Kinematic Character Controller

User Guide

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

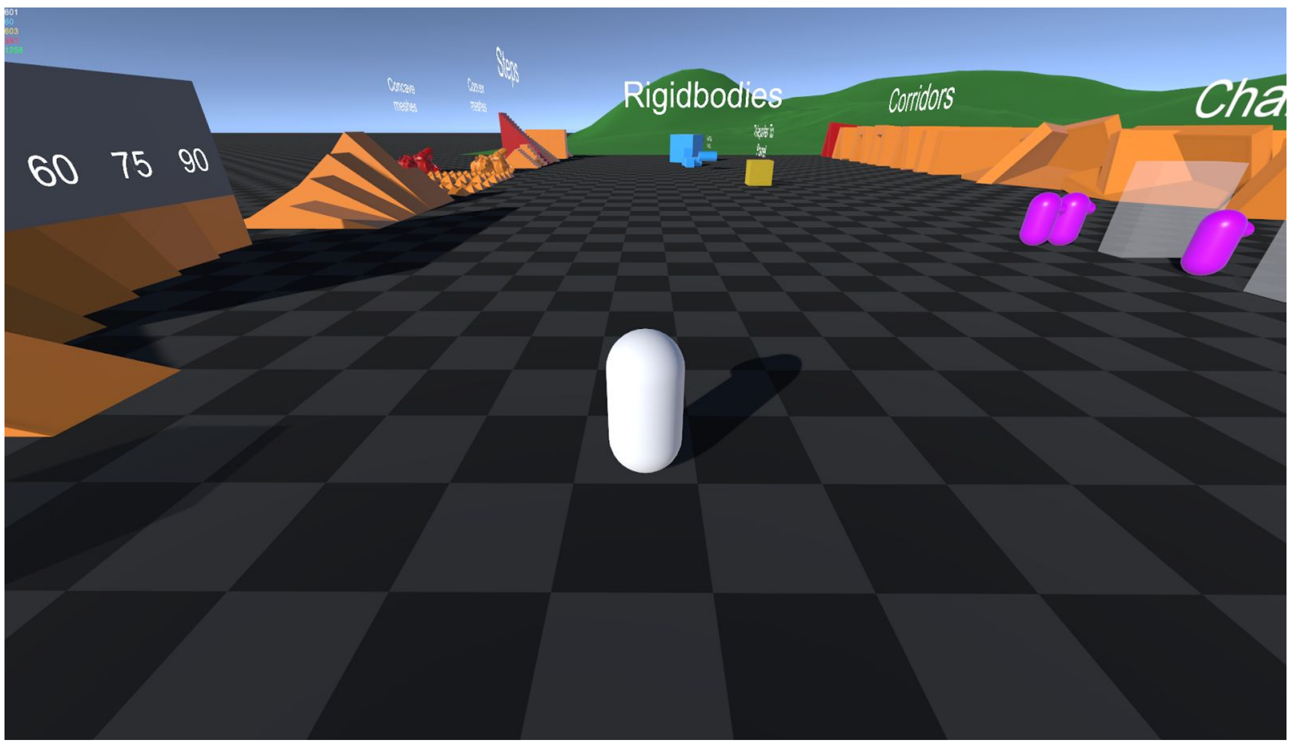
Table of contents

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 “**OrbitCamera**” 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.
- **Examples**: This folder contains all the optional example content that is meant to demonstrate the usage of the aforementioned core scripts. This entire folder can be deleted if you don't need 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.
- **BaseCharacterController**: BaseCharacterController is the class you inherit from in order to create custom character controllers. It's a component that receives callbacks from KinematicCharacterMotors, and tells it how to behave through these callbacks.
- **PhysicsMover**: This script moves kinematic physics objects (moving platforms) so that characters can properly stand on the and be pushed by them.
- **BaseMoverController**: BaseMoverController is the class you inherit from in order to create custom physics mover controllers (moving platforms). It's a component that receives callbacks from PhysicsMover, and tells it how to behave through these callbacks.
- **KinematicCharacterSystem**: Handles updating the KinematicCharacterMotors and PhysicsMovers in the correct order.

Example scripts

- **ExampleCharacterController**: Handles communicating 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

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 a child class of "BaseCharacterController", implement its abstract methods, and assign it to a KinematicCharacterMotor. The abstract methods that you need to implement for your child class of BaseCharacterController 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 "BaseMoverController" fills the same role as BaseCharacterController. By implementing the abstract methods of BaseMoverController and assigning it to 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?

Aside from reading the User Guide, there are two main ways I would suggest for getting started learning all this:

1. 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.
2. Open the "CharacterPlayground" scene, and study "ExampleCharacterContoller", "ExamplePlayer", and "ExampleMovingPlatform". You can even copy them, rename them to something different, and use that as a starting point for your own character controller.

Core scripts reference

BaseCharacterController

Summary

BaseCharacterController is the class you inherit from in order to create custom character controllers. It's a component that receives callbacks from KinematicCharacterMotors, and tells it how to behave through these callbacks.

Fields

- **KinematicCharacterMotor**: This is a reference to the KinematicCharacterMotor component assigned to this character controller

Methods

- **SetupCharacterMotor**: Called automatically by the KinematicCharacterMotor on Awake to setup references
- **MustUpdateGrounding**: This asks if the character motor should probe for ground and snap to surfaces on this update tick. Return false if you want to skip ground probing, and return true otherwise
- **UpdateRotation**: This gives you a reference to the current rotation of the character in the character motor's calculations, and gives you an opportunity to modify it. Grounding information is up to date at the moment when this is called
- **UpdateVelocity**: This gives you a reference to the current velocity of the character in the character motor's calculations, and gives you an opportunity to modify it. Grounding information is up to date at the moment when this is called
- **BeforeCharacterUpdate**: This gives you an opportunity to add logic before the KinematicCharacterMotor update cycle begins
- **AfterCharacterUpdate**: This gives you an opportunity to add logic once the entire KinematicCharacterMotor update cycle has finished
- **IsColliderValidForCollisions**: This asks if the collider passed as argument should be considered for collisions. Return false if you want to ignore collisions with it
- **CanBeStableOnCollider**: This asks if the character should be allowed to be "stable" on the collider passed as argument
- **OnGroundHit**: A callback received for all ground probing hits
- **OnMovementHit**: A callback received for all movement hits

KinematicCharacterMotor

Summary

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.

Fields

- **TransientPosition**: The character's position in the movement calculations
- **TransientRotation**: The character's rotation in the movement calculations
- **CharacterController**: The BaseCharacterController child component that controls this motor's behaviour
- **CharacterCapsule**: The CapsuleCollider used by the CharacterMotor
- **CharacterRigidbody**: The character's kinematic rigidbody
- **CharacterTransform**: The character's transform
- **FoundAnyGround:** Has the character found any surface with its ground probing?
- **IsStableOnGround**: Is the character considered "stable" on a surface right now?
- **WasStableOnGround**: Was the character stable during the last update?
- **GroundNormal**: The normal of the current ground
- **GroundCollider**: The collider of the current ground
- **GroundColliderTransform**: The transform of the current ground
- **StableInteractiveRigidbody**: The dynamic rigidbody or the PhysicsMover's rigidbody of the current ground, if applicable.
- **CollidableLayers**: The layermask that this character can collide with (uses the same as what's defined in the physics collisions matrix for the gameobject's layer)
- **CharacterUp/Right/Forward**: Character directions that are cached and up to date in the character motor's calculations. You should always use these instead of transform.up/right/forward
- **Velocity**: The character's absolute velocity in world space
- **BaseVelocity**: The part of the character's velocity that is caused by actual character movement
- **StableInteractiveRigidbodyVelocity**: The part of the character's velocity that is caused by standing on an interactive rigidbody

Methods

- **HandlePhysics:** Controls the activation or deactivation of movement solving and capsule collisions
- **SetCapsuleDimensions/SetCapsuleDimensionsAuto**: Resizes the character's capsule. It is very important to always resize the capsule using this method, because it caches information about the capsule's dimensions. The "Auto" version of this method is for resizing the capsule relatively to the character's transform origin
- **CharacterOverlap**: Returns all colliders that the character would be overlapping with at a given position and rotation. Ignores the character capsule
- **CharacterCollisionsOverlap**: Returns all colliders that the character would be overlapping with at a given position and rotation. Ignores the character capsule, and only takes into account colliders that the character could physically collide with
- **CharacterSweep**: Acts as a CapsuleCastNonAlloc using the character's dimensions and returns all hits. Ignores the character capsule
- **CharacterCollisionsSweep**: Acts as a CapsuleCastNonAlloc using the character's dimensions and returns all hits. Ignores the character capsule, and only takes into account colliders that the character could physically collide with
- **ForceUnground**: Forces the character to un-snap from ground on its next grounding update. Useful for jumping or applying any force on the character that would lift it into the air. If you tried applying an upwards velocity to the character without calling ForceUnground() first, the character would just keep snapping to ground and wouldn't move up
- **GetDirectionTangentToSurface**: Returns the direction adjusted to be tangent to a specified surface normal relatively to the character's orientation. Useful for reorienting a direction on a slope without any deviation in trajectory
- **ProbeGround**: Use this to probe for ground at any given pose. The method will modify its "transientPosition" passed as ref parameter if ground snapping would occur. The results of the ground probing can be found in the GroundProbingReport
- **GetVelocityForMovePosition**: Calculates the velocity required to move the character to the target position over a specific deltaTime. Useful for when you wish to work with positions rather than velocities in the UpdateVelocity callback of BaseCharacterController

BaseMoverController

Summary

BaseMoverController is the class you inherit from in order to create custom physics mover controllers (moving platforms). It's a component that receives callbacks from PhysicsMover, and tells it how to behave through these callbacks.

Fields

- **PhysicsMover**: This is a reference to the PhysicsMover component assigned to this mover controller

Methods

- **SetupMover**: Called automatically by the PhysicsMover on Awake to setup references
- **UpdateMovement**: Called on every character system update. This is where you must tell the PhysicsMover where to move and rotate to at all times

PhysicsMover

Summary

This script handles movement for kinematic physics objects (moving platforms) so that characters can properly stand on the and be pushed by them.

Fields

- **MoverController**: The BaseMoverController child component assigned to this PhysicsMover
- **MoverRigidbody**: The rigidbody of this PhysicsMover

Kinematic Character System In-depth

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.

The KinematicCharacterSystem

(Refer to the KinematicCharacterSystem.cs script for this section)

Your KinematicCharacterMotors and PhysicsMovers require a very specific execution order in order to work as expected. This is handled by the KinematicCharacterSystem class.

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.

In the FixedUpdate(), the KinematicCharacterSystem does the following:

- Call **CalculateVelocities()** for all PhysicsMovers
- Call **CharacterUpdatePhase1()** for all KinematicCharacterMotors
  - Within this phase, the KinematicCharacterMotors will ask their assigned character controllers to give them their velocity via the “UpdateVelocity” callback
- Call **SimulateAtGoal()** for all PhysicsMovers
- Call **CharacterUpdatePhase2()** for all KinematicCharacterMotors
  - Within this phase, the KinematicCharacterMotors will ask their assigned character controllers to give them their target rotation via the “UpdateRotation” callback
- Call **CharacterUpdatePhase3()** for all KinematicCharacterMotors
- Call **Desimulate()** for all PhysicsMovers
- If you are working in Unity 2017.2 or above, call **Physics.SyncTransforms()** at this point. It is highly suggested to have it turned off for performance reasons, but make sure you clearly understand what it does if you choose to do so. You don’t need this step at all if you are in 2017.1. (note: you need this step in 2017.2 regardless of if you have Physics.autoSyncTransforms on or off)
- Call **UpdateMovement()** for all PhysicsMovers

Refer to the next few sections for a deeper look at these methods.

PhysicsMover.CalculateVelocities

This step tells PhysicsMovers to evaluate their target pose by calling BaseMoverController.UpdateMovement, and then calculates the required velocity and angularVelocity to reach that pose before the next update cycle. The velocities are stores into the rigidbody.velocity and rigidbody.angularVelocity fields.

KinematicCharacterMotor.CharacterUpdatePhase1

Resolve initial overlaps

The character motor will de-collide from any collidable objects

Ground probing and snapping

The character will cast downwards to detect ground, and remember pertinent information such as grounding status, ground normal, ground collider, etc… Additionally, if a ground was found, the character motor will modify its position so that it “snaps” to the surface with a small offset.

Velocity update

Here the character motor asks its character controller to calculate the desired velocity with BaseCharacterController.UpdateVelocity.

Interactive rigidbody handling

An “interactive rigidbody” is the name given to any PhysicsMover or non-kinematic rigidbodies. They are considered as a special case by the movement code, because the character has to be able to stand on them, be pushed by them, and push against them. In this step, we detect if the stable ground collider is an interactive rigidbody. If it is, we store the velocity that this interactive rigidbody’s movement would apply on the character as a result of standing on it.

Calculate movement from interactive rigidbody

The character motor will process movement and collisions, but only for the velocity caused by stable interactive rigidbodies (if any). The end position of this movement will be remembered for later.

PhysicsMover.SimulateAtGoal

The mover will place its rigidbody pose and transform pose directly at its goal pose. The rigidbody pose update is required because the character movement physics queries in the next steps will require up-to-date information. The transform pose update is required because the ComputePenetration query used by the character will need the up-to-date transform pose as well as the rigidbody’s.

KinematicCharacterMotor.CharacterUpdatePhase2

With the PhysicsMover poses now updated, some overlaps with the character might have happened. This is where we solve them

Solve potential stable interactive rigidbody overlap

When standing on an interactive rigidbody that rotates around an axis that does not coincide with the character's up direction, it can create a situation where the rigidbody's collider overlaps with the character's. This will in turn cause the character to solve the overlap in the following step. However, solving the overlap this way will cause the character to slightly deviate from the point where it was originally standing on the rigidbody. To solve this deviation, we do a special downward capsule cast and move the character along its up direction to snap directly on the stable surface.

Resolve physics movers overlaps

This step will resolve any other overlap that might have been caused by a physics mover moving into the character.

Calculate character movement

This is the point where we process movement and collisions for the character's desired velocity (also known as "baseVelocity"). The final calculated position is remembered for later.

Rotation update

Here the character motor asks its character controller to calculate the desired rotation with BaseCharacterController.UpdateRotation.

Apply final movement

Here the character motor moves its rigidbody pose to match the end goal pose calculated by all of the character movement code. We need to move the rigidbody directly, because other character motors will need up-to-date information to properly process inter-character collisions.

KinematicCharacterMotor.CharacterUpdatePhase3

Once all of the character motors have properly calculated their end goal poses, we reset the rigidbody pose to what it was at the beginning of the update. We then move the rigidbody with correct interpolation using rigidbody.MovePosition and rigidbody.MoveRotation.

PhysicsMover.Desimulate

The mover resets its rigidbody and transform poses back to where they were at the start of the update. This is required for interpolation to work correctly in the final step.

Physics transforms sync

This step only happens in 2017.2 if the autoSyncTransforms feature is turned off. We sync all rigidbodies with their transforms. Again, this is required for interpolation to work correctly in the final step.

PhysicsMover.UpdateMovement

The mover moves its rigidbody to its end goal with correct interpolation using rigidbody.MovePosition and rigidbody.MoveRotation.

Additional information

Important character controller notes:

- Attempting to move the character in any way other than the BaseCharacterController's "UpdateVelocity" method override will result in unexpected behaviour. Only move the character with transform.position when you wish to teleport it without handling collisions.
- 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.
- The KinematicCharacterMotor will use the proper collision layers corresponding to the character gameObject's layer for its calculations.
- 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.
- Your custom character controller component (such as ExampleCharacterController) does not absolutely have to be on the same object as the KinematicCharacterMotor. It can be anywhere.

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 rigidbody interpolation for all KinematicCharacterMotors and PhysicsMovers by modifying the "UseInterpolation" constant in KinematicCharacterSystem

AutoSyncTransforms

Setting Unity 2017.2's "Physics.autoSyncTransforms" feature to false is fully supported. The manual transforms sync is done at some point during the KinematicCharacterSystem update. However, make sure you understand exactly what this option does before turning it off.

AutoSimulation

Setting "Physics.autoSimulation" to off and simulating physics manually on Update can be supported, but it will make PhysicsMover interactions much less accurate because they would usually rely on predicting exactly where they'll be in the future with the fixed deltatime. However, all of the regular character movement code would work perfectly fine, so a variable physics timestep would be perfect for a game where moving platforms do not have an important role. All that would be required to make this change is to:

- Change the "FixedUpdate" in KinematicCharacterSystem to an "Update"
- Be sure that "Physics.Simulate(Time.deltaTime)" is called exactly once per Update. If you're not sure where to call it, call it at the beginning of the update cycle in KinematicCharacterSystem
- Set the "UseInterpolation" constant to false 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 BaseCharacterController. Same goes for rotation and the "UpdateRotation" callback. An example of this is available in the Walkthrough.