KCC Plugin

user guide

for https://github.com/zode/kcc
minimum Flax Engine: 1.9

Table of Contents

.cs script content overview

KCC	The game plugin itself
KCCSettings	Container class for plugin settings
KinematicBase	Base class from which KinematicMover and KinematicCharacterController extend from
KinematicMover	Optional utility class for writing moving kinematic objects (platforms, elevators, etc)
KinematicCharacterController	The “secret sauce”, this class contains all of the movement logic
IKinematicMover	The necessary interface to control a KinematicMover with
IkinematicCharacterController	The necessary interface to control a KinematicCharacterController with
ColliderType	Enum for the KinematicCharacterController’s shape
GroundingState	Enum for the KinematicCharacterController’s grounding state
RigidBodyInteractionMode	Enum for the KinematicCharacterController’s RigidBody interaction solver mode
RigidBodyMoveMode	Enum for the KinematicCharacterController’s RigidBody move-with solver mode
StairStepGroundMode	Enum for the KinematicCharacterController’s stairstep ground requirement mode
RigidBodyInteraction	Container class to hold information about KinematicCharacterController’s collision with another RigidBody

Quickstart

Expectations

This character controller provides a more classic approach to movement similar that of Valve's Half-Life or Croteams's Serious Sam, however replicating this in modern engines can be a bit of a pain as the built in controllers usually fall flat with tiny issues that lessen from the user experience, and full rigidbody controllers can be hard to execute properly. This plugin aims to provide a ready to go package to act as an alternative to the built in character controller found in Flax Engine, based on "Improved Collision Detection and Response" by Kasper Fauerby and "Improving the Numerical Robustness of Sphere Swept Collision Detection" by Jeff Linahan. To avoid re-inventing the wheel where possible (and to provide similarity to users) this plugin also pulls in inspiration from the free "Kinematic Character Controller" for Unity by Philippe St-Amand by mirroring the idea of using interfaces to control the KinematicMovers and KinematicCharacterControllers, along side pulling other ideas like using a callback to filter collisions.

You will need to provide your own logic for actually moving the character, this is simply just a "lower level" alternative to the built in character controller

This plugin overrides the RigidBody/Transform's Position and Orientation properties, KinematicBase's TransientPosition and TransientOrientation properties can be used in place of those. If you wish to forcibly set the position or forcibly set the orientation of the kinematic object (eg. teleporting), you need to use KinematicBase's functions "SetPosition" and "SetOrientation". Both KinematicMover and KinematicCharacterController extend from KinematicBase.

But how does it work?

After making your character, you need to make a script that implements the "IkinematicCharacterController" interface and assign the script to the controller's "Controller" property (this can be simply done by just assigning it on the script's OnEnable, check the KCCExample repository)

This interface will provide you with the necessary callbacks to alter the behavior during runtime.

The controller **MUST** have an uniform scale of 1, if you wish to to resize the character please update the relevant properties in the KinematicCharacterController.

Note: While the system does execute during a FixedUpdate tick and as such it usually does work with code that also relies on FixedUpdate, it has its own order of internal processing. If you need to somehow exactly step in time with the system outside of the callbacks, you may attach yourself to the exposed Actions in the KCC plugin.

KCC update loop

If “Auto Simulation” is enabled in the plugin settings, the following execution order will happen:

FixedUpdate →

1. PreSimulationUpdate
2. SimulationUpdate
3. PostSimulationUpdate

If “Auto simulation” is disabled, it is up to you as the developer to call the above functions manually.

During these the following Actions are called (if any attached) from the KCC plugin:

“PreSimulationUpdateEvent”: At the very beginning of the call, before InitialPosition/InitialOrientation are assigned from TransientPosition/TransientOrientation.

“SimulationUpdate”: At the very beginning of the call, before all movers and characters are processed.

“PostSimulationUpdate”: At the very end of the call, after necessary interpolation info is set up.

If “Interpolate” is enabled in the plugin settings, the plugin will automatically interpolate all KCC registered objects, allowing you to run the game at a lower physics tick rate. If disabled it is up to you as the developer to call the function “InterpolationUpdate” manually if interpolation is wanted.

The IKinematicCharacter interface:

Public void **KinematicMoveUpdate**(out Vector3 velocity, out Quaternion orientation)

Called when the simulation needs to know the velocity and orientation for a tick before sweeping movement, the character will attempt to move until the length of the velocity is more or less zero. You may transfer root motion to the system by extracting it from the animation and applying it here (please refer to Flax engine documentation to see how you may do this from the animation graph).

Public Vector3 **KinematicGroundProjection**(Vector3 velocity, Vector3 gravityEulerNormalized)

Called the the character velocity needs to be projected along side the current ground plane during the sweep, the velocity supplied here is the remaining velocity for the tick at the point where this callback is triggered. This is necessary if you wish to move up sloped surfaces without issues. Vector3’s ProjectOnPlane will suffice for modern use.

Tip: the KinematicCharacterController supplies the function “GroundTangent” to help with retro style projection where the ground normal does not affect any lateral speed.

Public bool **KinematicCollisionValid**(Collider other)

Called when the character collides with something during a sweep, this can be used to precisely filter out collisions (eg. teammates).

Public void **KinematicCollision**(RayCastHit hit)

Called when the character collides with something during a sweep, this may be useful if you need to have something external react to the collision as the final position of the controller's collider may not actually end up colliding with whatever it hit at the end of the sweep.

Public void **KinematicunstuckEvent**(Collider collider, Vector3 penetrationDirection, float penetrationDistance)

Called when the character unstucks itself during a sweep, this may be useful if you want to implement crushers for example.

Public void **KinematicGroundingEvent**(GroundState groundingState, RayCastHit? Hit)

Called when the character's ground state changes during a sweep, this may be useful if you wish to implement particle effects upon landing on ground

Public bool **KinematicCanAttachToRigidBody**(RigidBody rigidBody)

Called during the sweep to check if the character can attach to a rigidbody to move with it.

Public void **KinematicAttachedRigidBodyUpdate**(RigidBody rigidBody)

Called for every tick the character is attached to a rigidbody to move with, may be useful if you wish to rotate the camera along side the rigidbody it is attached to.

Public void **KinematicRigidBodyInteraction**(RigidBodyInteraction rbInteraction)

Called during the sweep when the character collides with a rigidbody, may be useful if you wish to have rigidbodies react interactively.

Public void **KinematicPostUpdate**()

Called after the character sweep is done