PBC, Physically Based Character controller

There is no going back

Patrik Tegelberg, 2015-sept-27

To me games are part escape and part intense experience. Good games are not technology driven, but bad technology makes for a inferior game experience. Good games have technology to match the story.

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

To achieve this a physical character (known as a ragdoll) is augmented to accept any canned animation. The ragdoll can thus have all the functionality of a regular character controller, but also interacts realistically with the environment. It pushes on, and is pushed by, the environment.

The automated ragdoll creation and character setup eliminates much of the boring and tedium. Just drop the CreateHumanRagdoll script onto your character (with a valid mecanim avatar), choose an option and click createNow. The character will, without any more work now: interact with the game objects in your scene, be knocked back, react to hits, follow moving platforms and have good foot IK.

There is no going back. In eight years all characters will be physically based, just like all shaders soon are physically based. There is no going back, be a pioneer now, or a follower later.

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 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 performance than non-physical state machines. A few of possibilities mentioned below will require some coding, whether you consider the coding needed easy or hard will depend on your skill level. 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.

Knock back

The character has a will, it desires to do your bidding. It wants to follow the animations and root motions, but it is human and has limited 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 also possible (in code) to switch animation and have the ragdoll try to follow a custom fall animation during fall. After the fall the character will regain strength and smoothly rise to continue following your animations.

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.

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.

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.

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.

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 centre of mass directly above the transform. Imagine how you look when you carry one heavy grocery bag.

Natural ground follow

When you run over undulated terrain the distance from the ground to your centre 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.

Advanced 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.

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

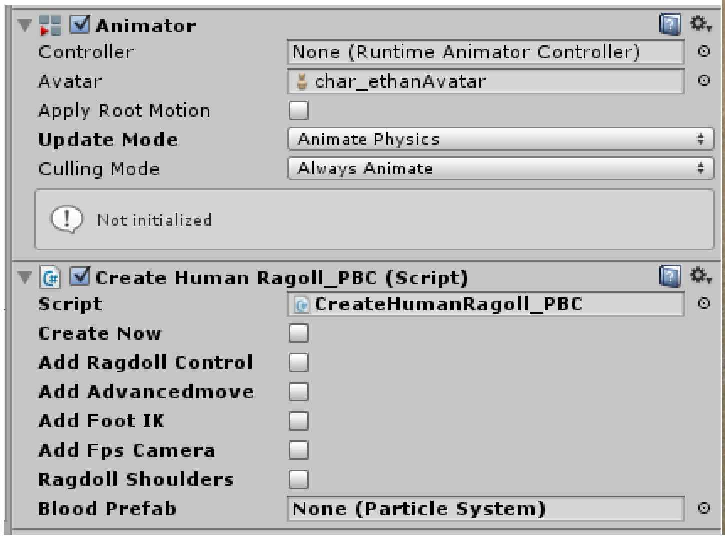
Choosing the NPC option makes the character disregard the normal user inputs and follow the animation and animation root motion. This could also be used to make your own animator based motion controller.

No MMO button

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

Setup

- ✗ 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.

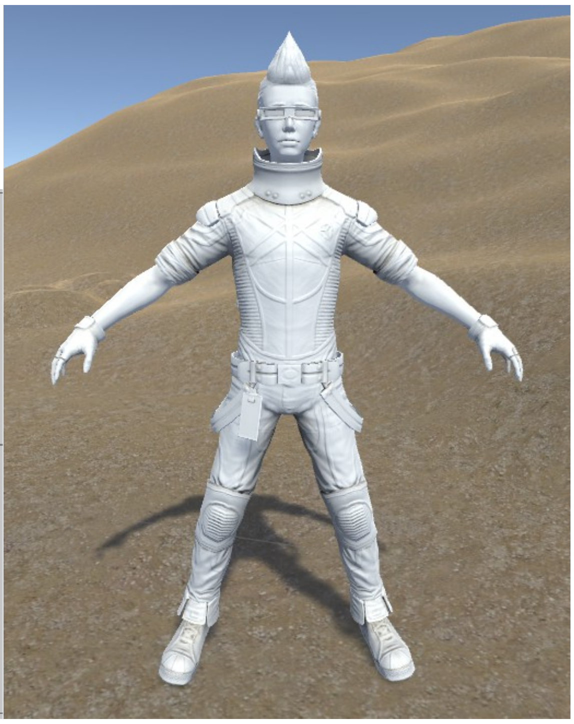


Illustration 1: Good setup pose.

- ✗ 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 centre of the limbs' rotational limits.
- ✗ Drag and drop the **CreateHumanRagdoll** 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, and that the colliders have appropriate size.

Optionally

- ✗ In the Camera3rd inspector, assign your character's hips to the lookAtTransform. Assign your characters headCamera (if you choose the FPS camera) to camera3rd's head camera field.
- ✗ Assign any ragdoll transform to the cannon ball's hit transform field.
- ✗ Download this script: http://pastebin.com/j3DWqe3R
and drop it on any transform in the scene. Change the GetMouseButton(0) to GetMouseButton(2) in the script, and change the spring from 50 to about 5000.

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.

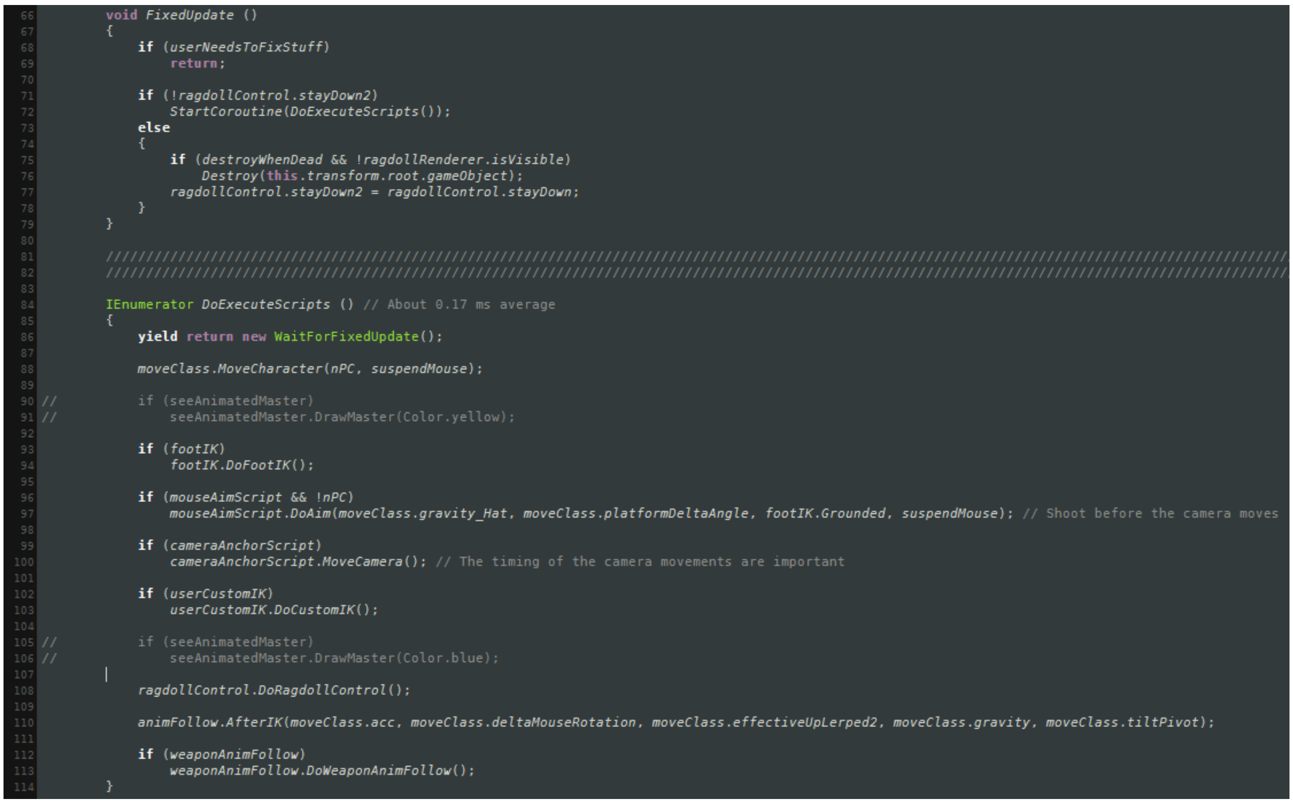
Scripts overview

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 behaviour.

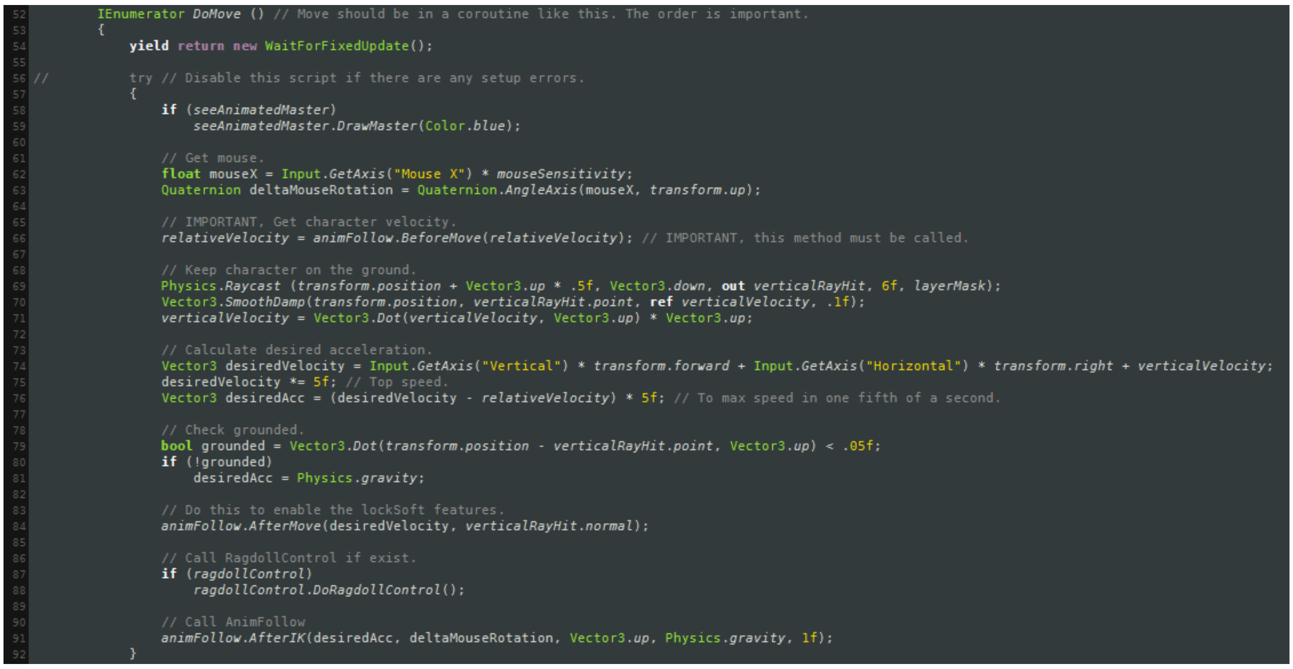


```
        void FixedUpdate ()
        {
            if (userNeedsToFixStuff)
                return;

            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.

Script SimpleMoveScript

This asset is complicated. Many things must happen in sync to get the desired result. As a service to anyone feeling overwhelmed I provide the SimpleMoveScript, Illustration 4, that is an example of what must happen to have any kind of working system. To get the simple character you choose nothing in the setup script, just click create now.

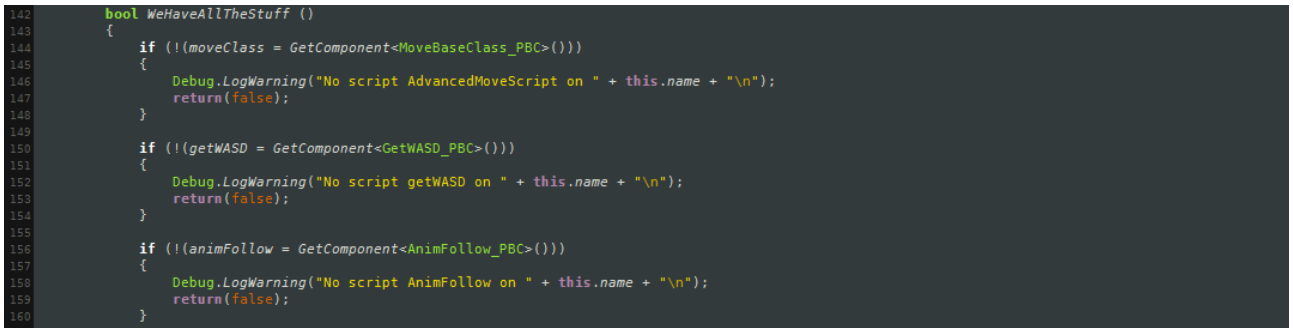


```
        IEnumerator DoMove () // Move should be in a coroutine like this. The order is important.
        {
            yield return new WaitForFixedUpdate();

//          try // Disable this script if there are any setup errors.
            {
                if (seeAnimatedMaster)
                    seeAnimatedMaster.DrawMaster(Color.blue);

                // Get mouse.
                float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity;
                Quaternion deltaMouseRotation = Quaternion.AngleAxis(mouseX, transform.up);

                // IMPORTANT, Get character velocity.
                relativeVelocity = animFollow.BeforeMove(relativeVelocity); // IMPORTANT, this method must be called.

                // Keep character on the ground.
                Physics.Raycast (transform.position + Vector3.up * .5f, Vector3.down, out verticalRayHit, 6f, layerMask);
                Vector3.SmoothDamp(transform.position, verticalRayHit.point, ref verticalVelocity, .1f);
                verticalVelocity = Vector3.Dot(verticalVelocity, Vector3.up) * Vector3.up;

                // Calculate desired acceleration.
                Vector3 desiredVelocity = Input.GetAxis("Vertical") * transform.forward + Input.GetAxis("Horizontal") * transform.right + verticalVelocity;
                desiredVelocity *= 5f; // Top speed.
                Vector3 desiredAcc = (desiredVelocity - relativeVelocity) * 5f; // To max speed in one fifth of a second.

                // Check grounded.
                bool grounded = Vector3.Dot(transform.position - verticalRayHit.point, Vector3.up) < .05f;
                if (!grounded)
                    desiredAcc = Physics.gravity;

                // Do this to enable the lockSoft features.
                animFollow.AfterMove(desiredVelocity, verticalRayHit.normal);

                // Call RagdollControl if exist.
                if (ragdollControl)
                    ragdollControl.DoRagdollControl();

                // Call AnimFollow
                animFollow.AfterIK(desiredAcc, deltaMouseRotation, Vector3.up, Physics.gravity, 1f);
            }
```

Illustration 4: The SimpleMoveScripts 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.

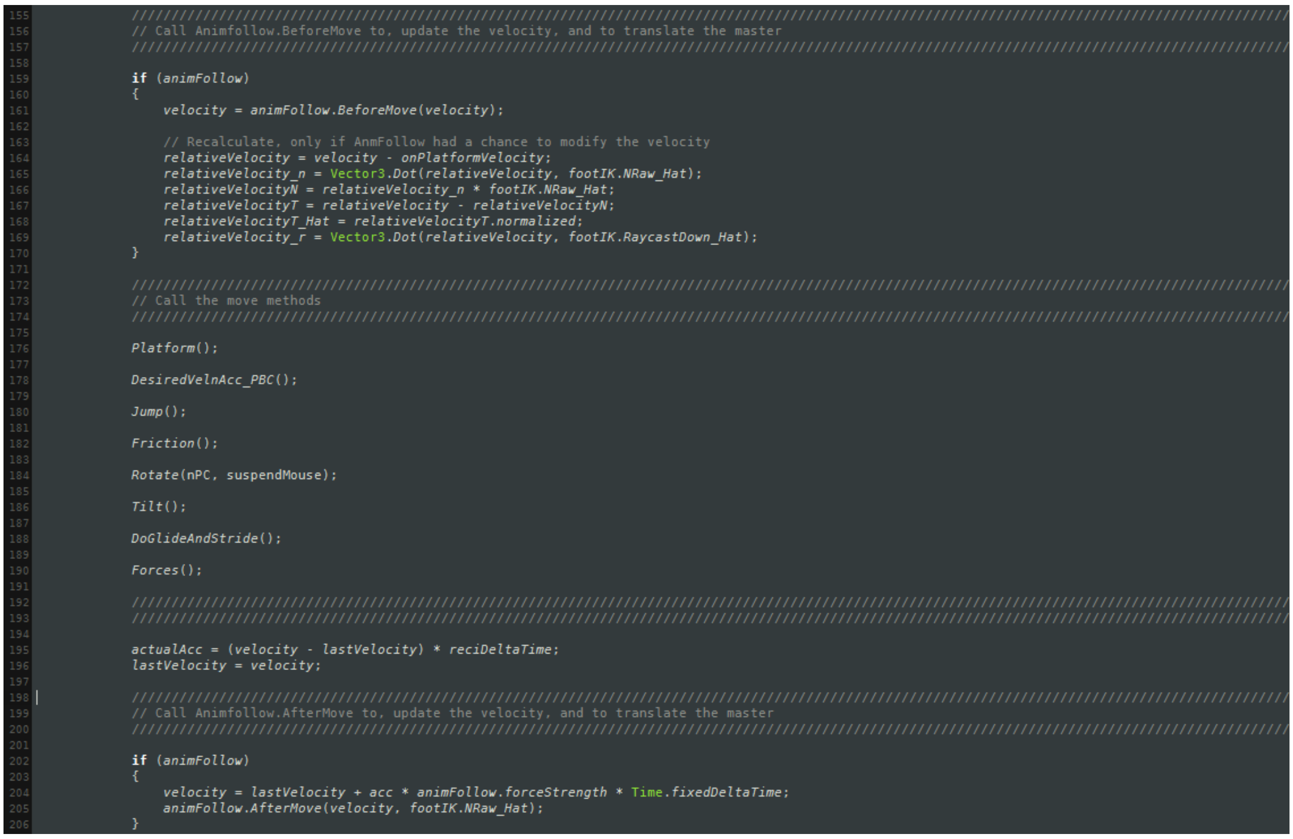


```
        bool WeHaveAllTheStuff ()
        {
            if (!(moveClass = GetComponent<MoveBaseClass_PBC>()))
            {
                Debug.LogWarning("No script AdvancedMoveScript on " + this.name + "\n");
                return(false);
            }

            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 both the Advanced- and SimpleMoveScript call methods in AnimFollow. Tese calls 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 methods AnimFollow.BeforeMove and AfterMove get 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.

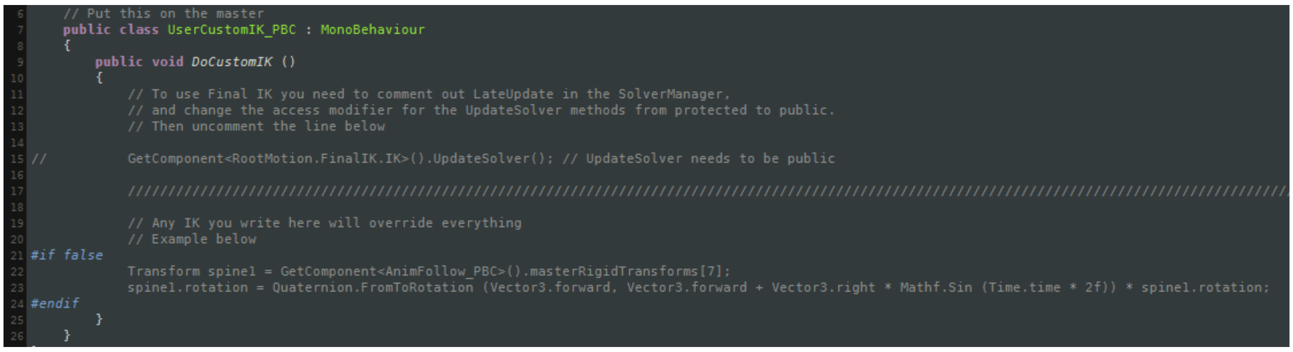
Script AdvancedFootIK

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.

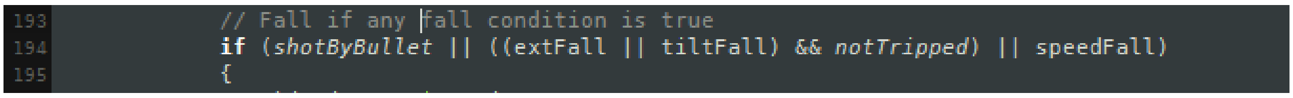


```
    // Put this on the master
    public class UserCustomIK_PBC : MonoBehaviour
    {
        public void DoCustomIK ()
        {
            // To use Final IK you need to comment out LateUpdate in the SolverManager,
            // and change the access modifier for the UpdateSolver methods from 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 if the character is to ragdoll and fall. Illustration 8 shows the line with the conditions to trigger a fall. You can add 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.

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 customise their character's behaviour.

Inspector parameters

In this section some variables are explained, both variables that are made visible in the inspector and variables that you may want to make visible to play around with.

AnimFollow inspector

ragdollRigidTransforms is an array containing the ragdoll transforms. It is made visible so that I 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** parameter. The joint strengh profile changes only takes effect if the **updateInspectorChanges** is clicked. There are two types of strength: joint and force, experiment to get the effect you desire. The **Dforce** parameter sets if the ragdoll is biased to mimic the master limb positions or the master limb velocities. A low Dforce makes the ragdoll stiffer, and is follows the animation more closely. **ForceTune** also affects the ragdolls stiffness, set to one for max stiffness. The **softRotation** and **softTilt** set the strength with which the ragdoll's rotation is trying to align with the master's rotation. Keep softRotation at zero for for FPS characters and if you have rotating platforms. **LockSoftN** and **T** disable the character's ability to be pushed around by other colliders. It can be set to only affect the directions normal and tangential to the ground. The animations and jumps will still work. A**utoLockSoft** turns off the lockSofts when a collision is detected by the limb scripts. 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).

There are a few more variables in the script that you might want to fiddle with:

```
float angularDrag = 12f;                // Rigidbodies angular drag
float drag = 0f;                        // Rigidbodies drag
float jointDamping = .1f;               // Does not yet work in Unity 5. Bug in Unity, not in AnimFollow
float fullJointStrength = 10000f;       // PhysX joint strength value when jointStrength is set to one
```

RagdollControl inspector

External force limit sets the amount of external force the ragdoll can experience before a fall is triggered. **Tilt limit** sets the amount of tilt angle between the master and the ragdoll that is allowed before a fall is triggered. **Speed limit** sets the maximum relative speed allowed in a collision before a fall is triggered. The **Ext, tilt** 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 (also dependent on the ragdolls velocity at impact). **Fall lerp** sets how fast the ragdoll loses its residual strength. **Stay down** makes the ragdoll not get up after a fall (is reversible). **Shot by bullet** can be used to trigger a fall. InhibitFall turns off the ragdolls ability to fall (looks cool to inhibit fall and shoot the ragdoll).

There are a few more variables in the script that you might want to fiddle with:

```
float settledSpeed = .5f;           // Velocity limit below which the falling state is over and the get up starts
```

FollowRabbit inspector

As soon as you assign a “rabbit” transform to the **followTransform** field the character moves towards that rabbit transform. When it is close enough it will also follow the rabbit's ratation. To use this with NavMesh or equivalent AI, move the rabbit with the AI.

MouseAim inspector

Mouse sensitivity independently for the horizontal and the vertical direction. **Mouse smooth time** smooths mouse movements, is set independently if the head camera smooth time to avoid the hysteric feel.

There are a few more variables in the script that you might want to fiddle with:

```
float zoom = .25f;                      // Zoom magnification (reciprocal)
float mouseZoomSensRatio = .5f;         // Mouse sensitivity when zoomed in
int mouseMode = 0;                      // Three modes to choose from
```

AdvancedMoveClass inspector

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. **GravitySet** is this characters individual gravity. **Use tilt acc** makes the character accelerate in the direction the ragdoll is tilted relative to the master. **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. **Use root motion** alternates between the option of having the character's motion driven by the animation's root motion or another option which scales the step length to fit the desired speed. Use slip lerp about 0.3 if you do not use root motion. **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. **Mouse force** is a work in progress. **Tame leaning** will make the character lean less into the acceleration. **AimToBodyRotVelRatio** and **aimToBodyLerp** sets how the character reacts to mouse velocity and how fasr the character aligns with the mouseAim transform. Enforce rotation makes the rotation disregard maxRotationAcc.

There are a few more variables in the script that you might want to fiddle with:

```
// In the Jump script
float P_Vertical = 100;                 // Parameter to adjust the PD controller used to follow the ground
float D_Vertical = .01f;                // Parameter to adjust the PD controller used to follow the ground
float verticalTune = .15f;              // Parameter to adjust the input to the PD controller
// In the Rotate script
float maxRotationAcc = 4000f;           // Basically this works as a friction parameter for rotations
float aimToBodyRotVelRatio = .3f;       // Mouse feel
float aimToBodyLerp = .1f;              // Mouse feel
// In the Tilt script
float effectiveUpSmoothTime = .25f;     // Controls the smoothing of the character lean
float effectiveUpSmoothTime2 = .1f;     // Controls the smoothing of the character lean
```

Main inspector

Destroy when dead makes the character be destroyed when going off screen, if the character is fallen and RagdollControl.stayDown is true. **nPC** disables WASD and mouse control and the character move by the root motion of its animation.

General good to know things

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

The character is soft

That is kind of the point, it should act human. There are several settings to adjust how soft it is. It can be made quite stiff if you need it to be. Here follows parameters that affect the characters stiffness.

Dforce, lower is stiffer.

ForceTune, higher is stiffer.

SoftTiltSet and softRotationSet, lower is stiffer.

Force- and jointStrenght, lower is softer.

Force- and jointStrengthProfile, lower is softer per individual limb.

LockSoft, true is stiffer.

FrictionSet, higher is stiffer.

Edit/ProjectSettings/Time/Fixed timestep, lower is stiffer. (very stiff, but costs performance)

MaxRotationAcc, higher is stiffer. (private parameter in the Rotate script)

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.

Customizing the controls

To make your own customized controls you might, or might not, have to do things differently than you are used to. If you take the time to study how I have implemented my controls I am sure you will, after some swearing, be able to have it precisely the way you want.

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: Can I use the limb scripts to get different damage on different limbs? A bad question is: i wanna make a game like this link how do I do that.