Kinematic Character Controller

User Guide

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

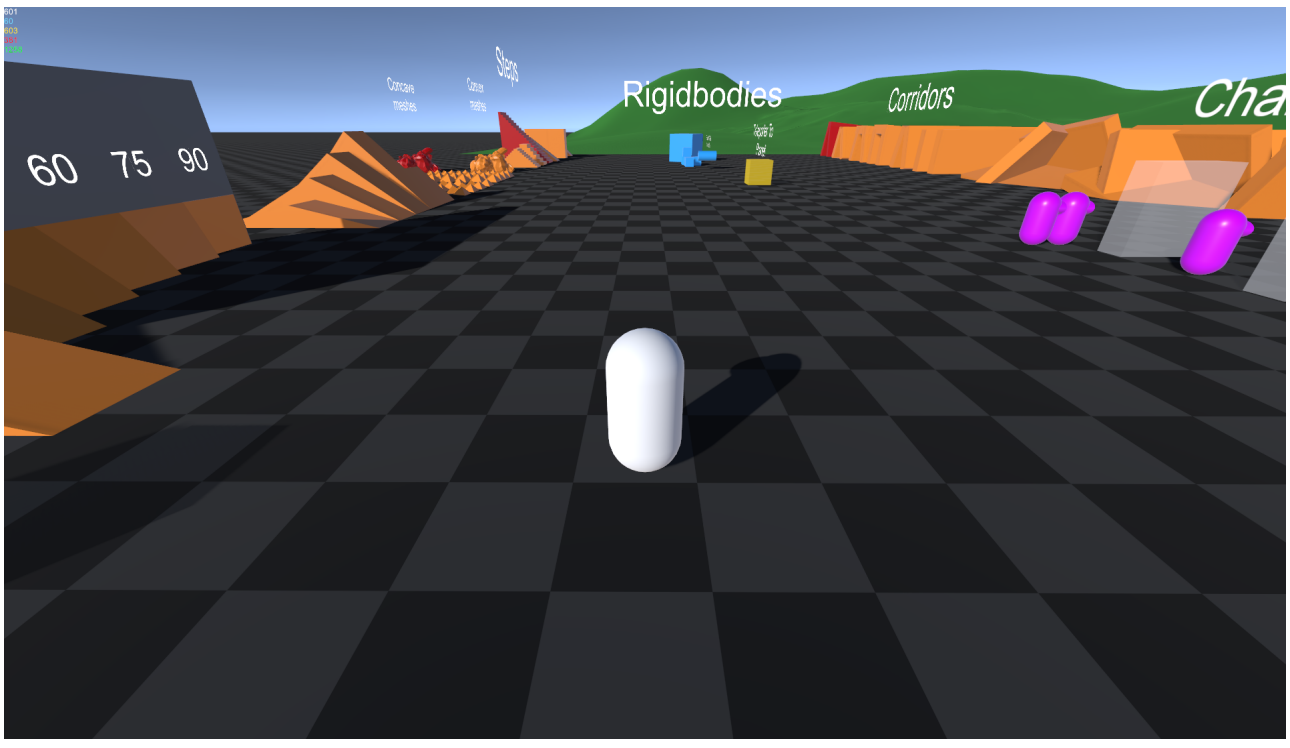
Table of contents

Quickstart

The CharacterPlayground scene

To try out the example character controller right away:

1. Import the KinematicCharacterController package into Unity
2. Open the “CharacterPlayground” scene under *KinematicCharacterController/Examples/Scenes*
3. Press play and try out the example character



The scene is arranged as follows:

- The “**Player**” object is responsible for sending input to the player’s character and camera, and also to give the character information about the camera’s orientation
- The “**Character**” object is the character you control in play mode
- The “**Camera**” object is the game’s camera
- The “**Level**” object contains all other level geometry and content, arranged in different modules
- After pressing play, a “**KinematicCharacterSystem**” object is automatically spawned through singleton-like instantiation in order to handle updating all characters and moving platforms in the correct sequence

The controls are:

- W, S, A, D to move
- Mouse to look around
- MouseWheel to zoom camera in/out
- Space to jump
- C to crouch

Package Overview

Package structure

- **Core**: This folder contains the core scripts of the character system. These scripts are the ones you must absolutely keep if you use this package.
- **ExampleCharacter**: This folder contains an optional example character and camera that is meant to demonstrate the usage of the aforementioned core scripts. This entire folder can be deleted if you don't need it, but the Walkthrough folder depends on it.
- **Examples**: This folder contains the rest of the optional example content. This entire folder can be deleted if you don't need it, but the Walkthrough folder depends on it.
- **Walkthrough**: This folder contains a series of fully-documented examples of character controller feature implementations. See the "Walkthrough" document for more information. This entire folder can be deleted if you don't need it.

Core scripts

- **KinematicCharacterMotor**: This is the heart of the character system. This script solves all character movement and collisions based on a given velocity, orientation, and other factors.
- **PhysicsMover**: This script moves kinematic physics objects (moving platforms) so that characters can properly stand on the and be pushed by them.
- **KinematicCharacterSystem**: Handles simulating the KinematicCharacterMotors and PhysicsMovers in the correct order.
- **ICharacterController**: Make a class that implements this interface and assign it to a KinematicCharacterMotor in order to implement your character controller
- **IMoverController**: Make a class that implements this interface and assign it to a PhysicsMover in order to implement your mover controller

Example scripts

- **ExampleCharacterController**: Handles communicating with KinematicCharacterMotor in order to create a moveable character.
- **ExampleMovingPlatform**: Uses math functions to move a PhysicsMover around and create a moving platform.
- **ExamplePlayer**: Handles input for the ExampleCharacterController, serves as a manager for the camera and character.
- **ExampleAIController**: Simulates input for AI characters in the scene
- **ExampleCharacterCamera** : A simple camera script that can orbit around a character

How to use this package

What to expect?

Character controllers are infamously difficult to pull off well, and every game will have vastly different needs for them. The possibilities are unlimited. For this reason, Kinematic Character Controller does not try to come pre-packaged with every possible solution to every problem. Instead, it focuses on solving the difficult core physics problems of creating highly dynamic and responsive character controllers, and leaves all the rest in your hands. This design philosophy makes sure you have the power to make character controllers that are tailor-made for your specific game, and that can fit cleanly into any project architecture.

This package expects you to write your own code for: player input, camera handling, animation, and even character movement (telling it what its velocity should be, what its orientation should be), etc….. What it does provide, however, is a set of low-level components that handle complex character physics-solving and help you write fully-custom character controllers with relative ease.

An example character controller is provided in the “CharacterPlayground” scene, but this is really just an *example* that serves as a proof of concept and/or learning resource. It is not meant to be a final solution that could fit any project.

How does it work?

This entire package revolves around the “KinematicCharacterMotor” component, which represents the character capsule and solves movement properly given a set of inputs (velocity, rotation, etc….). When you give it a certain velocity, it will run its movement code and make it collide/slide on surfaces appropriately. Additionally, it gives you proper information on its grounding status, handles standing on moving platforms, handles pushing other rigidbodies, etc, etc… This is what you will build your character controllers on.

In order to give those inputs to a KinematicCharacterMotor, you need to create your own custom class that implements the ICharacterController interface, and assign it to the KinematicCharacterMotor.CharacterController variable. By doing this, your class will now start receiving “callbacks” fomr the motor. Most of these “callbacks” can be interpreted as questions that are asked by the KinematicCharacterMotor:

- **UpdateVelocity**: “What should my velocity be right now?”
- **UpdateRotation**: “What should my orientation be right now?”
- **IsColliderValidForCollisions**: “Can I collide with this collider, or should I pass right through it?”
- Etc…..

These callbacks are all automatically called at the right time by KinematicCharacterMotor in the character update loop, so you don’t have to worry about the execution order of things. By implementing them, you can tell the KinematicCharacterMotor exactly how you want it to behave.

The same principles apply to the creation of moving platforms. In that case, “PhysicsMover” fills the same role as KinematicCharacterMotor, and “IMoverController” fills the same role as ICharacterController. By implementing the callbacks of a PhysicsMover, you can tell your moving platform exactly where you want it to go.

All the things contained under the “Example” and “Walkthrough” folders of this package are not really part of the Kinematic Character Controller system. They are just learning resources to help you get started.

How to get started?

There are two main ways I would suggest for getting started learning all this:

1. Read the “Example Character Overview” section of the User Guide
2. Find the “Walkthrough” document and project folder. This contains a very in-depth series of exercises with full sources available. It will walk you step-by-step through the creation of an entire custom character controller from A to Z, and also contains an example for creating a moving platform with PhysicsMovers.

An API reference is also available with the project, in HTML form. Simply extract the .zip and open “APIReference.html” to access it.

Example Character Overview

This section will give a summary of how several components work together in order to create the example character in the “CharacterPlayground” scene. Open the scene and follow along with this section.

The Components

Here are the main components that are important for the character:

- **ExamplePlayer (on the Player object)**: This is the class that handles player input, and serves as a link between the character and the camera
- **ExampleCharacterCamera (on the Camera object)**: This handles camera movement around a specified transform to follow
- **ExampleCharacterController (on the Character object)**: This is the actual custom character controller script that implements the motor’s callbacks. You are expected to do one of these by yourself if you want to create a custom character controller
- **KinematicCharacterMotor (on the Character object)**: A core component of the package that communicated back and forth with the character controller. Its job is mostly to solve all character physics given a velocity and a rotation

Note that nothing forces you to separate things in a player, a camera, and a controller. It’s just one way of doing things. The only mandatory component here is KinematicCharacterMotor.

Input handling in ExamplePlayer

In its Update(), the ExamplePlayer script handles input for the camera and for the character (in HandleCameraInput() and HandleCharacterInput(), respectively).

For the camera, it sends mouse movement and scroll wheel movement to the ExampleCharacterCamera with UpdateWithInput()

For the character, it builds a struct containing all the input information that the character will need, and sends that to the ExampleCharacterController with SetInputs()

Camera movement

When UpdateWithInput() is called from ExamplePlayer, the camera script automatically calculates the new pose from that input, and applies it instantly.

Character movement

When SetInputs() is called from ExamplePlayer, the ExampleCharacterController processes all of these inputs and stores information about its move direction, its look direction, its jumping and crouching states, etc….

Later, when the KinematicCharacterSystem will call its update cycle on all characters (this happens in a FixedUpdate by default), the ExampleCharacterController will use that processed information in its various callbacks that it receives from KinematicCHaracterMotor. In UpdateVelocity, it’ll handle calculating what its current velocity should be. In UpdateRotation, it’ll calculate its new rotation, etc, etc…

See the next section for more information on the KinematicCharacterSystem

Kinematic Character System Overview

This section gives an overview of how all of the “core” scripts work together in order to handle character movement properly and create the character system. Your KinematicCharacterMotors and PhysicsMovers require a very specific execution order in order to work as expected. This is handled by the KinematicCharacterSystem class.

Components Registration

When KinematicCharacterMotors and PhysicsMovers are created, they register themselves into the KinematicCharacterSystem in OnEnable(). Similarly, they unregister themselves in OnDisable(). The KinematicCharacterSystem will then handle the update behaviour of all registered KinematicCharacterMotors and PhysicsMovers in the correct order.

Basic Simulation Loop

In the FixedUpdate(), if AutoSimulation is true, the KinematicCharacterSystem does the following:

- **PreSimulationInterpolationUpdate()**
  - Handles saving all registered motor and mover initial poses before any movement has been done.
  - Handle finishing interpolation
- **Simulate()**
  - Calculates velocities for all PhysicsMovers (based on what their target poses should be).
  - Calls UpdatePhase1() on all KinematicCharacterMotors, which solves initial overlaps and handles grounding logic.
  - Places all PhysicsMovers at their destination directly.
  - Calls UpdatePhase2() on all KinematicCharacterMotors, which solves movement and calculates what the motor’s final pose should be. Then, move motors to their target poses.
- **PostSimulationInterpolationUpdate()**
  - This will make all motors and mover transforms move to the poses saved in PreSimulationUpdate()

Manual Simulation

If you need precise control over the simulation, you can set KinematicCharacterSystem.AutoSimulation to false. This will give you the responsibility of handling the simulation yourself. This can be useful in the context of networking, where you may need to call Simulate() several times within the same frame to re-simulate past inputs.

Look at KinematicCharacterSystem.FixedUpdate() to observe how the default autoSimulation loop is done

Additional information

Important character controller notes:

- If you truly want to “teleport” your character instead of making it move, use KinematicCharacterMotor.SetPosition()
- Never make the character a child of a moving transform.
- The character gameObject’s lossy scale **must** be (1,1,1), otherwise the physics calculations will not work properly. This means that all parents of that object must also have a (1,1,1) scale. You will receive errors in editor if this condition is not respected. However, you are free to set any scale you want on child objects.
- Always use the “SetCapsuleDimensions” method of KinematicCharacterMotor if you want to resize the capsule during play, because it caches information about the capsule’s dimensions that is later used by most of the movement code.

Physics queries capacity

The KinematicCharacterMotor uses non-GC-allocating methods for physics queries, which means it has arrays with fixed sizes to store the results of these queries. By default, it can support up to 32 RaycastHit results and 32 collider overlap results. If you need more, feel free to modify the “MaxHitsBudget” and “MaxCollisionBudget” constants in KinematicCharacterMotor.

Interpolation

You can activate or deactivate interpolation for all KinematicCharacterMotors and PhysicsMovers by modifying “Interpolate” in KinematicCharacterSystem

Root motion

You can make the character move with animation root motion if you want to. All you need to do is store the animator’s root motion (animator.deltaPosition) in OnAnimatorMove() in your custom character controller, and convert it to a velocity (animator.deltaPosition / deltaTime) so that you can set the motor’s velocity in the “UpdateVelocity” callback of the motor. Same goes for rotation and the “UpdateRotation” callback. An example of this is available in the Walkthrough.