PBC, Physically Based Character

There is no going back

Patrik Tegelberg, 2015-dec-15

I think one of my customers nailed it in this quote:

> *As an animator, it's my job to bring characters to life, and it's amazing how much physical based interaction adds to that. Your PBC system demo is absolutely wonderful, and after reading the docs, I realized it pretty much has everything I could hope to learn, and looks like great fun to work with.*

What makes this active ragdoll stand out is how it behaves when it is not falling, where it simultaneously is both soft and strong. Lesser system can be made soft, but not without totally losing the ability to accurately mimic the animation.

This asset relieves the pressure of needing a fantastically complicated animation code in order to have the character move naturally, or interact with the environment. At the same time you can use mecanim animations in almost the same way you normally do.

Physically interacting characters are no longer prohibitively expensive and will soon be everywhere. Additionally this system can seamlessly and automatically suspend the ragdoll when it is not needed, leaving the character no more costly than a regular character.

Table of Contents

History

A few years back I decided to play some counter strike. I died too much, and figured I did not have the time to get good playing, I needed something with more intense training. I made a small game with no story but lots of shooting. It bugged me that the enemies did not react believable (or at all) to non lethal hits, not even in the AAA games. I have not played counter strike since, I did this instead.

Feature overview

It turns out that to get realistic physics, one feature often depends on another. That means you have to develop several to get one. On the other hand, if you have built a few features, then you often get a few more for cheap. The features are, as much as possible, implemented as continuous physical mathematics, with higher fidelity than non-physical state machines. To be high performing, care have been taken to make the system able to run at the Unity standard fixed delta time of 0.02 seconds.

Soft character

The character can be pushed or dragged by simply adding a force to any limb. You could literary take it by its hand (or ear) and lead it away. It will also push on objects. The pushing force, or the force needed to drag the character, depends on the friction. There are several ways to adjust the character stiffness, in pose, position and rotation. The systems ability to follow fast paced animations and still be soft makes it an excellent choice for any application.

Knock back

The character has a will, it desires to do your bidding. It wants to follow the animations and root motions, but is limited by being about about human strength. Thus if it is hit by a object, or force, it will be knocked back in accordance to the physical law of preservation of momentum. This is true for the character as a whole and for individual limbs.

Ragdoll and back again

If the character is hit by a severe enough impact a fall is triggered. The character then ragdolls and falls, but it does not go completely ragdoll immediately. The characters ability to follow the current animation fades, making the fall look more life like. It is optional to switch animation during a fall, and have the ragdoll try to follow a custom fall animation. After the fall the character will regain strength and smoothly rise to continue following your animations (there is an option to have it stay fallen).

Newtons first law

To any force there is an equal and opposite force. Any motion of the physically based character is accomplished by adding forces to it. An equal and opposite force (and toque) is automatically applied to the object the character is standing on. If you jump on a car, the car will move. If you jump onto a trolley, you and the trolley will roll away. If you stand on a boat, the boat will tilt and sink a little. The object you stand on must be physical for this to be true, but other than that no coding is required.

Compatible with custom IK

There is a script called CustomUserIK. Whatever changes you make to the character's pose in that script will be the pose that the ragdoll tries to mimic.

Precision foot friction

A good foot friction model is central to physically based motion. Friction determines how far you will be knocked back and how sharp you can turn. If the friction is non-physical, other parts of the system must be non-physical too. The implemented friction model accurately takes into account both gravity, slopes and acceleration forces. You can turn sharper if you have a berm.

Balance tilt

You lean inwards when you turn and forward when you accelerate. Good animators accomplish this with AI and many animations. This system does not require special animations to achieve that effect, it will modify the animations on the fly to tilt the character to the physically appropriate angle. This is true both when running and when standing still. If you make for example the hand's rigidbody heavy the character will tilt away from that hand to keep the center of mass directly above the transform. Imagine how you look when you carry one heavy grocery bag. There is also a fancy feature that gives you the External force and torque acting on the character, no coding required, no matter where the force came from. This could be used to build a very advanced balance implementation. You can view the force by clicking ShowExternalForce in the inspector.

Natural ground follow

When you run over undulated terrain the distance from the ground to your center of gravity will vary to allow you to move in the most energy efficient way. A character not moving in an energy efficient way does not look natural.

Soft legs

Legs are not infinitely strong, hence they bend when your up/down velocity changes, e.g. when you land, when you run and the incline changes, or when the platform on which you stand accelerates vertically.

Predictive foot IK

Making a simple foot IK is easy. Making a foot IK that looks natural in any situation a game character can encounter is very complex. The included (optional) foot IK tries to predict where the next step is going to be planted and moves the foot there on a natural path, as if the character had vision. It also adjusts the step length on slopes. I do not recommend trying to implement your own foot IK. The foot IK is good enough to be hard to beat.

Moving platforms

The character automatically follows any platform it stands on. The velocity change is not instant, it is physical and depends on the friction. If the platform is rotation or tilting and the friction is not sufficient the character will glide. Since the character both follows and moves the platform it stands on, effects like jumping on a trolley and roll away with preserved momentum requires no extra programming. This may not work work stably for very light platforms (less than about one character mass). Heavy or kinetic platforms work well.

Independent character gravity

The characters gravity is not governed by PhysX, but is independent. This allows for some creative game play, like spherical worlds, ceiling walks, or infinite wall runs.

Motion controller

It is possible to use this system with any animator (legacy animations are hard). The included animator solves a few common obstacles for natural looking motion.

- Few transitions, most animations are blended to a continuous spectrum.
- Jump height is controllable. A tap on the space bar will result in a smaller jump than a full press will.
- When friction sliding, a parameter can adjust the blend for, if the feet are to move with the speed of the animation (gives cartoony look), or follow the ground's relative speed (more realistic look).
- Parameters in the inspector to set animation speed, and to scale the step length.

Synced foot sound

If the foot was moving, in the air, and is now on the ground, not moving, a foot step sound is played. Volume is linear with character velocity relative to the ground or platform.

Character/ragdoll automatic set-up

In the automatic set-up the ragdoll is automatically created to fit your model, the scripts are added and the variables are assigned. Read the setup instructions below.

Partial ragdoll pro

Since the strength of the ragdoll can be set in real time for each individual limb, partial ragdolls are easy. Complete partial ragdolls look unnatural. This system lets you go partially, partial ragdoll, and fade in and out of any blend between animation and ragdoll.

NavMesh ready

To integrate this with NavMesh or similar AI I recommend having the NavMesh move a “rabbit” transform, and using the FollowRabbit feature to to have the character follow the rabbit.

NPC characters / Custom animations

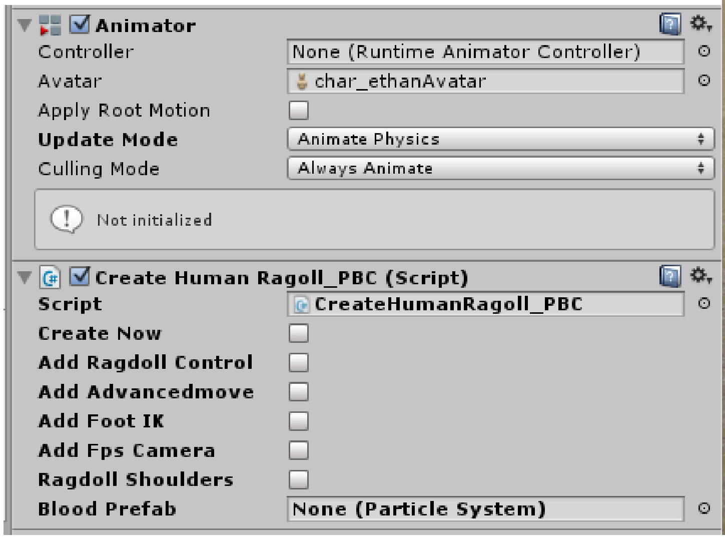
The character will animate just like a regular character. It will follow the pose and root motion without any additional coding. The NPC character will still have all the active ragdoll features.

No MMO button

There is no MMO button implemented. To have a complete game you may have to do work.

Setup humanoid

- ✗ Put your character in the scene.



Animator
Controller: None (Runtime Animator Controller)
Avatar: char_ethanAvatar
Apply Root Motion: ☐
Update Mode: Animate Physics
Culling Mode: Always Animate
Not initialized

Create Human Ragoll_PBC (Script)
Script: CreateHumanRagoll_PBC
Create Now: ☐
Add Ragdoll Control: ☐
Add Advancedmove: ☐
Add Foot IK: ☐
Add Fps Camera: ☐
Ragdoll Shoulders: ☐
Blood Prefab: None (Particle System)

Illustration 2: The inspector of the setup script.

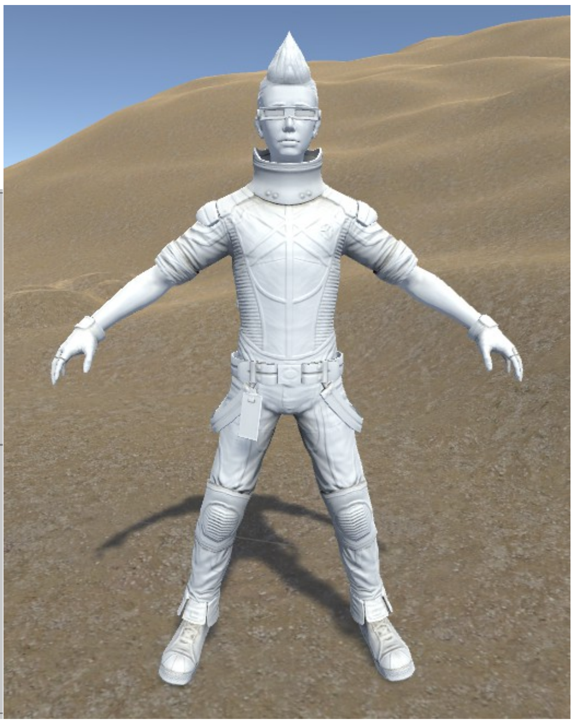


Illustration 1: Good setup pose.

- ✗ Make it strike a pose similar to the one shown in Illustration 1, with the legs slightly spread and the arms slightly dropped. The reason this is a good pose is because this will be the center of the limbs' rotational limits (you can adjust the limits, but there will be no need).
- ✗ Drag and drop the **CreatePBCCharacter** script, Illustration 2, onto the character. Assign a valid avatar. Choose an option, and click createNow. If you do not assign a custom blood particle system, the default will be used.
- ✗ Check that there are no warnings in the console. Check that the colliders have appropriate size. Adjust the position of the heel and toe2 transforms.
- ✗ Replace the InputManager file, in Folder ProjectSettings, with the included InputManager.

Optionally

- ✗ In the Camera3rd inspector, assign your character's hips to the lookAtTransform.
- ✗ Assign any ragdoll transform to the cannon ball's hit transform field.
- ✗ Download this script: http://pastebin.com/j3DWqe3R
  and drop it on the directional light in the scene. Change the GetMouseButton(0) to GetMouseButton(2) in the script, and change the spring from 50 to about 2000.

I recommend not to to choose the ragdollShoulders option. It will work but the PhysX chain becomes unnecessary long and can possibly get out of shape. If possible use animations where the shoulders are not animated and where the shoulders is set to none in the avatar configuration. Save a copy of your character before you change anything in the avatar configurer.

Setup generic

For generic animation's only the basic move script is supported. For humanoids I have taken advantage of that it is known, how a humanoid is configured, and how we want a humanoid to act. A generic animation can have any form, and thus no assumptions can be safely made. Generic rigs will only be as polished as you make them.

- ✗ Put your master character in the scene.
- ✗ Make it strike a pose that you want to be the center of the joint's rotation limits.
- ✗ Child the master character to an empty game object. Zero the transform position and rotation.
- ✗ Make another empty, named "ragdoll", and child it to the first empty.
- ✗ Make a copy of the master. Drop the root bone of the copy on the "ragdoll" empty (first child of the "ragdoll" empty should be the ragdoll root bone).
- ✗ Delete bones from the ragdoll until only the bones you wish to use as the ragdoll remains.
- ✗ Drag and drop the **CreateBOnlyGeneric** script, onto the **master** character, click createNow.
- ✗ The CreateBOnlyGeneric script will now check that it is childed, and have a sibling with a name containing "ragdoll". If it does it will make a ragdoll from the "ragdoll" hierarchy and add the basic scripts.
- ✗ Check that there are no warnings in the console. Check that the colliders have appropriate size / shape / position , that the rigidbodies have appropriate masses (larger masses / colliders in the body, smaller in the limbs is a good setup).
- ✗ Set the joint limits.
- ✗ Replace the InputManager file, in Folder ProjectSettings, with the included InputManager.

  Optionally

- ✗ In the Camera3rd inspector, assign your character's hips to the lookAtTransform.
- ✗ Assign any ragdoll transform to the cannon ball's hit transform field.

NOTE! To make generic animations work well I recommend you use only as few transforms as is needed. The more transforms the ragdoll has the the more tuning it will require to behave smooth and stable. Fewer colliders is also better performance.

Make certain that the colliders do not interfere during normal operation. Colliders that are jointed together may overlap because their collisions are disabled.

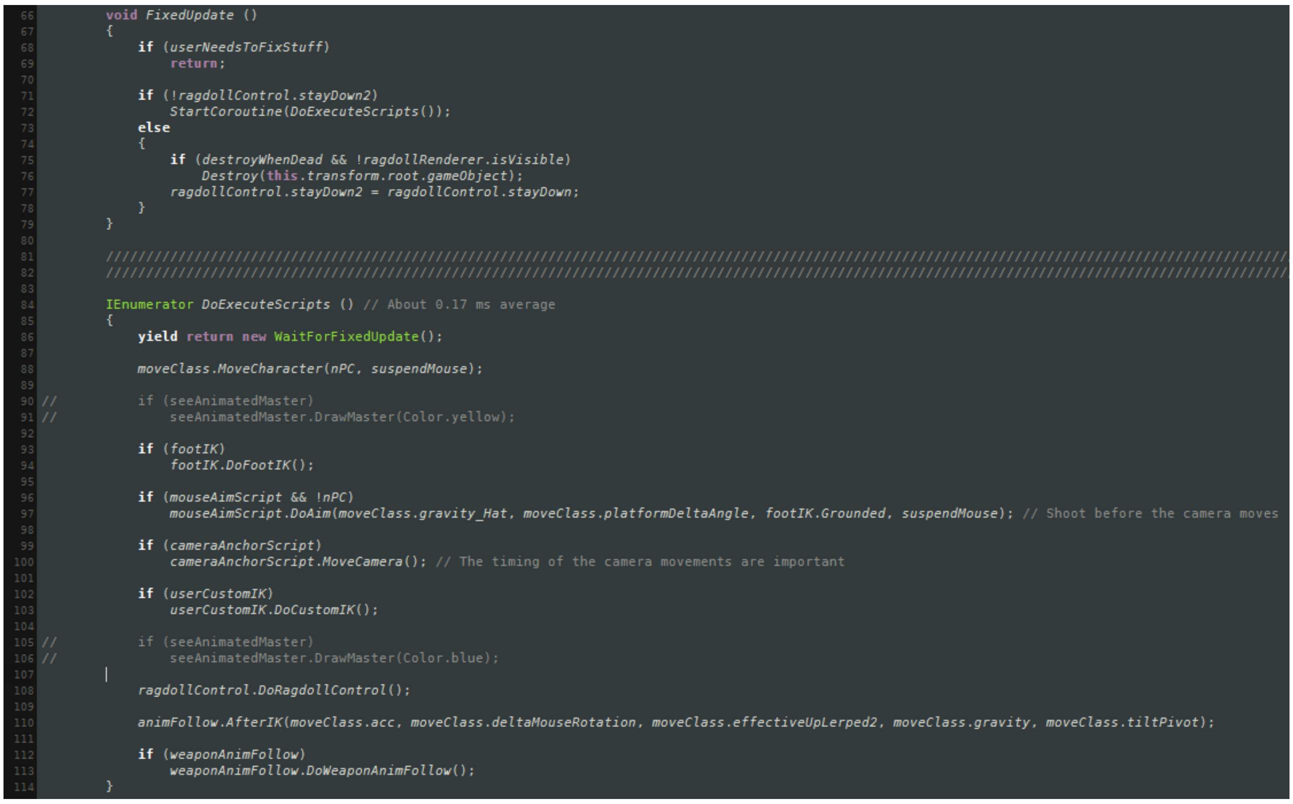
Scripts overview

This package is fairly comprehensive. It is expected that many users will want to modify this to get their own functionality. In this section the broad strokes (and some details) of the scripts are explained, in order to provide the user with the knowledge needed to expand this into his/her unique game.

Script Main

This script, shown in Illustration 3, controls the order in which the systems are executed. The Main is run with the frequency of FixedUpdate to be in sync with the physics engine. To be high performing, care have been taken to make the system able to run at the Unity standard fixed delta time of 0.02 seconds.

The commented out lines, in Main, that says SeeAnimatedMaster. DrawMaster can be uncommented to let you see a stick figure of the master at that point in the execution. This allows you to simultaneously see what the master's and the ragdoll's poses are. Under normal operation only the ragdoll is shown. There is also an option in the script AnimFollow to instead see only the master. These options are off course there as debugging tools if the character exhibits unexpected behavior.

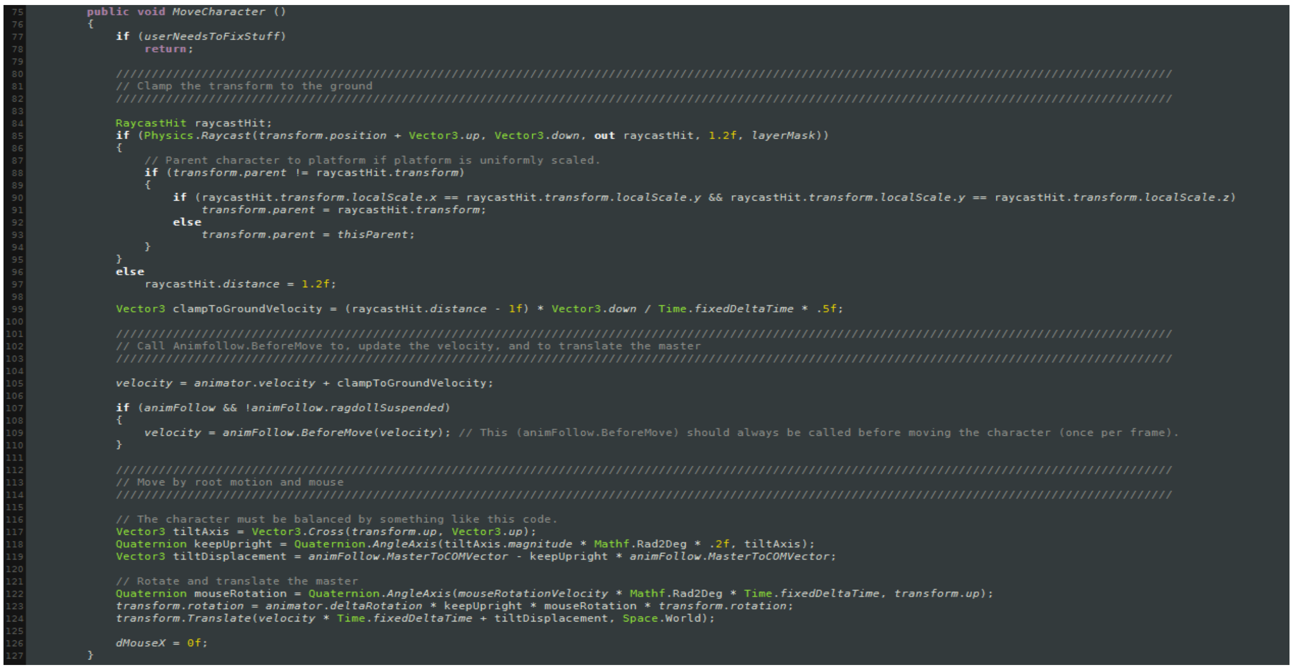


```
	void FixedUpdate ()
	{
		if (userNeedsToFixStuff)
			return;

		if (!ragdollControl.stayDown2)
			StartCoroutine(DoExecuteScripts());
		else
		{
			if (destroyWhenDead && !ragdollRenderer.isVisible)
				Destroy(this.transform.root.gameObject);
			ragdollControl.stayDown2 = ragdollControl.stayDown;
		}
	}

	//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
	//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

	IEnumerator DoExecuteScripts () // About 0.17 ms average
	{
		yield return new WaitForFixedUpdate();

		moveClass.MoveCharacter(nPC, suspendMouse);

//		if (seeAnimatedMaster)
//			seeAnimatedMaster.DrawMaster(Color.yellow);

		if (footIK)
			footIK.DoFootIK();

		if (mouseAimScript && !nPC)
			mouseAimScript.DoAim(moveClass.gravity_Hat, moveClass.platformDeltaAngle, footIK.Grounded, suspendMouse); // Shoot before the camera moves

		if (cameraAnchorScript)
			cameraAnchorScript.MoveCamera(); // The timing of the camera movements are important

		if (userCustomIK)
			userCustomIK.DoCustomIK();

//		if (seeAnimatedMaster)
//			seeAnimatedMaster.DrawMaster(Color.blue);

		ragdollControl.DoRagdollControl();

		animFollow.AfterIK(moveClass.acc, moveClass.deltaMouseRotation, moveClass.effectiveUpLerped2, moveClass.gravity, moveClass.tiltPivot);

		if (weaponAnimFollow)
			weaponAnimFollow.DoWeaponAnimFollow();
	}
```

Illustration 3: The Main script, determines the execution order.

MoveScript of the BasicOnlyCharacter

This asset is complicated. Many things must happen in sync to get the desired result. As a service to anyone feeling overwhelmed I provide a BasicOnly character with a simple MoveScript, Illustration 4, that is an example of what must happen to have any kind of working system. Maybe you don't need all the bells and whistles of a full PBC character, or you enjoy building your own code. This may be the place to start for you.

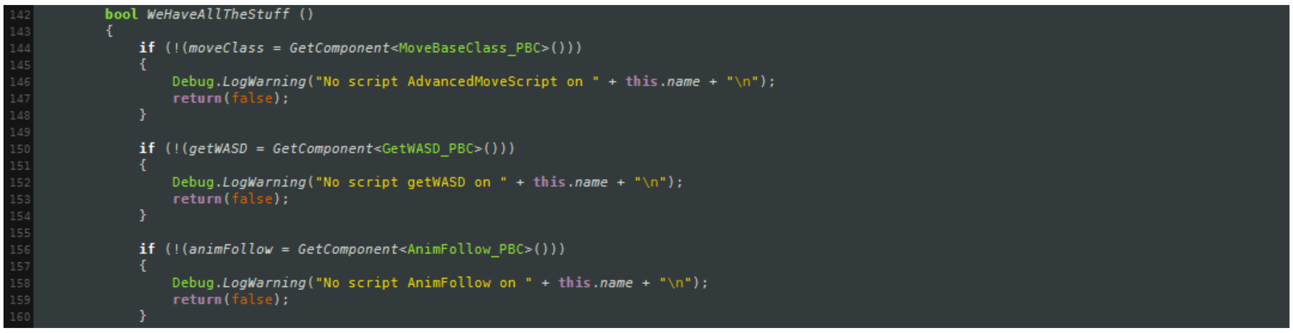


```
        public void MoveCharacter ()
        {
            if (userNeedsToFixStuff)
                return;

            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            // Clamp the transform to the ground
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

            RaycastHit raycastHit;
            if (Physics.Raycast(transform.position + Vector3.up, Vector3.down, out raycastHit, 1.2f, layerMask))
            {
                // Parent character to platform if platform is uniformly scaled.
                if (transform.parent != raycastHit.transform)
                {
                    if (raycastHit.transform.localScale.x == raycastHit.transform.localScale.y && raycastHit.transform.localScale.y == raycastHit.transform.localScale.z)
                        transform.parent = raycastHit.transform;
                    else
                        transform.parent = thisParent;
                }
            }
            else
                raycastHit.distance = 1.2f;

            Vector3 clampToGroundVelocity = (raycastHit.distance - 1f) * Vector3.down / Time.fixedDeltaTime * .5f;

            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            // Call Animfollow.BeforeMove to, update the velocity, and to translate the master
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

            velocity = animator.velocity + clampToGroundVelocity;

            if (animFollow && !animFollow.ragdollSuspended)
            {
                velocity = animFollow.BeforeMove(velocity); // This (animFollow.BeforeMove) should always be called before moving the character (once per frame).
            }

            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            // Move by root motion and mouse
            ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

            // The character must be balanced by something like this code.
            Vector3 tiltAxis = Vector3.Cross(transform.up, Vector3.up);
            Quaternion keepUpright = Quaternion.AngleAxis(tiltAxis.magnitude * Mathf.Rad2Deg * .2f, tiltAxis);
            Vector3 tiltDisplacement = animFollow.MasterToCOMVector - keepUpright * animFollow.MasterToCOMVector;

            // Rotate and translate the master
            Quaternion mouseRotation = Quaternion.AngleAxis(mouseRotationVelocity * Mathf.Rad2Deg * Time.fixedDeltaTime, transform.up);
            transform.rotation = animator.deltaRotation * keepUpright * mouseRotation * transform.rotation;
            transform.Translate(velocity * Time.fixedDeltaTime + tiltDisplacement, Space.World);

            dMouseX = 0f;
        }
```

Illustration 4: The simple MoveScripts is an example of the minimum of things that must happen to have any kind of working system.

Method WeHaveAllTheStuff

In several scripts you will see a bool called userNeedsToFixStuff that is set by a method called WeHaveAllTheStuff. The WeHaveAllTheStuff method assigns the variables that needs assigning and checks that the other dependencies are ready. This is a helper function to give the user tips and warnings about things that may cause the package to not execute properly. With the advent of the automatic set-up the need for these checks are greatly diminished, but for the time being they persist. This does not affect performance but can, if you want to, be removed (lines that do assigning must persist). WeHaveAllTheStuff consists mostly of things like the code shown in Illustration 5.

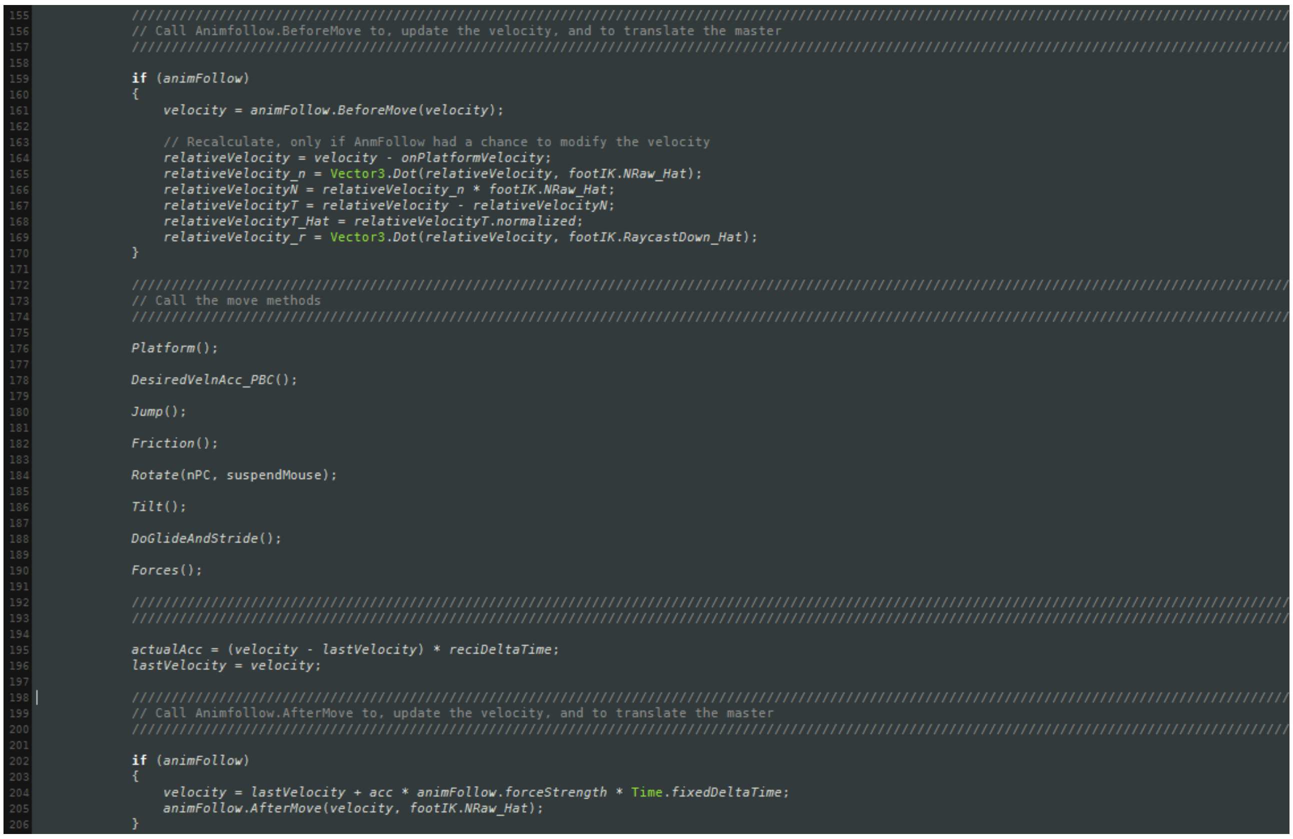


```
        bool WeHaveAllTheStuff ()
        {
            if (!(moveClass = GetComponent<MoveBaseClass_PBC>()))
            {
                Debug.LogWarning("No script AdvancedMoveScript on " + this.name + "\n");
                return(false);
            }

            if (!(getWASD = GetComponent<GetWASD_PBC>()))
            {
                Debug.LogWarning("No script getWASD on " + this.name + "\n");
                return(false);
            }

            if (!(animFollow = GetComponent<AnimFollow_PBC>()))
            {
                Debug.LogWarning("No script AnimFollow on " + this.name + "\n");
                return(false);
            }
```

Illustration 5: The method WeHaveAllTheStuff tries to help the user find setup errors.

Script MoveClass

The first method called by Main is MoveCharacter in the AdvancedMoveScript Illustration 6. This method does in turn manage the execution order of the methods handling the motion of the character. Notice that the AdvancedMoveScript calls a method in AnimFollow before it does any moving of the transform. That call must be made like this for the system to work.



```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Call Animfollow.BeforeMove to, update the velocity, and to translate the master
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

if (animFollow)
{
    velocity = animFollow.BeforeMove(velocity);

    // Recalculate, only if AnmFollow had a chance to modify the velocity
    relativeVelocity = velocity - onPlatformVelocity;
    relativeVelocity_n = Vector3.Dot(relativeVelocity, footIK.NRaw_Hat);
    relativeVelocityN = relativeVelocity_n * footIK.NRaw_Hat;
    relativeVelocityT = relativeVelocity - relativeVelocityN;
    relativeVelocityT_Hat = relativeVelocityT.normalized;
    relativeVelocity_r = Vector3.Dot(relativeVelocity, footIK.RaycastDown_Hat);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Call the move methods
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Platform();

DesiredVelnAcc_PBC();

Jump();

Friction();

Rotate(nPC, suspendMouse);

Tilt();

DoGlideAndStride();

Forces();

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

actualAcc = (velocity - lastVelocity) * reciDeltaTime;
lastVelocity = velocity;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Call Animfollow.AfterMove to, update the velocity, and to translate the master
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

if (animFollow)
{
    velocity = lastVelocity + acc * animFollow.forceStrength * Time.fixedDeltaTime;
    animFollow.AfterMove(velocity, footIK.NRaw_Hat);
}
```

Illustration 6: It is important that the method AnimFollow.BeforeMove gets called.

Physics work by accurately applying forces to acquire motion. The desired result of a motion is a position or a velocity. Crucial to quality motion is the purity of the signals. If the signals is noisy, filtering is needed and filtering introduces lag. In going from delta position to velocity, noise is introduced and in going to acceleration more noise is introduced. This system is therefore built with acceleration as the primary control signal.

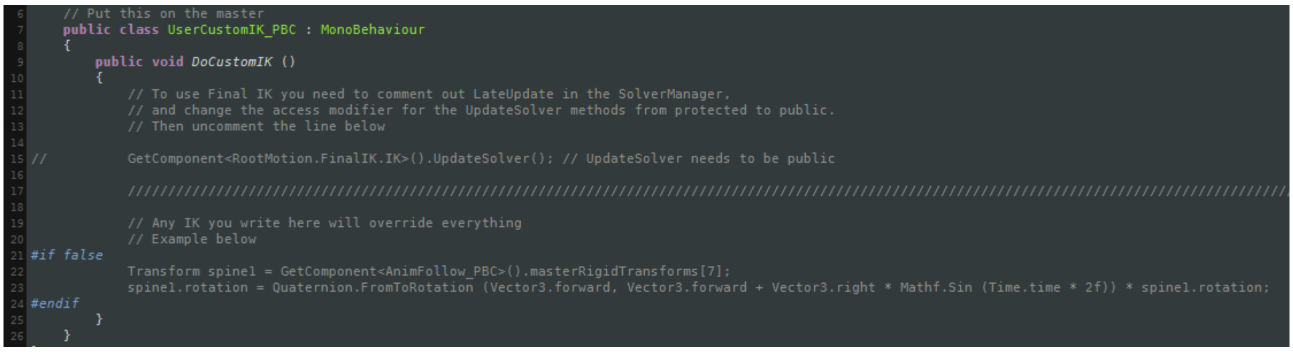
Script AdvancedFootIK

The second method to be called by Main is DoFootIK. DoFootIK manage the execution order of the methods involved in making the feet move naturally. The script communicate with the other systems via an abstract class. If a foot IK solution is not present, then a bare minimum of information is provided by a class called NoFootIK to keep the characters other functions operational.

Raycasts are potentially expensive, so the footIK script keeps the rays short and use a layerMask. You can modify the layerMask to suit your scene. There are also several options to turn off raycasts. The system default is setup to turn raycasts on and off based on velocity, but you could easily code other LOD schemes.

Script UserCustomIK

In UserCustomIK, Illustration 7, you may insert your own IK solution. Any change to the master's pose here will be the pose that the ragdoll tries to mimic. The script comes with a simple example of how a custom IK will override all other animations and previous IK. This code is not otherwise used and can all be deleted.

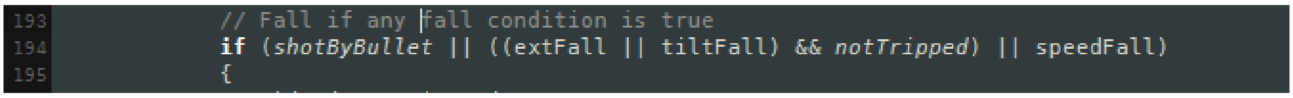


```
    // Put this on the master
    public class UserCustomIK_PBC : MonoBehaviour
    {
        public void DoCustomIK ()
        {
            // To use Final IK you need to comment out LateUpdate in the SolverManager,
            // and change the access modifier for the UpdateSolver methods from protected to public.
            // Then uncomment the line below

//          GetComponent<RootMotion.FinalIK.IK>().UpdateSolver(); // UpdateSolver needs to be public

            ////////////////////////////////////////////////////////////////////////////////////////////////////////

            // Any IK you write here will override everything
            // Example below
#if false
            Transform spine1 = GetComponent<AnimFollow_PBC>().masterRigidTransforms[7];
            spine1.rotation = Quaternion.FromToRotation (Vector3.forward, Vector3.forward + Vector3.right * Mathf.Sin (Time.time * 2f)) * spine1.rotation;
#endif
        }
    }
```

Illustration 7: The UserCustomIK script.

Script RagdollControl

RagdollControl is the script that controls how and when the character is to ragdoll and fall, or get back up. Illustration 8 shows the line with the conditions to trigger a fall. You can add or remove any fall condition you want for your project (in code).



```
            // Fall if any fall condition is true
            if (shotByBullet || ((extFall || tiltFall) && notTripped) || speedFall)
            {
```

Illustration 8: The conditions in script RagdollControl that triggers the ragdoll to fall.

The RagdollControl script is where you would add AI regarding for example falling animations or health system. The Limb scripts report to the RagdollControll script and can be used to know which limb was hit.

Script Animfollow

AnimFollow makes the ragdoll come alive and mimic the animated master. It controls the strength of the ragdoll. In this system AnimFollow has received upgrades in both precision and performance.

Script Limb

The limb scripts are distributed to all colliders on the ragdoll. These scripts monitor if that limb is colliding, and can be very useful to users that want to customize their character's behavior.

Inspector parameters

In this section some variables are explained. Apart from the parameters made visible in the inspector there may be a few more in the code for the interested user to fiddle with.

AnimFollow inspector

Ragdoll rigid transforms is an array containing the ragdoll transforms. It is made visible so that you can look at it and know the array index of any particular limb. This becomes useful if you want to adjust the **strength profiles,** which set the strength of an individual limb. To set the overall strengths use the **strength** parameters. The joint strength profile changes only takes effect if the **update inspector changes** is clicked. There are two types of strength: joint and force, experiment to get the effect you desire. The Stiffness parameters are for fine tuning the balance of the ragdoll's ability to follow the animation and the response to external forces. The **see master** bool is for debugging and will show what the master is doing (remember that you are looking at a ragdoll most of the time and that the ragdoll only tries to do what the master does). **Suspend ragdoll** makes the character a regular character. **Auto suspend ragdoll** makes the character transition back and forth between being a ragdoll as needed. **No lift off** makes the character harder to lift off the ground.

RagdollControl inspector

External force limit sets the amount of external force the ragdoll can experience before a fall is triggered. **Speed limit** sets the maximum relative speed allowed in a collision before a fall is triggered. The **Ext** and **speed limiten** bool indicates why a fall was triggered (should not be clicked on). The **residual strengths** affects the strength left in the ragdoll just after a fall is triggered. **Fall lerp** sets how fast the ragdoll loses its residual strength. The strength during a fall is also dependent on the ragdoll's velocity. **Stay down** makes the ragdoll not get up after a fall (is reversible). **Shot by bullet** can be used to trigger a fall. **Inhibit fall** turns off the ragdoll's ability to fall, you still have the non falling hit reactions because the character is soft.

AdvancedMoveClass inspector

The character falls by **gravity** if the character is not grounded if **always grounded** is not chosen. For animations such as climbing choose **use vertical root motion**. The character is normally balanced so that the character's center of gravity is kept above the transform, If that is not desired you should unmark **CG balance. Tilt as gravity** and **tilt as normal** sets the balance up direction to be constantly in the direction of the gravity or the ground normal. If the **current planet** field is not set to none, then the character will gravitate towards the transform chosen as current planet. **Perpendicular gravity** sets the character gravity to be perpendicular to the surface the character stands on. **Gravity set** is this characters individual gravity. **Acceleration input sensitivity** makes the character be controlled by phone acceleromerers. If a transform is assigned as **fake acceleration input** tilting that transform will have the same effect as tilting the phone. Use **turn towards velocity** when using accelerometer input. **Friction set** is the friction the character uses if the **use material friction** is not chosen (in which case the friction used will be the static friction specified by the physics material of the object the character stands on). **Scale step length** and **animator speed** modifies the movement animation currently played (slightly different effects if root motion is not chosen). A high value on **slip lerp** will give the character a cartoony running style. **Jump velocity** sets the jump height. The jump height is also controllable by the time the jump button is held. A tap will result in a smaller jump than a full press. The feel of the jump button can be tuned by the **jump response time** parameter. **Tame leaning** will make the character lean less into the acceleration.

GetWASD inspector

Assigning a transform to the **rabbit transform** makes the character chase that transform. You can use any navmesh system to move the rabbit around. If **mimic rabbit rotation** is chosen the character will orient as the rabbit is it is the **nav rotation distance**. You can **tune** the rabbit **follow speed. Mouse sensitivity** can be adjusted here if you have chosen to **use mouse**. **Master mouse thune** is for adjusting the mouse when the ragdoll is suspended. On some animations you may / may not, want to **use root rotation**. The GetWASD scriot is quite simple code so you would probably have no problem changing it to your liking. You can disable some of the buttons by choosing to **disable buttons**, and even more can be disabled by this line in the Main script: if (!disableInput), found at about line number 50.

Main inspector

Destroy when dead makes the character be destroyed when going off screen, if the character is fallen and RagdollControl.stayDown is true.

AdvancedFootIK inspector

The **grounded** bool is exposed just for debugging reasons. **Stick to transform** is experimental code that works kind of like parenting. It is also possible to parent the master to a (uniformly scaled) transform, but it might cause trouble in some systems that would needs some tweaking. **Foot IK weights** can be adjusted for left and right foot. You can disable some functionality by not choosing to **use predict, more predict** or both **toe and heel rays**, this could gain a little performance. There are some intelligence built In to automatically disable not needed raycasts. Raycasts will **ignore** some **layers**. For debugging purposes it is possible to **show raycasts**. The animation layer that bends the legs when the character is in the air is not fully polished and should be **disable aired layer** at first sign of trouble. The **limit foot snap** is used to limit the feet velocity at vertical edges. With for example crawling animations you may want to **disable foot fix** (use when the foot collider is interfering with the animation). **Ragdoll left and right foot** should be assigned (done in the automatic setup).

General good to know things

Here follows some things that I figured could be good to know.

Customizing the controls

Start by studying the GetWASD script.

Friction really is a good thing

At first some may feel it makes the character sluggish compared to a non physical character. Try sprinting and pressing R to do a roundkick, imagine how bad that would look if the transition did not have the friction gliding to aid it. Also try dragging the character over the undulated terrain, or running and turning on slopes, It looks really natural. Increase the friction coefficient to have more rapid response, but it will soon start to look unnatural.

Drag the character

If you did this step in the setup: Download this script: http://pastebin.com/j3DWqe3R
and drop it on the directional light in the scene. Change the GetMouseButton(0) to GetMouseButton(2) in the script, and change the spring from 50 to about 2000.
Then you can apply force to the ragdoll by clicking the middle mouse button and dragging.

Hot keys

There are some hot keys already coded (found in the GetWASD script).

Key B launches the CannonBall towards the transform assigned in the BallTest script.

Key H toggles suspend ragdoll.

Key N toggles slow motion.

Key L locks the cursor.

Keys shift and ctrl are sprint and walk.

Key C toggles camera.

Key G toggles gravity to be perpendicular to the surface the character is standing on.

Look at the animator, there are some more keys to try animations, like climb or roundkick.

Making animations work

Most animations just work. Some may need a custom setting. In the included animator I have shown how to make some animations work. In the animation's inspector try things like enabling / disabling “Bake into pose”. Fiddle with the options in the character inspector. With the right combination the animation will work.

Angular- and forceStrength

It is not entirely obvious what should happen when angular- or forceStrength is not zero or one. I can make the character do what ever, but any choice I make will only please some. If you cannot find a setting that makes the character behave the way you want, I can. I am confident I have built this tool to be very able, with several solutions lying dormant in the code.

Optimization

I always have performance in mind, but this is a general tool that should handle many scenarios. For your specific case there may be several places where performance can be gained by not caring about all the other possible scenarios.

The character is soft

That is kind of the point, it should act with human strength. There are several settings to adjust how soft / stiff it is. The BasicOnly character does not use the friction to limit the motion. It is very fast and has easy control code, but less features.

The code is sometimes complex

Because it handles a complex system. Expect that you will have to familiarize yourself with the code before you can do significant changes.

I am sure the code can, to some degree, be simplified more, but this is how it sits at the moment.

I am glad to help, but I will not build your game for you

Ask good questions, and you will get good answers. A good question is precise and has a real answer, like: Where is the grounded bool set? A bad question is: i wanna make a game like this link how do I do that.

Disable The aired layer

The animation layer that bends the legs when the character is in the air is not fully polished and should be **disable aired layer** at first sign of trouble. It does not look good over sharp vertical edges, and it can interfere with animations that takes the character off the ground.

It is still beta

There might be bugs, and patches that aren't perfectly polished or optimized.
I am working on a solution for generic animations.

Be helpful

The character should handle most situations gracefully, but it is possible to make a scene in which the character struggles. If it is an honest scene, with situations it is reasonable that the character should handle, then I'll fix so that it does. In the mean time, help the character by building scenes that make the character look good, because it makes you look good.