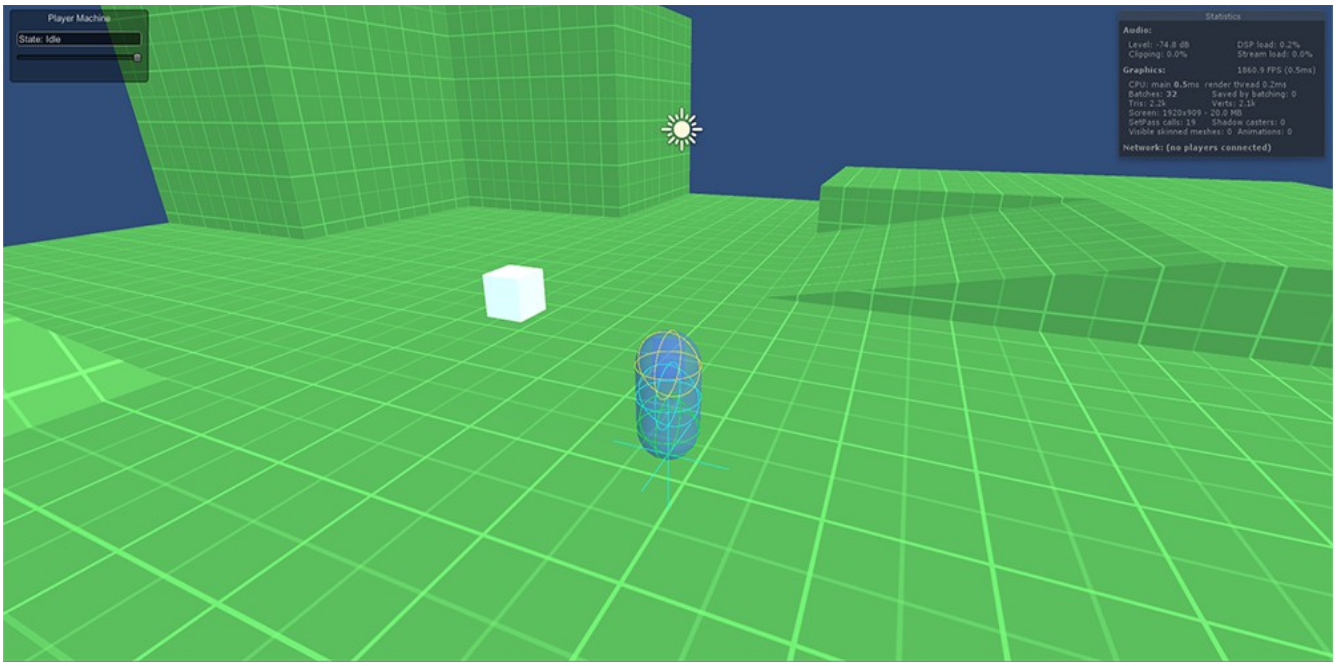
SUPER CHARACTER CONTROLLER

Erik Ross
http://roystanross.wordpress.com/

TABLE OF CONTENTS



PURPOSE

The Super Character Controller (SCC) functions as a replacement for the built-in Unity Character Controller component. This component's purpose (both the original and the new) is to resolve collision detection between the character and other objects as well as to provide useful data about the collision. In addition, it also provides common features that most characters need, such as detecting ground below the controller and limiting what angle of slopes he can climb.

This document serves to explain how to use the SCC in your projects as well as highlights any current issues with it, and what future work could be done.

INSIDE THE PACKAGE

At the top level directory:

DebugDraw: Useful class with a few drawing methods that use Debug.DrawLine.

Math3d: Incredibly essential class, taken from the Unity wiki, filled with all sorts of 3d math methods.

Inside the *Core* directory:

SuperCharacterController: Main component that is attached to your object to resolve collision and provide useful data.

SuperCollider: Set of static functions to retrieve the nearest point on the surface of a collider.

SuperCollisionType: Data structure required to be attached to all objects the SCC collides with. This is passed to the SCC during collision. Extend this class to add further fields.

SuperMath: Library of useful functions, not all really math.

SuperStateMachine: Models a State Machine specifically for use with the SCC. Extend this class to use as a base for your characters.

Inside the *RPGController* directory:

RPGMesh: Attach this to any mesh to bake a triangle tree on it, and allow *SuperCollider* to find the nearest point on it's surface during collision. This is *required* for any Mesh collider to be detected by the SCC.

RPGTriangleTree: Used by RPGMesh, no need to do anything with this.

SCC OVERVIEW

All of the main logic is done in the SingleUpdate() method. It does the following:

- Probes the ground to see what is below the controller
- Sends a message to run the “SuperUpdate” method on all components attached to the object the SCC is attached to. This is where you run any movement logic you want on your character.
- Resolves any collision with objects
- Slope limits the character
- If the character is clamped to the ground, adjust his position so it matches the ground beneath him
- If the character is clamped to the ground and the ground is *moving,* move him along with it

SingleUpdate is called one or more times per Update(). You can set a fixed timestep by enabling the fixedTimeStep boolean and setting the minimum fixed updates per second to be fixedUpdatesPerSecond (I used 60). This ensures SingleUpdate is run *at least* 60 times per second.

PushBack

This method checks to see if any of the collision spheres are contacting any objects, and resolves the collision if so. It also updates the collisionData variable with *each* collision that occurred during the frame. The collision data also retrieves the SuperCollisionType component from the collided object.

SuperCollisionType

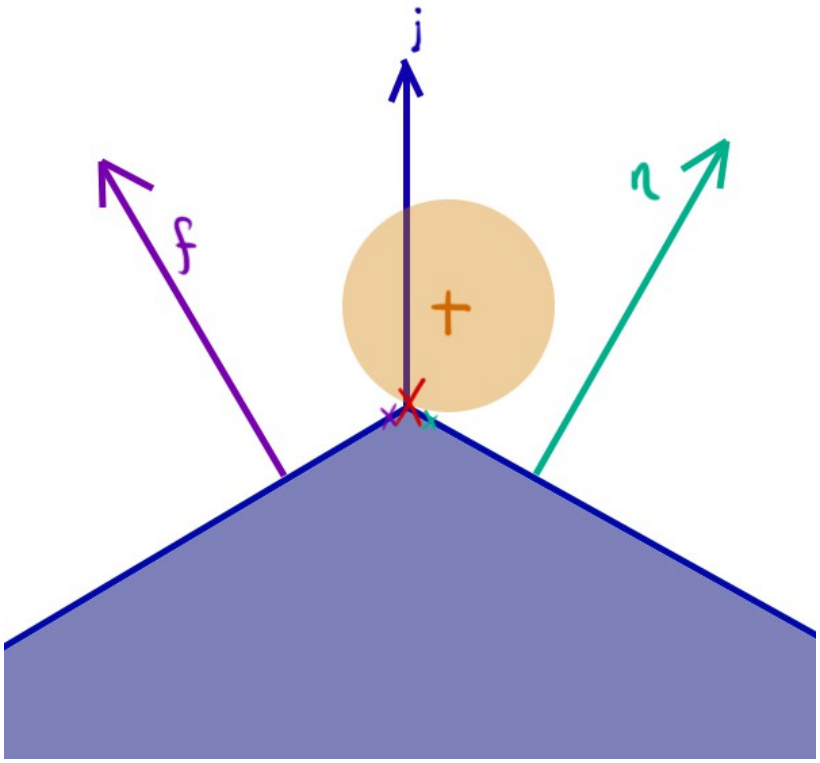
Component required to be attached to each object the SCC collides with. It has two fields: SlopeLimit and StandAngle. SlopeLimit defines (*if* the controller currently has slope limiting enabled!) the maximum slope the controller can travel over. StandAngle defines the maximum angle *that will be detected by the ProbeGround method.* Anything above that is treated as a “wall.”

ProbeGround

Proper ground detection is extremely important to making a character move smoothly across surfaces and against walls. ProbeGroundRecursive SphereCasts below the player, checking for ground. If it contacts a surface, but the angle of the surface is greater than the StandAngle of the object, it will SphereCast again downwards, with the origin of the cast adjusted to be just below the point contacted with the previous cast (hence it being *recursive*)*.

An issue (or feature?) with SphereCast is that when it contacts an *edge* of a surface (rather than just a triangle) the *RaycastHit.normal* that is return is the interpolated value of the two triangles joined to the edge. Because we typically want to know what our character is actually standing on, we use two further Raycasts (one of each triangle) to get the actual normals of the surface we're standing on. These are named *NearHit* and *FarHit,* meaning the normal of the point closest to the center of the controller, and the point furthest from the center of the controller (one on either side of the edge the primary SphereCast contacted).

*Because of SphereCast being unreliable, this part needs work and can be buggy. See the **Future Work** section



Above is shown in blue (vector i) the interpolated normal of the two surfaces. In purple (f) is the FarHit, and in teal (n) the NearHit. The Red X shows where the SphereCast contacted, while the purple and teal approximate where the Raycasts would.

PlayerMachine

This is a demo character implementing the SuperCharacterController *as well* as the SuperStateMachine. It *IS NOT* intended to be a drag-and-drop solution, but rather just an example to show how these tools are meant to be used. Most of the code included is rather simple and exists for only demo purposes, but it's worth bringing up the IsGroundedAdvanced method.

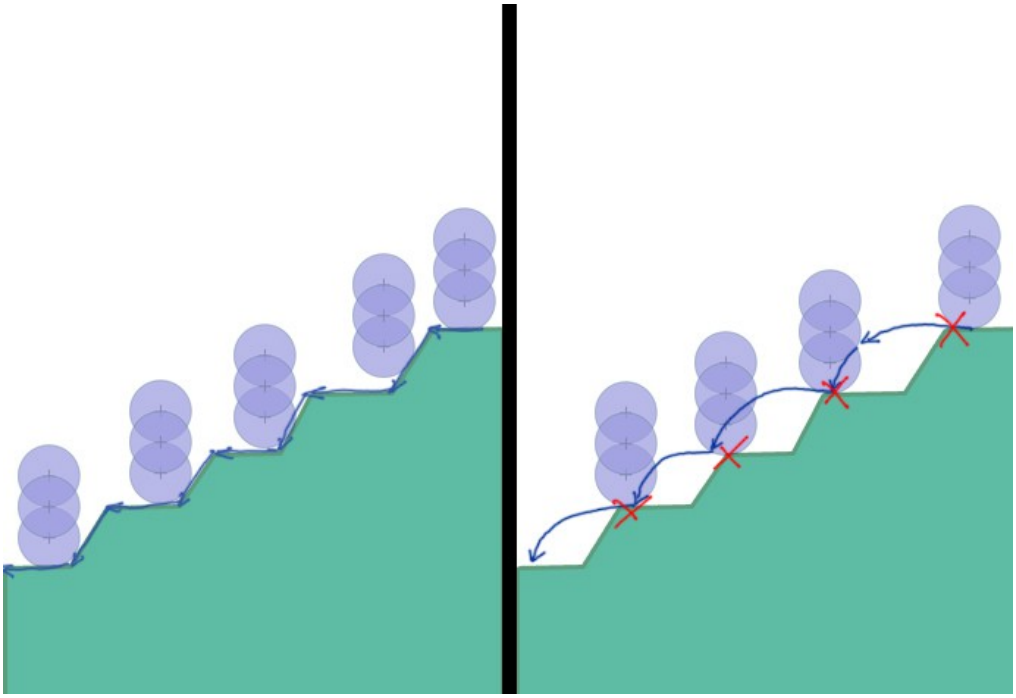
IsGroundedAdvanced

Simply named “IsGrounded” in the Mario project

Detecting ground isn't enough—you actually need to do something with the data for it to be useful. However, this isn't really the SCC's job, so included in the PlayerMachine is an example of how to make use of this data.

It's worth noting that for most characters, when they are counted as “grounded” they are typically also “clamped” to the ground, essentially locking their movement over the surface.

A naive way to query if our character is “grounded” or not would be to check how large the currentGround.Hit.distance is, and if it's too large for our character's legs to be plausibly touching then we return a false value. This is a good start but comes with a few issues. What should we decide the distance should be that he can stand on? If you make it too low, the character will continuously be declared ungrounded as he moves over uneven surfaces (and thus declamped) making for frustrating controls.

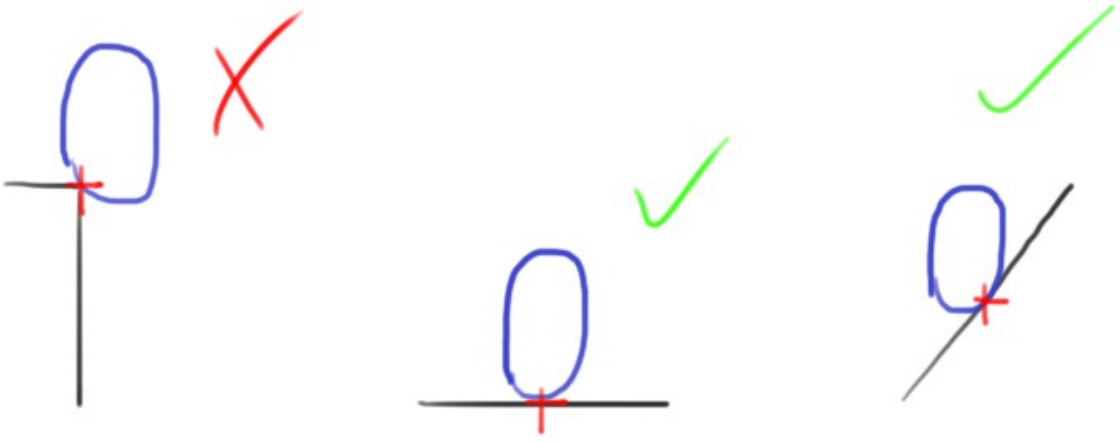


The left image shows how the character's movement follows the uneven surface by ground clamping. On the right, we see how he "bounces" across the surface when clamping is not applied. Each red "X" represent when the downward force of gravity is zeroed out

The trick is to pick a large value for when you are currently "grounded" (about 0.5) and querying if your character is *maintaining* ground, and a small value if you are currently ungrounded.

This is still a fairly naive method, since it assumes that if our player is touching a ground of less than *X* distance, we are grounded. It doesn't, however, take into account at all how far away from the center of the controller that point is (are we standing *just* off a ledge maybe? Should this count as "ground"?).

To resolve this we can check if the point is far away from the controller's center (say greater than 90% it's radius, for example), and declare that that is ungrounded. The problem that arises from this is that we don't actually always desire that to count as ungrounded, as we can see in the image below.



In the middle image above, we see the controller standing on a flat surface, with the contact point (a red cross) directly below him. This should obviously be grounded. On the left he is standing on a ledge, *right* at the edge. In our case we are stating that the point is too far from the center of the controller and *should NOT* be counted as grounded. The far right presents an interesting problem. The contact point of the ground is very far from the center of the controller, but our character is not standing on a ledge—he's standing on a slope. We desire that this be counted as grounded, but how to logically express that?

This is solved by creating a relationship between the allowed distance the contact point can be from the center of the controller and the slope of the surface we're standing on. If the surface is flat, the contact point should be required to be quite close to the center, so that we can't stand on ledges. If the surface is quite angled, the point should be allowed to be further away, as in the far right image.

All of this is shown in the IsGroundedAdvanced method.

FUTURE WORK

Ground Detection

The unreliable ground detection is an issue. The main goal should be to—either through SphereCast or OverlapSpheres—get a clear picture of what's below the player, and what we're "standing" on. The main problem is when "walls" angled at around 85-90 degrees get in the way of the SphereCast. To solve this, I added in a recursive SphereCast that keeps recasting if it collides with a wall. This works pretty well, but SphereCast isn't very reliable. Recursive Raycasting is pretty easy, since you just set the origin to slightly past the point that was contacted by a previous raycast, and you are *assured* that it will not contact this again.

SphereCasting in Unity is a bit different. Sometimes it doesn't detect anything within the radius of it's origin, sometimes it does. Either way it can lead to some bugs where the controller detects ground that isn't there or is unable to detect ground that *is* there.

Nearest point on Mesh and RPGTriangleTree

RPGMesh is an extremely useful piece of code, but I've found that with lots of SCCs and large mesh colliders (i.e., with lots of triangles) can lead to pretty significant slowdowns. Implementing a Binary Space Partitioning Tree (BSP Tree) as a way to quickly locate vertices on a mesh has been on my list of things to do for awhile, but I'm not sure when I'll get to it. I'd like the SCC to be as efficient as possible, so it would be a nice addition.

SOURCES

- RPGMesh, RPGTriangleTree and the original PushBack algorithm are from fholm's RPGController package, which can be viewed here: https://github.com/fholm/unityassets/tree/master/RPGController
- UnityGems.com (now dead, but can be accessed through web archives) had a series of tutorials on State Machines where the Super State Machine was adapted from: https://web.archive.org/web/20140702051240/http://unitygems.com/fsm1/
- BitBarrelMedia wrote the awesome Math3d class: http://wiki.unity3d.com/index.php/3d_Math_functions