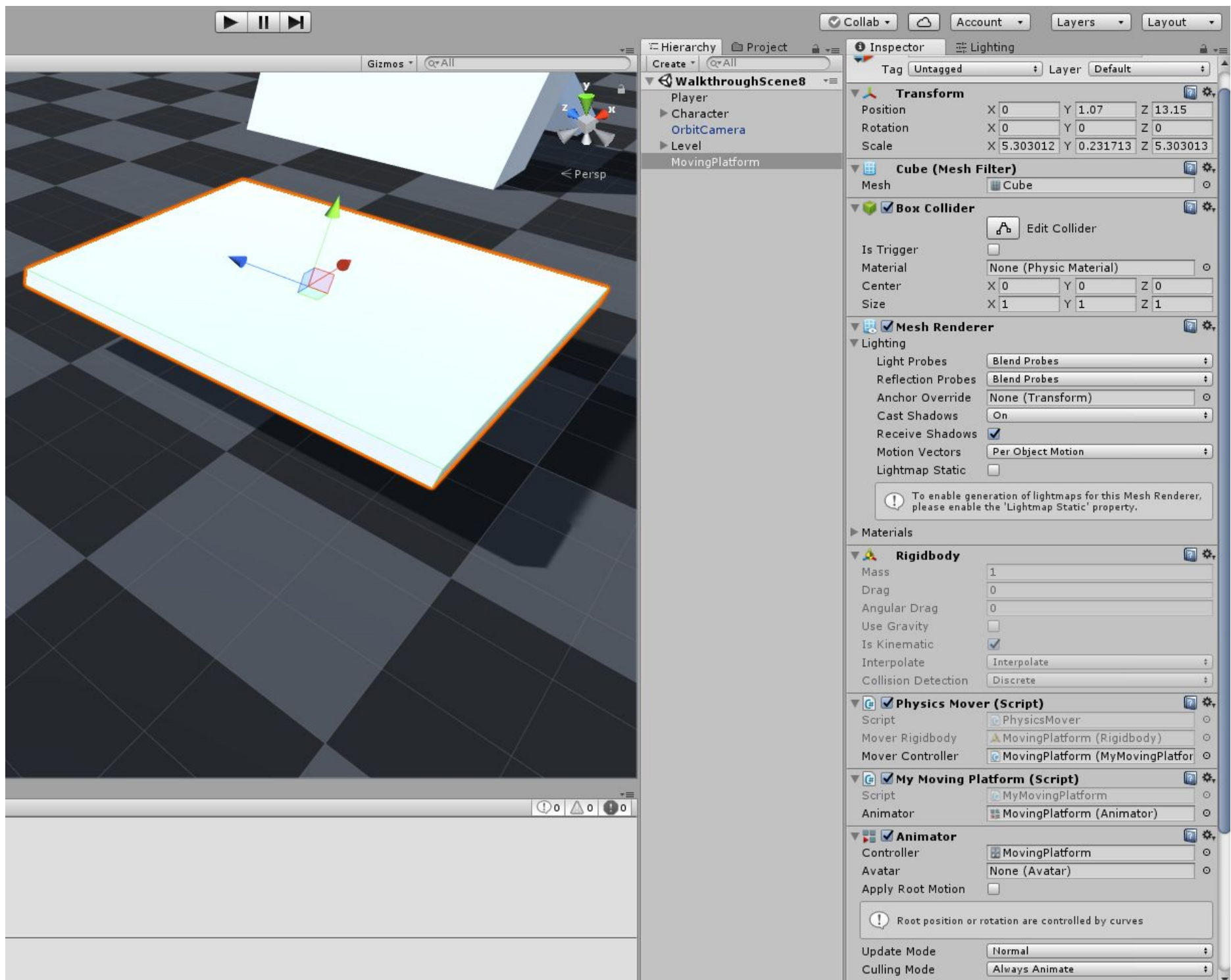
In “UpdateRotation”, which is the method in which you specify what rotation you want your character to have right now, we simply tell the character to rotate from its current up to the inver-gravity direction. This does the trick. Now try to activate “Orient Towards Gravity” in the character’s inspector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



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
- Using a PlayableDirector component and the Timeline window, create a playable animation that moves and rotate that platform



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 make sure our PlayableDirector's update mode is set to "Manual", so we can update it ourselves. We'll then use the Evaluate() method to apply the animation at any time we want

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:

See the “IgnoredColliders” field in `MyCharacterController`

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`

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 "CharacterState" enum representing all the states the character can be in. For now, there's just one Default state.
2. We will add a "CurrentCharacterState" field to MyCharacterController, which will remember what our current state is
3. In all pertinent methods of MyCharacterController, we will add a switch case that'll allow us to write different logic depending on the current state of the character
4. We will add the TransitionToState, OnStateEnter and OnStateExit methods to MyCharacterController to make transitions simpler to manage

Look at the implementation of this in MyCharacterController.

Some of the methods that now have state-dependant logic are:

- SetInputs
- UpdateRotation
- UpdateVelocity
- AfterCharacterUpdate
- OnMovementHit
- AddVelocity
- ProcessHitStabilityReport

We only have one "Default" character 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 add a “Charging” state to the CharacterState enum. Then, we will add some input handling code to call a transition to that state when Q is pressed.

Take a look at the charging input handling in the `MyPlayer.HandleCharacterInput` and `MyCharacterController.SetInputs` methods. The actual state transition happens at the beginning of `SetInputs()`

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

- In `OnStateEnter`, for the Charging state, cache a charging velocity based on the character’s forward direction
- In `UpdateVelocity`, for the Charging state, keep applying the charging velocity every update unless we are in the pause at the end of the charge
- In `OnMovementHit`, for the Charging state, look for collisions with walls in order to trigger the end of the charge
- In `AfterCharacterUpdate`, for the Charging state,
 - 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 some of the physics activation methods 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 character state to `MyCharacterController` (in the `CharacterState` enum). We will call this one “NoClip”. We will also add an input to `PlayerCharacterInputs` for when we are requesting a charge.

Take a look at `MyPlayer` and `MyCharacterController` to see how the Q input is translated to a transition to the NoClip state in `SetInputs()`

Now for the implementation of the state’s logic. 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.SetCapsuleCollisionsActivation()` and `KinematicCharacterMotor.SetCollisionSolvingActivation()` methods. When `SetCollisionSolvingActivation` is set to false, no collision solving is done whatsoever, and the character can go through anything. `SetCapsuleCollisionsActivation` controls the activation of the kinematic capsule collider itself.

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 “JumpHeld” and “CrouchHeld” inputs to the `PlayerCharacterInputs` struct

See how these things were implemented in `MyPlayer` and `MyCharacterController`

Try entering Play mode and pressing “Q” 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 adding the “Swimming” state to the CharacterState enum

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 logic 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 MyCharacterController.UpdateVelocity

Finally, we need to make sure the character doesn’t try to snap to the ground while swimming. For this, we will use KinematicCharacterMotor.SetStabilitySolvingActivation(). Setting this to false will skip all ground probing/snapping logic.

Look at OnStateEnter() and OnStateExit() in MyCharacterController

Now enter Play mode and go in the water to try it out

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 adding a "Climbing" state to the states enum in MyCharacterController, we will implement the transition to that state. This will be done in MyCharacterController.SetInputs, when a "ClimbLadder" input is detected. When ClimbLadder is true, we 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.

Look at MyCharacterController.SetInputs() to see what happens when ClimbLadder is pressed

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.SetCollisionSolvingActivation to disable the character's movement and collision solving code. 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. 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`

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. 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 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 `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 `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 a unitypackage 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 gameobject 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

Note that we created a different inputs struct and a different SetInputs() method in MyCharacter specifically for AI characters. Since AIs do not rely on cameras for move/look directions, it makes sense that their inputs are handled differently. So we now have two input structs in MyCharacterController: PlayerCharacterInputs and AICharacterInputs. We also have one SetInputs() method for each struct.

Look at the two different SetInputs() methods in MyCharacterController. One is used for Player characters, and the other for AI characters. But both use the same movement code in the end

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.

Frame Perfect rotation

Due to the interpolated FixedUpdate nature of the movement of the character, you may notice a certain delay in the character's rotation even if you give it no rotation smoothing at all. This can become a problem in several cases, such as in the context of a First Person Shooter where the gun (a child object of the character) needs to follow your camera perfectly.

Start by entering the scene in play mode, and look around with the camera. You should be able to clearly notice that the object representing the "gun" in your screen is lagging behind. Now, under the MyCharacterController component, activate "Frame Perfect Rotation" and try moving the camera again. You should see that the problem is solved. Let's see how this is done....

The general strategy is this: the character controller rigidbody will keep rotating on FixedUpdate like it always had, but on top of that, we will also rotate the child transform of the character object that contains all the mesh on every frame. In other words; the physics representation of the character will keep rotating on FixedUpdate, but the visual/mesh part will now rotate on Update.

Look at how this was set up in MyCharacterController:

- We created a "PostInputUpdate" method, which is called on every frame after all inputs have been set. It sets the rotation of the "MeshRoot", which is the child transform of the character that contains all meshes
- We created a "HandleRotation" method that is used by both "PostInputUpdate" and "UpdateRotation" to make sure they both process rotation in the exact same way

And finally, look at the "Update" method in "MyPlayer". This is where we call "PostInputUpdate" after we're done applying all inputs.

Now press play and try to rotate your view around. Now try to change the activation of the "Frame Perfect Rotation" field in the inspector. When it is deactivated, you should notice the lag from interpolation.

Networking example

Introduction

This example demonstrates the usage of Kinematic Character Controller in a fully-authoritative server networking architecture, with client-side prediction, lag-compensation, and snapshot interpolation. The underlying networking system implemented here works, but it is not meant to be optimized or production-ready. However, the specific implementation of the kinematic character controller in this system is representative of how you should handle things in a serious, well-made project.

***Note*:**

Due to the great complexity of networking frameworks and the fact that it goes way beyond the intended scope of this character controller package, I will not be able to provide support on how the actual underlying networking system works, or how to make networked games in general. What I can do, however, is provide tips on how to approach networking for Kinematic Character Controller specifically regardless of the networking system you're using, assuming you already have some knowledge of how networked games work.

How to try out the example

Follow these steps to try the networked game:

- Make a build with the "WalkthroughScene18_lobby" scene and "WalkthroughScene18_game" (lobby scene must be first in build order)
- Run two instances of the build
- On the first instance, leave all settings as is, and click "Host". This starts the server on localhost and on port 7777
- On the second instance, set the "LocalHostPort" to 7778 (every different instance on the same machine needs a different LocalHostPort) and click "Connect". You should now see two players in the "Joined Players" panel
- On the server instance, click "Start Game", and the networked game will start

The game puts you in control of a character, and allows you to fire a laser with the left mouse button (you should go in first person view with right mouse button for shooting). When a laser hits a character, 3 things happen:

- A burst of red particles appears at the impact point. This is the instant-feedback that clients show even without a confirmation from the server that the hit was valid
- A "99 DMG" text appears over the hit character. This only happens if the server has really confirmed that the hit was valid. (note that damage doesn't actually do anything. There is no notion of hit points here)
- The hit character is stunned for about 2 seconds. This is very helpful to see if the lag-compensation system is working properly. This also only happens if the server has really confirmed that the hit was valid.

Mandatory reading

Because game networking is an extremely complex and broad subject, there is some mandatory reading to do before moving on with this section. All of those articles cover some similar concepts, but I believe your understanding of these things will only really start to sink in after reading about them from several different sources and perspectives.

- "Fast-Paced Multiplayer", by Gabriel Gambetta (read all 4 parts of the series)
<http://www.gabrielgambetta.com/client-server-game-architecture.html>
- "The DOOM III Network Architecture", by J.M.P. van Waveren
<http://mrelusive.com/publications/papers/The-DOOM-III-Network-Architecture.pdf>
- "Source Multiplayer Networking", by Valve
https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
- "Making Fast-Paced Multiplayer Networked Games is Hard", by Mark Mennell
https://www.gamasutra.com/blogs/MarkMennell/20140929/226628/Making_FastPaced_Multiplayer_Networked_Games_is_Hard.php
- "Snapshot Interpolation", by Glenn Fielder (most of his other articles are very pertinent too, but just this one will do for now)
https://gafferongames.com/post/snapshot_interpolation/

Overview of the system

While the actual implementation of a networking framework for your game isn't the subject of this walkthrough section, it's still very pertinent to get a basic understanding of how it all works before moving on to the actual character controller implementation.

Here are the main components:

- **OnlineSession:** This is essentially a sort of wrapper for UNET LLAPI. It is used to manage connections, send messages, connect, receive network events, etc.... If you'd ever want to use another networking library instead of UNET LLAPI, all you'd have to do is modify this relatively small class
- **SimulationSystem:** This is the core of the networking architecture. You start by giving it types for your PlayerControllers, your PlayerCommands and your WorldStater, and this system will handle all tick simulation and re-simulation from the past in order to create a fully authoritative server architecture, with lag-compensation and client-side prediction. It also manages keeping track of world snapshots (saving/reverting inputs)
- **KinematicCharacterSubsystem:** This is the important class to see if you want to specifically understand how KinematicCharacterController should be networked. This class handles ticking character simulation, and saving and restoring character states
- **MyGameManager:** This is the class that handles initializing the SimulationSystem for the game scene, and managing all network message events

The order of things basically goes like this:

- A LobbyManager is present in the lobby scene. This manager creates the OnlineSession which will persist into the game scene later. As players join the lobby, the LobbyManager builds a dictionary of "JoinedPlayers", which contains the player data of the players.
- When launching the game, a SceneLoadingMsg is sent to all clients to load the game scene. In the game scene, a MyGameManager is present. This MyGameManager creates the SimulationSystem for this game type and registers the KinematicCharacterSubsystem into the SimulationSystem. When clients are done loading the game scene, they send a msg to the server to signal that they are ready and synchronized with the server's tick.
- Once all clients have signaled that they are ready, the MyGameManager on the server spawns all characters, and the simulation ticking begins.
- The KinematicCharacterSubsystem receives callbacks from the SimulationSystem, which tells it when to do a simulation step, when to revert the world to a specific tick, when to save the world at a tick, etc....

How to “Tick” the character movement simulation

Whether it's to simulate the present frame or to resimulate past frames, these sorts of networking systems need a notion of “ticking” the simulation for all gameplay-related entities.

This is done in **KinematicCharacterSubsystem**.

- Upon initialization, we make sure to turn off KinematicCharacterSystem.AutoSimulation so we can simulate things manually.
- Right before the SimulationSystem simulates a frame, the SimulationSystem tells all PlayerControllers to apply their inputs to their character
- In **OnSimulateStep()**, which is called by the SimulationSystem every time we need to simulate a frame, we call KinematicCharacterSystem.Simulate(deltaTime). This is what “Ticks” the character simulation

How to save the character state

At every tick, we need to save the state of the whole game world so that we can revert to it later if we have to. Each type of game entity has their own way of saving their state, but here's how it works for the Kinematic Character Controller

This is done in **KinematicCharacterSubsystem.SaveToSnapshot()**

- We first save the state of the KinematicCharacterMotor component by using Motor.GetState()
- Then, we save the state for our custom character controller component. This will be different for every character, but here we save state info related to jumping and the stunning effect of the laser

How to restore a character state

To compensate for lag, we frequently have to restore the world to a previous saved state. Each type of game entity has their own way of restoring their state, but here's how it works for the Kinematic Character Controller

This is done in **KinematicCharacterSubsystem.RestoreStateFromSnapshot()**

- We first restore the state of the KinematicCharacterMotor component by using Motor.ApplyState(). The state we apply here is the one we previously saved using Motor.GetState()
- Then, we restore the state for our custom character controller component. This will be different for every character, but here we restore state info related to jumping and the stunning effect of the laser

What has to be networked?

There is a difference between what state information has to be saved to world snapshots locally, and what has to actually be sent over network for synchronization. Only a few key things have to be synchronized for two peers to recreate the same simulation for the character, because the simulation is deterministic enough to allow that.

This is done in **KinematicCharacterSubsystemState.ReadFromBuffer()** and **WriteToBuffer()** (you can find it in the same file as KinematicCharacterSubsystem)

We only need to sync 3 things for the KinematicCharacterMotor:

- MotorState.Position
- MotorState.Rotation
- MotorState.BaseVelocity (this can actually be omitted, or synchronized less often if you really want to save bandwidth, but you might start to feel position corrections much more often if you do that)

As for our custom character controller, we only sync the "IsStunned" variable

How interpolation is handled

In an online game context, interpolation is handled differently than in an offline game. While "local" entities use regular fixedDeltaTime interpolation, "remote" networked entities use an interpolation based on an estimated average time between network updates, with an extra padding in case of packet loss. For this reason, we don't use any built-in interpolation for the character, and we instead handle it all manually.

This is done in **KinematicCharacterSubsystem**

- Upon initialization, we make sure to set KinematicCharacterSystem.InterpolationMethod to "None" so we can handle interpolation manually.
- Right before the SimulationSystem simulates a frame, it calls **KinematicCharacterSubsystem.OnPreTick()**. Here we save the positions, rotations and times that our "local" characters are going to interpolate from
- Right after the SimulationSystem simulates a frame, it calls **KinematicCharacterSubsystem.OnPostTick()**. Here we save the positions, rotations and times that our "local" characters are going to interpolate to
- Right before the SimulationSystem receives a world snapshot from the server, it calls **KinematicCharacterSubsystem.HandlePreSnapshotInterpolation()**. Here we save the positions, rotations and times that our "remote" characters are going to interpolate from
- Right before the SimulationSystem receives a world snapshot from the server, it calls **KinematicCharacterSubsystem.HandlePostSnapshotInterpolation()**. Here we save the positions, rotations and times that our "remote" characters are going to interpolate to
- In **UpdateInterpolation()**, which is called on every Update(), we move the visual transforms of all our characters and physicsMovers from their "FromInterpolationPoint" to their "ToInterpolationPoint", based on the time between the two points

Note:

On clients, if you try looking at a remote character on a moving platform, you'll see that they appear to be desynchronized. This is because moving platforms are interpolated using local fixedDeltaTime, but remote characters are interpolated based on an

approximation of the time between network updates. Try turning off interpolation in the inspector of the KinematicCharacterSubsystem, on the “GameManager” object in the game scene, and you’ll see that the remote character now appears perfectly synchronized with the platform. This is a problem that is currently not solved and that might be revisited at some point, but it has a lot more to do with the custom interpolation solution we made for this example than with the actual Kinematic Character Controller code. Perhaps we would need to come up with an interpolation solution based on Motor.BaseVelocity and Motor.AttachedRigidbodyVelocity for this to work properly