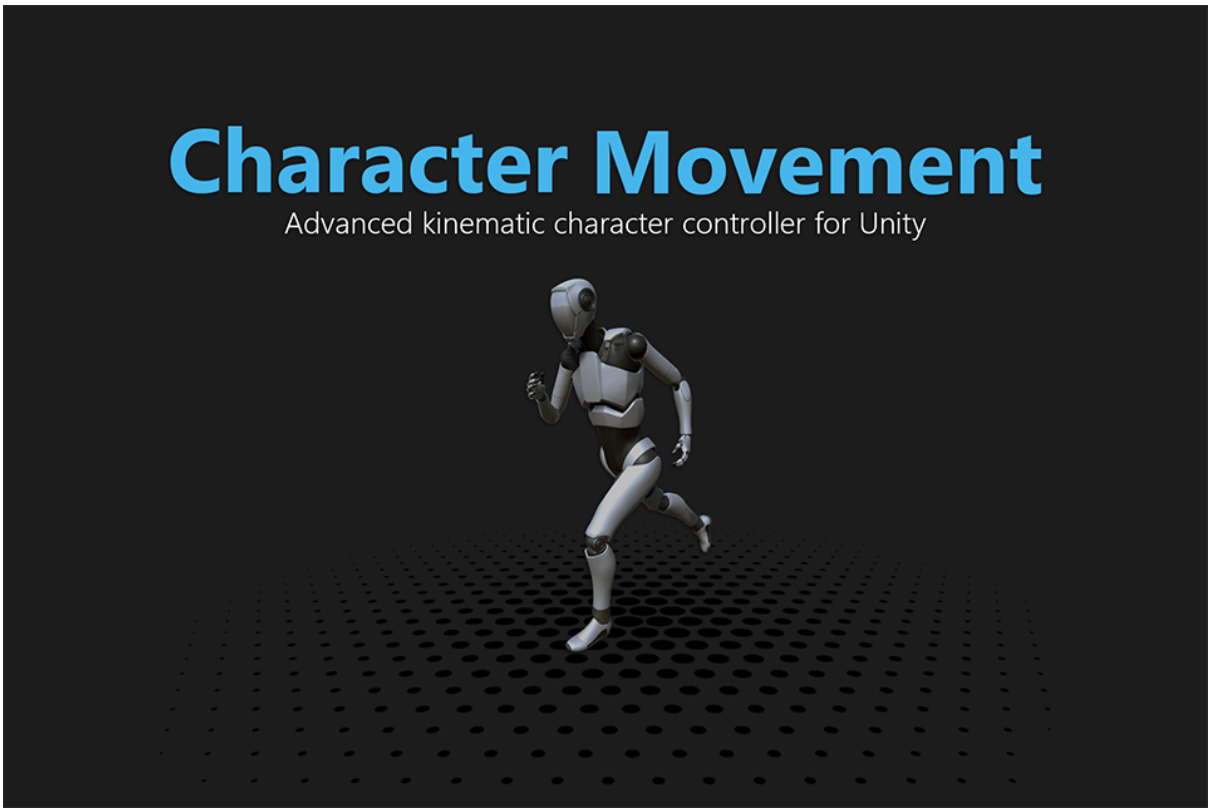
Character Movement

Advanced kinematic character controller for Unity

User Manual

Version 1.0.0



Important note on package dependencies

The `Character Movement` component does not require any external package in order to operate, however its Demo and the First/Third Person examples require the **Cinemachine** package to work.

Please make sure to install the **Cinemachine** package into your project when importing the demo and examples.

Preface

First of all I would like to thank you for purchasing this package, I sincerely appreciate your support and hope this helps you to make awesome games and projects!

If you have any comments, need some help or have a feature you would like to see added, Please don't hesitate to contact me at **ogracian@gmail.com** I'll be happy to help you.

Please include the **invoice number** you received as part of your purchase when requesting support via email. Thanks!

Kind Regards,
Oscar Gracian

The `CharacterMovement` component

The `CharacterMovement` component is a **robust**, **high-performance**, and **feature-rich** kinematic character controller.

It's been developed as an alternative to **Unity's Character Controller** offering the same workflow (*Move / SimpleMove* function) but with many features and advantages over it.

What is a character controller ?

A **Character Controller** allows you to easily do movement constrained by collisions without having to deal with a dynamic rigidbody.

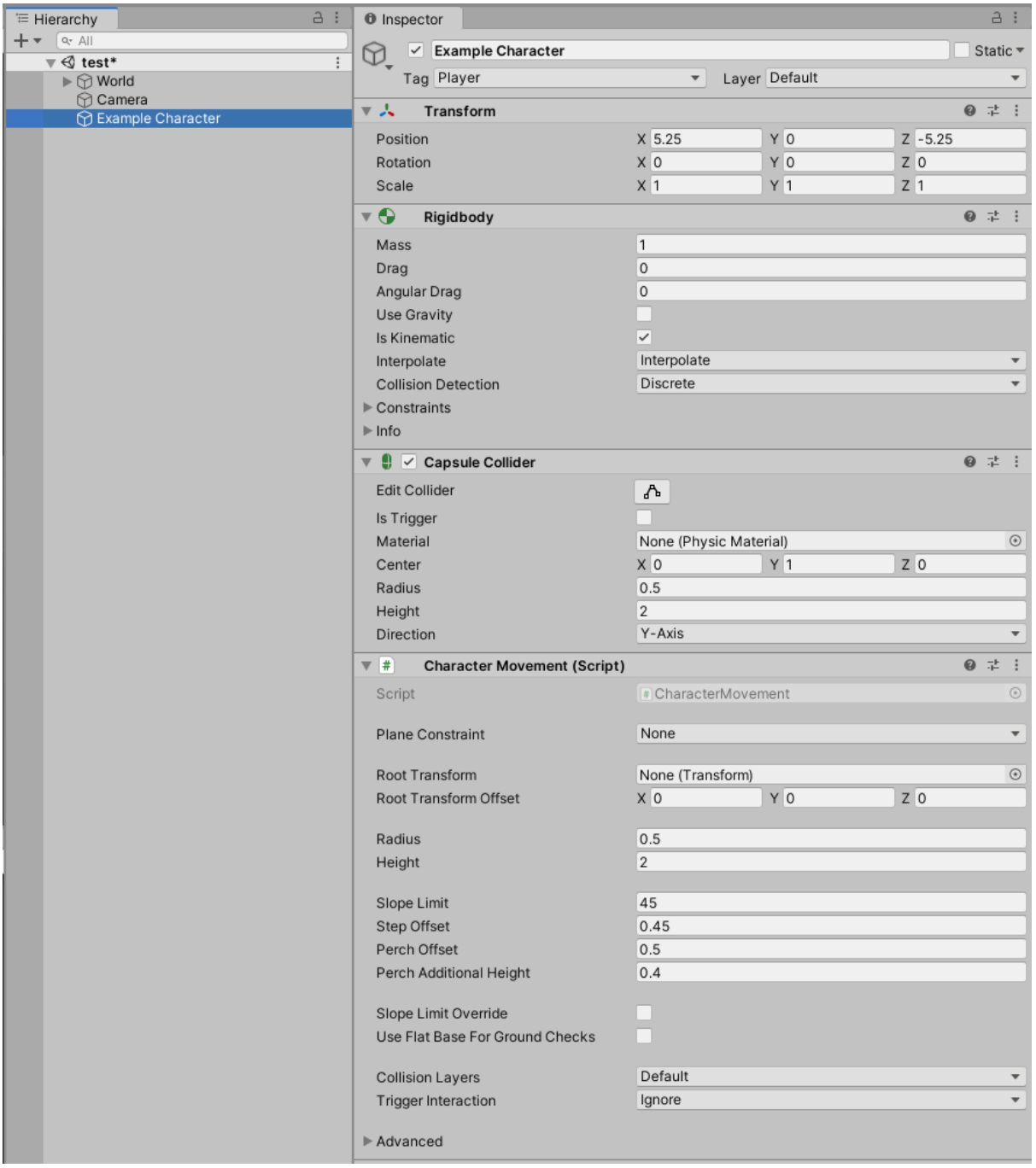
A **Character Controller** is not affected by forces and will only move when you call the *Move* function. It will then carry out the movement but be constrained by collisions.

In the past, games did not use a 'real' physics engine like the PhysX SDK (Unity's underlying physics engine). But they still used a character controller to move a player in a level. These games, such as **Quake** or even **Doom**, had a dedicated, customized piece of code to implement collision detection and response, which was often the only piece of physics in the whole game. It actually had little physics, but a lot of carefully tweaked values to provide a good feeling while controlling the player. The particular behavior it implemented is often called the '**collide and slide**' algorithm, and it has been 'tweaked for more than a decade'.

The `CharacterMovement` is an implementation of such an algorithm, providing a robust and well-known behavior for character control.

Using the `CharacterMovement` component

To use the `CharacterMovement` component, simply add it to a newly created `GameObject`, this will add the needed components for its correct operation.



Worth note by itself the `CharacterMovement` wont do anything, as it's expected you to call its *Move* (or *SimpleMove*) function to perform the desired movement.

Character Volume

The `CharacterMovement` component uses a capsule as its bounding volume, defined by a *vertical height* and a *radius*. The *height* is the total distance between the start and end of the capsule.

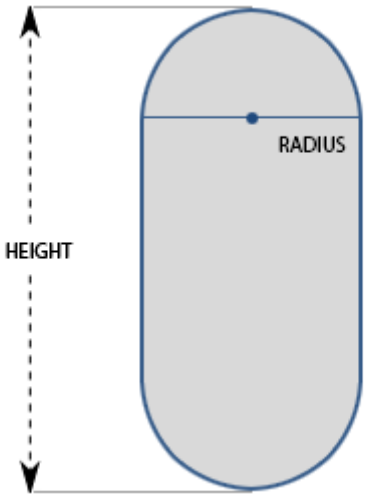


Fig 1. Character's Volume

A capsule (due its rounded shape) offers a better behavior to traverse the world geometry and its **the recommended choice**, however it can cause a situation where characters slowly lowers off the side of a ledge (as their capsule 'balances' on the edge), to overcome this, `CharacterMovement` offers the option to *Use Flat Base For Ground Checks*.

This, as its name suggests, lets the `CharacterMovement` perform the ground detection as if the character is using a shape with a flat base (*e.g.* a cylinder). This avoids the situation where characters slowly lower off the side of a ledge (as their capsule 'balances' on the edge).

A great feature to have when developing 2D / 2.5D side scrollers!

Volume Update

Sometimes it is useful to change the size of the character's volume at runtime. For example if the character is crouching, it might be required to reduce the height of its bounding volume so that it can then move to places he could not reach otherwise.

The related functions are:

```
public void SetDimensions(float characterRadius, float characterHeight);

public void SetHeight(float characterHeight);
```

Changing the size of a controller using the above functions does not change its *position*.

It is important to note that volumes are directly modified without any extra tests, and thus it might happen that the resulting volume overlaps some geometry nearby. For example when resizing the character to leave a crouch pose, i.e. when the size of the character is increased, it is important to first check that the character can indeed 'stand up': the volume of space above the character must be empty (collision free). It is recommended to use the overlap queries for this purpose:

```
public bool CheckHeight(float testHeight);

public int OverlapTest(Vector3 characterPosition, Quaternion
characterRotation, float testRadius, float testHeight, int layerMask,
Collider[] results, QueryTriggerInteraction queryTriggerInteraction);

public Collider[] OverlapTest(Vector3 characterPosition, Quaternion
characterRotation, float testRadius, float testHeight, int layerMask,
QueryTriggerInteraction queryTriggerInteraction, out int overlapCount);
```

Moving with `CharacterMovement` component

The heart of the `CharacterMovement` component is the *Move* function that actually moves characters around:

```
public Vector3 Move(Vector3 newVelocity);
```

Where *newVelocity* parameter, is the character's updated *velocity*, and returns the resultant *velocity*, *e.g*. if falling with a *velocity* of (0, -10, 0) and landed, the returned *velocity* will be (0, 0, 0) as the character has successfully landed on walkable ground.

```
public Vector3 Move();
```

Without any parameter and returns no value, internally this updates the character's velocity.

Independently of the *Move* function used, you can **read** and **write** the character's velocity using its *velocity* property.

Additionally to its *Move* function, the `CharacterMovement` component includes a *SimpleMove* function (just like **Unity's Character Controller**) to move your character:

```
public CollisionFlags SimpleMove(Vector3 desiredVelocity, float maxSpeed,
float acceleration, float deceleration, float friction, float
brakingFriction, Vector3 gravity = default, bool onlyHorizontal = true, float
deltaTime = 0.0f);
```

The *SimpleMove* is a higher level function implementing a highly configurable friction-based movement on top of the *Move* function and it takes care of accelerating / decelerating your character's *velocity* towards the given *desiredVelocity,* limit its *maxSpeed,* applying *friction*, limit *'wall'* contribution, etc.

Graphics Update

Each frame, after *Move* calls, graphics objects must be kept in sync with the new character positions. Character' positions can be accessed using the following properties and functions:

```
public Vector3 position;

public Vector3 worldCenter;

public Vector3 GetPosition();
```

This returns the position from the bottom of the collision shape, since this is what is used internally both within the **PhysX SDK** (e.g. *rigidbody.position*) and by usual graphics APIs (e.g. *transform.position*).

Internally and in order to ensure a proper world geometry navigation, a small *contact offset* is maintained around the character's volume, to avoid numerical issues that would otherwise happen when the character touches other shapes.

Alternative helper functions are provided to work using the character's bottom position, *a.k.a.* the foot position:

```
public Vector3 GetFootPosition();
```

The foot position takes the *contact offset* into account.

You can make use of the `Root Transform` property, when assigned, the `CharacterMovement` component will automatically update your model's position to use the character's foot position as shown on examples.

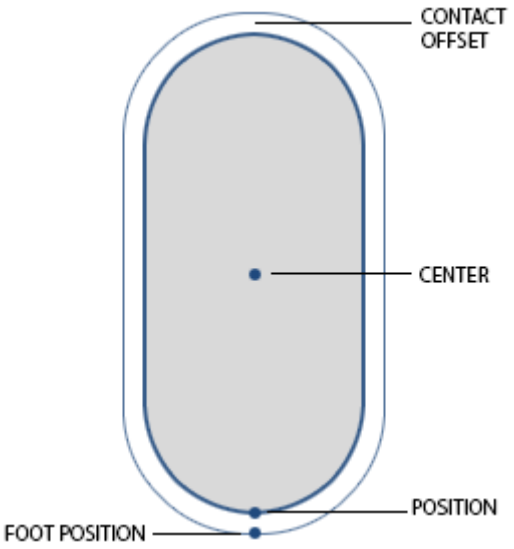


Fig 2. Position vs Foot Position

Ground Detection

An important feature in order to constrain the character's movement, this consists in analize the "ground" below the character's and deciding if it is standing on *walk-able* or *non walk-able* ground. Later, the character can use this information to update its *velocity* accordingly, play a different animation, etc .

The ground detection starts with a sweep along the character's down direction; if no hit, the function ends as no further tests are necessary, if a hit occurs the process will perform additional tests to determine if the collided "ground" is *walk-able* or not, at the end of this process, the current hit has been fully evaluated and its results can be queried through the `CharacterMovement` *currentGround* and additional related properties.

Walkable and Non-Walkable Ground

It is often desired to prevent walking on polygons whose slope is steep. The `CharacterMovement` can do this automatically thanks to a user-defined *slope limit*. All polygons whose slope is higher than the *slope limit* value will be marked as *non walk-able*, and will not let characters go there.

In order to correctly determine if a collided polygon is *walk-able* or not, it's important to retrieve the **real surface normal**.

The `CharacterMovement` is able to retrieve the **real surface normal** to accurately determine if the polygon slope is *walk-able* or not, it supports all Unity's primitive colliders (*Box*, *Sphere*, *Capsule*), *Terrain Collider* and *Mesh Colliders* (convex and non-convex).

Worth note, in the case of a *Mesh Collider* it must be read / write enabled and for convex *Mesh Collider* it falls back to a *raycast* method as unfortunately Unity does not expose the data needed to rebuild the surface normal.

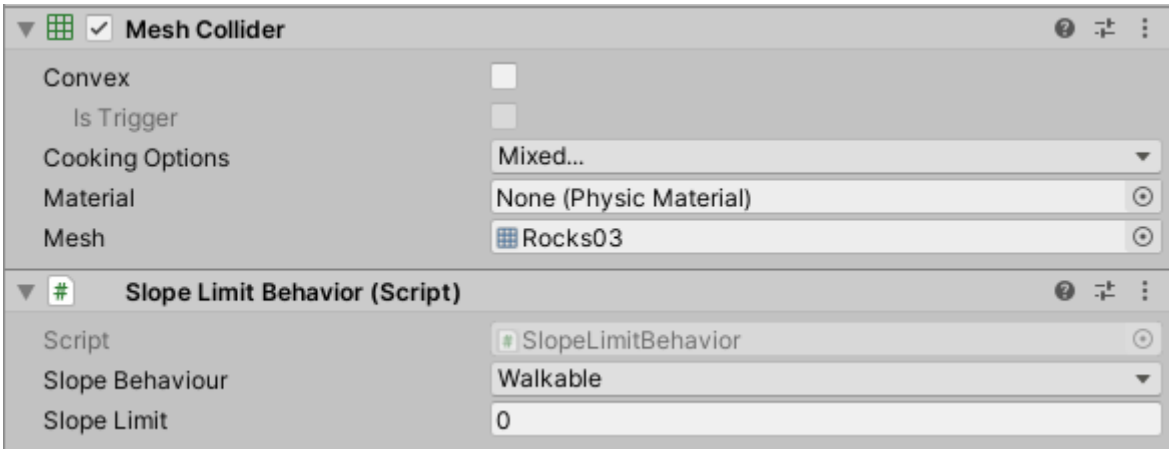
The `CharacterMovement` not only prevents the character from moving up *non walk-able* slopes but also forces it to slide down those slopes.

The slope limit is defined in *slopeLimit* property. The limit is expressed in degrees as the desired limit angle. For example, a slope limit of 45 degrees.

Using a *slopeLimit* = 90 degrees automatically disables the feature (i.e. characters can go everywhere).

Slope Limit Override

While the *slopeLimit* is more than enough to correctly restrict the character's movement, there are situations where you would like to restrict / allow the movement to a whole *collider* instead of *per-face*, you can make use of the `SlopeLimitBehavior` component.



The `SlopeLimitBehavior` component when added to a *Collider*, lets you override the *slopeLimit* for the whole *collider* rather than *per-face*.

For example, take the included rocks from the demo scenes, at plain sight it's expected the character to walk over them, however looking closer those small rocks have many tiny triangles which can easily pass than the defined *slopeLimit* value causing hiccups on character's movement, to overcome this situation simply add the `SlopeLimitBehavior` component to the rock's collider and set its *SlopeBehavior* to *Walkable*.

In a similar way, an easy method to force a character to slide off another character, is adding this `SlopeLimitBehavior` component to the NPC and setting its *SlopeBehavior* to *Not Walkable* now when a character (with its `SlopeLimitOverride` enabled) lands on this NPC it will slide off of it.

Perch Offset

This lets you refine the character's *walkable detection area* (e.g. the character's *feet* area) within the capsule's lower hemisphere and independently of the capsule *radius*. All hits on the capsule's bottom hemisphere but outside of this perch area will be considered as *non-walk-able.*

For example, a default character with a *radius* of 0.5 units; setting a *perch offset* value of 0.5 (same as radius) uses the whole capsule's bottom hemisphere as the character's *feet* area, by other hand setting a *perch offset* value of 0, will effectively reduce the character's *feet area* to a thin ray.

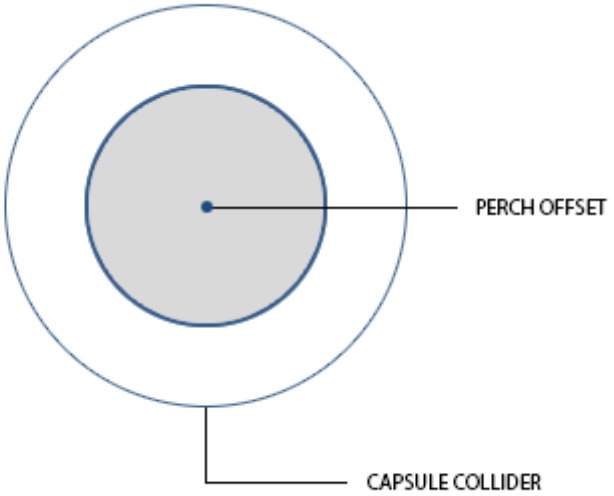


Fig 3. Reduced perch area (top view)

Ground Constraint

This is an important feature as it helps to keep the player 'glued' to *walk-able* surfaces and prevent it from climbing / accessing *non-walk-able* surfaces, bounce when walking down-hill, or even being launched off ramps when running at high speeds. However, due to this constraint, it is necessary to manually disable / pause it when a character is allowed to leave the ground, *e.g.* on a jump.

You can manage the ground constraint using the following properties and functions:

```
public bool isConstrainedToGround;

public bool constrainToGround;

public void PauseGroundConstraint(float unconstrainedTime = 0.1f);
```

This allows you to temporarily disable the ground constraint and automatically re-enable it when *unconstrainedTime* runs out.

Plane Constraint

Similar to the Ground Constraint, this limits the character so movement along the locked axis (or a custom plane) is not possible. A typical usage for this is to constrain the z-axis when implementing a 2D / 2.5D sidescroller.

Additionally this lets you specify an arbitrary plane (defined by its normal) to constrain the character's movement. A common usage for this is when a character is 'grabbing' a wall, or hugging a wall during a cover mechanic, etc.

The related properties and functions are:

```
public bool isConstrainedToPlane;

public void SetPlaneConstraint(PlaneConstraint constrainAxis, Vector3
planeNormal);
```

Stepping

This allows the character with a non-zero *step-offset* to go over obstacles up to the *step offset* value height.

While this tries to limit the maximum climbable height to the *step offset* value, and in most situations it does, however sometimes (even with a *step offset* of 0.0) capsules are able to go over small obstacles, since their rounded bottom produces an upward motion after colliding with a small obstacle.

Capsules with a non-zero *step offset* can go over obstacles higher than this, because of the combined effect of the auto-stepping feature and their rounded shape. In this case the largest altitude a capsule can climb over is difficult to predict, as it depends on the *step offset* value, the capsule's *radius*, and even the magnitude of the displacement vector.

`CharacterMovement` implements a constrained stepping algorithm, in which an attempt is made to make sure the capsule can not climb over obstacles higher than the *step offset*.

Up Vector

In order to implement the stepping feature, the `CharacterMovement` needs to know about the *'up'* vector. This up vector is defined by the capsule's up-axis and later available through the `transform.up` property.

The *up* vector does not need to be axis-aligned. It can be arbitrary, modified each frame, allowing the character to navigate on spherical worlds.

Modifying the *up* vector changes the way the `CharacterMovement` sees character volumes. For example a capsule is defined by a *height*, which is the *'vertical height'* along the *up* vector. Thus, changing the *up* vector effectively rotates the capsule from the point of view of the component. The modification happens immediately, without tests to validate that the

character does not overlap nearby geometry. It is then possible for the character to be penetrating some geometry right after the call. Using the overlap test is recommended to solve these issues.

While not mandatory, it is recommended to rotate your character before calling the *Move* function, so the *Move* function handles overlaps (if any) for you. This order is prefered and used in the included examples.

Collisions

`The CharacterMovement` component lets you know exactly what's happening during a *Move* call, at its simplest form you can access its *collisionFlags* property to know what part of the capsule collided with the environment during the last *Move* call, however when more control or further information is needed, you can subscribe to its *Collided* and *FoundGround* events or use its *GetCollisionCount* and *GetCollisionResults* functions to receive detailed information about any collision / overlap found during last *Move* call.

The *CollisionResult* structure is used to retrieve detailed information about the character's evolution. In particular, it is called when a character collides with something, be it a collider, another character, or a rigidbody.

When a character hits something, the *Collided* event is triggered. Various impact parameters are sent to the callback, and they can then be used to do various things like playing sounds, rendering trails, applying forces, and so on. Note that this will only be called during a *Move* call.

You can access the list of *CollisionResult* occurred during the last *Move* call using the functions:

```
public int GetCollisionCount();

public CollisionResult GetCollisionResult(int index);
```

In a similar way, the *FindGroundResult* structure is used to retrieve information about the ground the character is standing on.

You can access the character's current ground using:

```
public FindGroundResult currentGround;
```

Or the many ground related properties like: *isGrounded, groundPoint, groundNormal, groundSurfaceNormal*, etc.

A particularly important event is known when a character has landed, *e.g.* was on *non-walkable-ground* or air and found *walk-able* ground. You can make use of the *FoundGround* event or use helper properties to poll the character's ground status, *e.g*:

```
if (!characterMovement.wasOnGround && characterMovement.isGrounded)
    Debug.Log("Landed");
```

Lastly to access the character's terminal velocity when hitted the ground using the property:

```
public Vector3 landedVelocity;
```

Ignoring Colliders

The `CharacterMovement` component let you temporarily ignore collisions against specific colliders or even whole colliders attached to a particular rigidbody, using the following functions:

```
public void IgnoreCollision(Collider otherCollider, bool ignore = true);

public void IgnoreCollision(Rigidbody otherRigidbody, bool ignore = true);
```

Worth note this makes the `CharacterMovement` collision detection functions ignore the specified colliders, however in case of collisions against dynamic rigidbodies, the character's *capsule* can still collide against those, as this gives you the flexibility to decouple the movement collisions with physics collisions and vice-versa.

For example, you can make the capsule ignore all collisions against a particular rigidbody so the physics completely ignore this using Unity's collision matrix, however your character can still collide against those letting you handle the collision response.

You can disable / enable *capsule* collisions using the following function:

```
public void CapsuleIgnoreCollision(Collider otherCollider, bool ignore =
true);
```

Collision Callbacks

The `CharacterMovement` component has been developed with extensibility in mind, so you can fine tune it to match your game needs.

It offers a set of callback functions to determine how a character should proceed when a collision occurs, e.g. Let you filter / ignore other objects, define the desired `CollisionBehavior` or even determine the collision response impulses, all in a friendly and easy to implement way.

```
public delegate bool ColliderFilterCallback(Collider collider);

public delegate CollisionBehavior CollisionBehaviorCallback(Collider
collider);

public delegate void CollisionResponseCallback(ref CollisionResult
inCollisionResult, ref Vector3 characterImpulse, ref Vector3 otherImpulse);
```

The *ColliderFilterCallback* lets you determine if the collided character should be considered as an 'obstacle' or ignored by collision detection routines.

The *CollisionBehaviorCallback* it's a powerful mechanism to influence the character's behavior, or in other words how it should proceed when a collision occurs. Using a combination of bitwise flags you can fine tune the collision behavior overriding the default one in a performant and practical way.

The collision behavior flags are defined as follows:

```
public enum CollisionBehavior
{
	Default = 0,

	/// <summary>
	/// Determines if the character can walk on the other collider.
	/// </summary>

	Walkable = 1 << 0,
	NotWalkable = 1 << 1,

	/// <summary>
	/// Determines if the character can perch on the other collider.
	/// </summary>

	CanPerchOn = 1 << 2,
	CanNotPerchOn = 1 << 3,

	/// <summary>
	/// Defines if the character can step up onto the other collider.
	/// </summary>

	CanStepOn = 1 << 4,
	CanNotStepOn = 1 << 5,

	/// <summary>
	/// Defines if the character can effectively travel with the object it
is standing on.
	/// </summary>

	CanRideOn = 1 << 6,
	CanNotRideOn = 1 << 7
}
```

For example, making use of the *CollisionBehaviorCallback* function you can decide if the collided should be *walkable* or not independently of the character's *slopeLimit*, allow / prevent the *perch behavior*, determine if the character can step on and even determine if can use it as a moving platform or not.

Physics interactions

`The CharacterMovement` component (when enabled), is able to interact with other rigidbodies (push / be pushed) and other characters, apply forces, explosion forces, downward forces, etc.

When enabled, it will perform a basic collision response against dynamic rigidbodies and other characters. However it is often not a very convincing solution.

The bounding volumes around characters are artificial (*boxes*, *capsules*, *etc*) and invisible, so the forces computed by the physics engine between a bounding volume and its surrounding objects will not be realistic anyway. They will not properly model the interaction between an actual character and these objects. If the bounding volume is large compared to the visible character, maybe to make sure that its limbs never penetrate the static geometry around, the dynamic objects will start moving (pushed by a bounding volume) before the actual character touches them - making it look like the character is surrounded by some kind of force field.

Pushing effects are usually dictated by gameplay, and sometimes require extra code like inverse kinematic solvers, which are outside of the scope of the *CharacterMovement* module. Even for simple use cases, it is for example difficult to push a dynamic box forward with a capsule controller: since the capsule never hits the box exactly in the middle, applied force tends to rotate the box - even if gameplay dictates that it should move in a straight line.

Thus, this is an area where the *CharacterMovement* component should best be coupled to specific game code, to implement a specific solution for a specific game. This coupling can be done in many different ways. For simple use cases it is enough to use the *Collided* callback to apply artificial forces to surrounding dynamic objects or the *CollisionResponseCallback f*or a more complete solution.

Character Interactions

The interactions between characters (i.e. between two *CharacterMovement* objects) are limited (basic push), since in this case both objects are effectively kinematic objects. In other words their motion should be fully controlled by users, and neither the PhysX SDK nor the *CharacterMovement* module should be allowed to move them.

The *Callback* functions are used to define basic interactions between characters (or any other collider). It's *ColliderFilterCallback* function can be used to determine if two characters should collide at all with each other:

To make characters always collide-and-slide against each other, simply return *true*.

To make characters always move freely through each other, simply return *false*.

Otherwise, customized and maybe gameplay-driven filtering rules can be implemented in this callback. Sometimes the filtering changes at runtime, and two characters might be allowed to go through each other only for a limited amount of time, etc.

Additionally you can make use of the *CharacterMovement* functions, to externally influence a character, such as:

```
public void AddForce(Vector3 force, ForceMode forceMode = ForceMode.Force);

public void AddExplosionForce(float forceMagnitude, Vector3 origin, float
explosionRadius, ForceMode forceMode = ForceMode.Force);

public void LaunchCharacter(Vector3 launchVelocity, bool
overrideVerticalVelocity = false, bool overrideLateralVelocity = false);
```

Moving Platforms

`The CharacterMovement` component is able to transparently (without further steps needed by you) handle moving platforms, being dynamic rigidbodies, scripted, animated, etc. The only requirement to keep the character on the moving platform is it to be *walkable*.

For example, you can jump on a fully dynamic vehicle and the character will use it as a moving platform without you having to do any further tasks or write custom scripts for that.

Additionally you can 'parent' the character to a desired moving platform so it moves when the assigned platform moves even if the character is not standing or touching it.

Extras

The *CharacterMovement* exposes the same collision detection methods it uses internally so you can take advantage of those when needed, i.e. such as implementing custom character mechanics (vaulting, mantle, wall-grabb, etc).

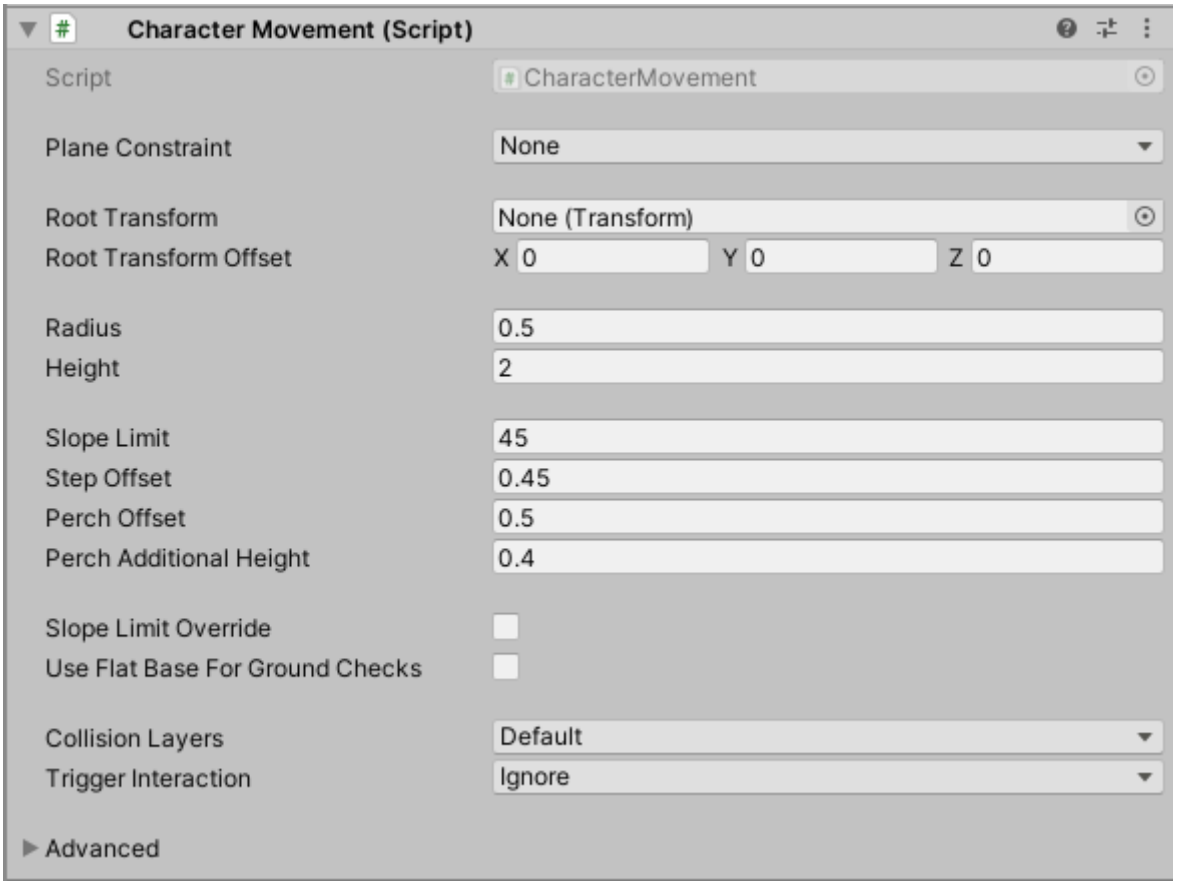
```
bool Raycast(Vector3 origin, Vector3 direction, float distance, int
layerMask, out RaycastHit hitResult, float thickness = 0.0f);

bool MovementSweepTest(Vector3 characterPosition, Vector3 sweepDirection,
float sweepDistance, out CollisionResult collisionResult);

void ComputeGroundDistance(Vector3 characterPosition, float sweepRadius,
float sweepDistance, float castDistance, out FindGroundResult
outGroundResult);

void FindGround(Vector3 characterPosition, out FindGroundResult
outGroundResult);
```

The `CharacterMovement` component



Plane Constraint

Constrain the character so movement along the locked axis is not possible. This is particularly useful when developing a 2D game, so you can prevent the character's movement on the z-axis.

Root Transform

The root transform in the avatar (e.g. your model).

Root Transform Offset

The root transform will be positioned at this offset from foot position.

Capsule Radius

The Character's capsule collider radius. This automatically configures the `CapsuleCollider` for you.

Capsule Height

The Character's capsule collider height. This automatically configures the `CapsuleCollider` for you.

Slope Limit

The maximum angle (in degrees) for a walkable slope.

Step Offset

The maximum height (in meters) for a valid step.

Perch Offset

Allow a character to perch on the edge of a surface if the horizontal distance from the character's position to the edge is closer than this. A character will not fall off if they are within `stepOffset` of a walkable surface below.

Perch Additional Height

When perching on a ledge, add this additional distance to `stepOffset` when determining how high above a walkable floor we can perch. Note that we still enforce `stepOffset` to start the step up, this just allows the character to hang off the edge or step slightly higher off the ground.

Slope Limit Override

If enabled, colliders with the `SlopeLimitBehavior` component will be able to override this `slopeLimit` property value.

Use Flat Base for Ground Checks

If enabled, it performs ground checks as if the character is using a shape with a flat base. This avoids the situation where characters slowly lower off the side of a ledge (as their capsule 'balances' on the edge).

Collision Layers

Character collision layers mask. If desired, you can use the context menu option to automatically configure this from the Physics collision matrix.

Trigger Interaction

Overrides the global `Physics.queriesHitTriggers` to specify whether queries (*raycasts*, *spherecasts*, *overlap tests*, etc.) should hit Triggers by default.

Use **Ignore** for queries to ignore trigger colliders.