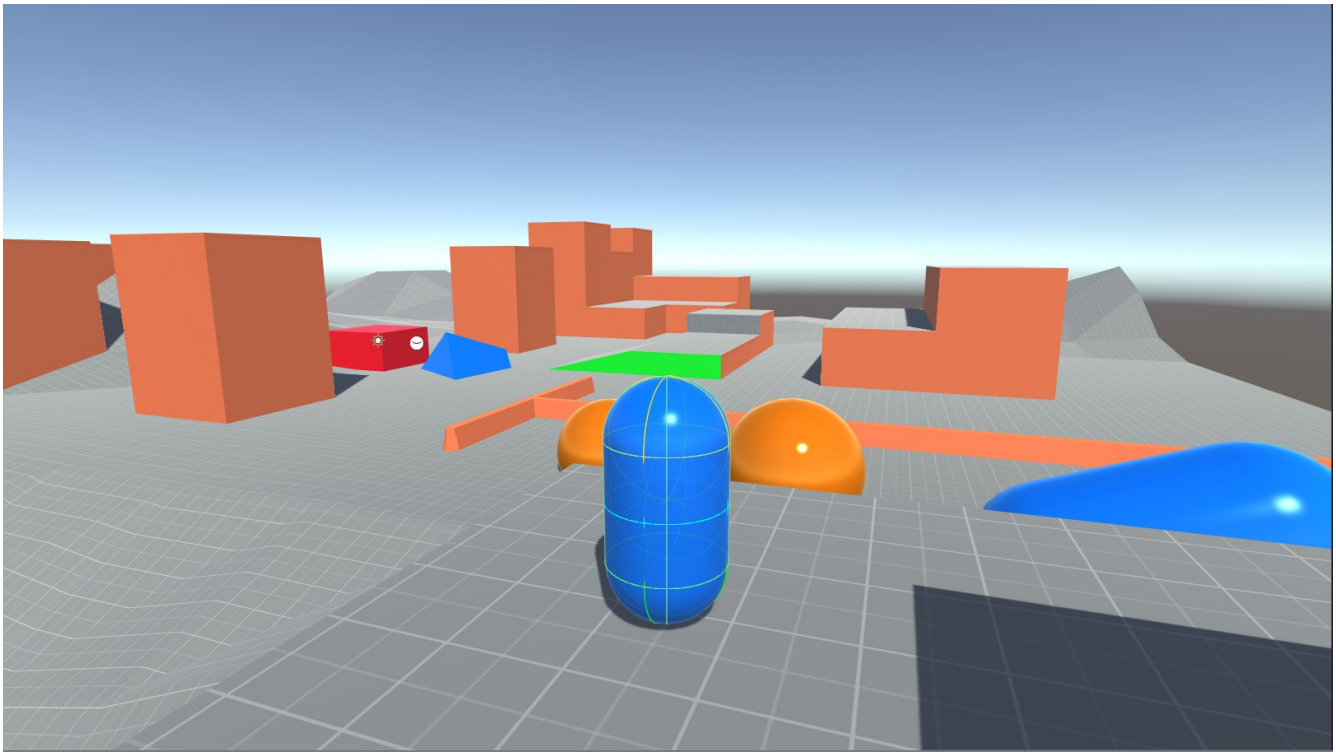
SUPER CHARACTER CONTROLLER V 2.x.x

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

TABLE OF CONTENTS

PURPOSE.....2
INSIDE THE PACKAGE.....2
OVERVIEW.....3
PlayerMachine.....4
TEST SCENES.....4
INITIAL SETUP.....4
MOVING PLATFORMS.....6
FUTURE WORK.....7
SOURCES.....7



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 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.

BSPTree: Attach this to any mesh to bake a BSP triangle tree onto 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.

BruteForceMesh: Runs the same nearest point algorithm as BSPTree, minus the tree pre-baking (i.e., every triangle is checked if it’s nearest). Used for debugging purposes mostly.

[Deprecated in favor of BSPTree] 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, in order:

- 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.

If the SCC is *still* colliding with other objects at the end of the frame, PushBack is run recursively until it is not colliding with any objects or a maximum allowed number of iterations occurs (*MaxPushbackIterations*)

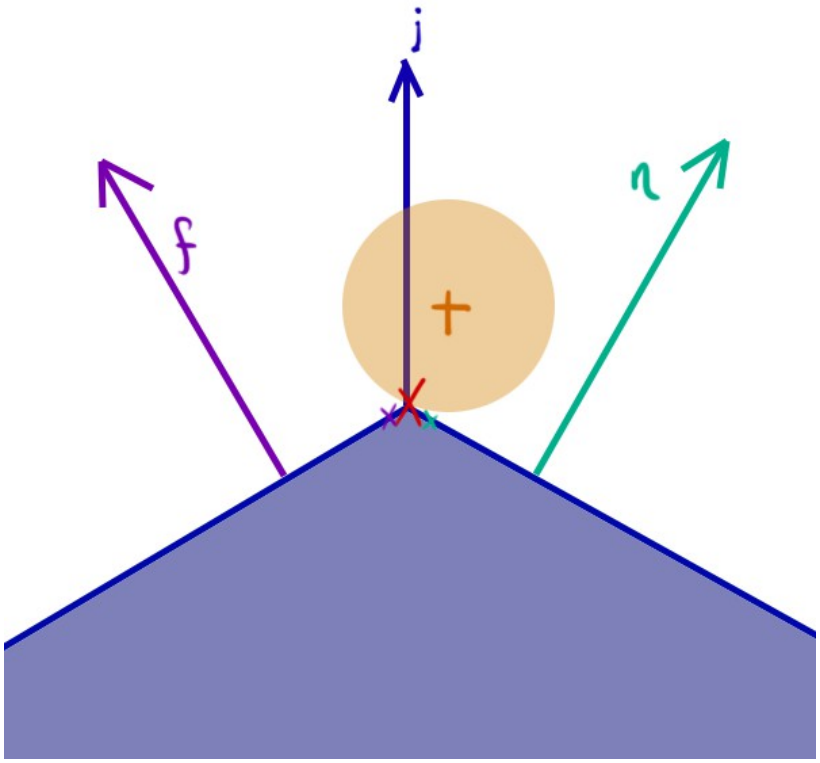
SuperCollisionType

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.” This class is intended to be extended upon for your own projects, to allow you to add further properties to it.

ProbeGround (method of *SuperGround*)

Proper ground detection is extremely important to making a character move smoothly across surfaces and against walls. ProbeGround 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 Raycast down the surface to find the “true” ground.

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).



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.

A more in-depth analysis of the ProbeGround method can be found here: https://roystanross.wordpress.com/2015/05/10/custom-character-controller-in-unity-part-6-ground-detection/

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.

The PlayerMachine extends the SuperStateMachine class, allowing it to dynamically detect state methods. In each scene is a Player gameobject that implements the SCC and the PlayerMachine. Notice that the player's "Art" object has a collider attached and is placed in the OwnCollider slot of the SCC. This allows other objects to collide with the SCC at runtime.

Test Scenes

TestZone: Contains a player controller and a series of objects to test against.
SpaceZone: Demonstrates the rotation of the player controller to walk around a planet.
TerrainTest: Demonstrates the TerrainCollider implementation.

Initial Setup

The following steps will setup a new project with the SCC package, from scratch (an alternative to

using the example project.

- Import the SCC package
- Rename one layer “TempCast”

To setup a new controller, go through the following steps:

- Attach the SuperCharacterController.cs script to an object
- Ensure the Spheres array is defined (check DebugSpheres to see where they are)
- Modify the radius/placement of the spheres to your needs
- Make sure some layers are in the Walkable LayerMask
- Make sure the objects in your world you want to collide with are also on that layer!

To implement the SCC, do the following:

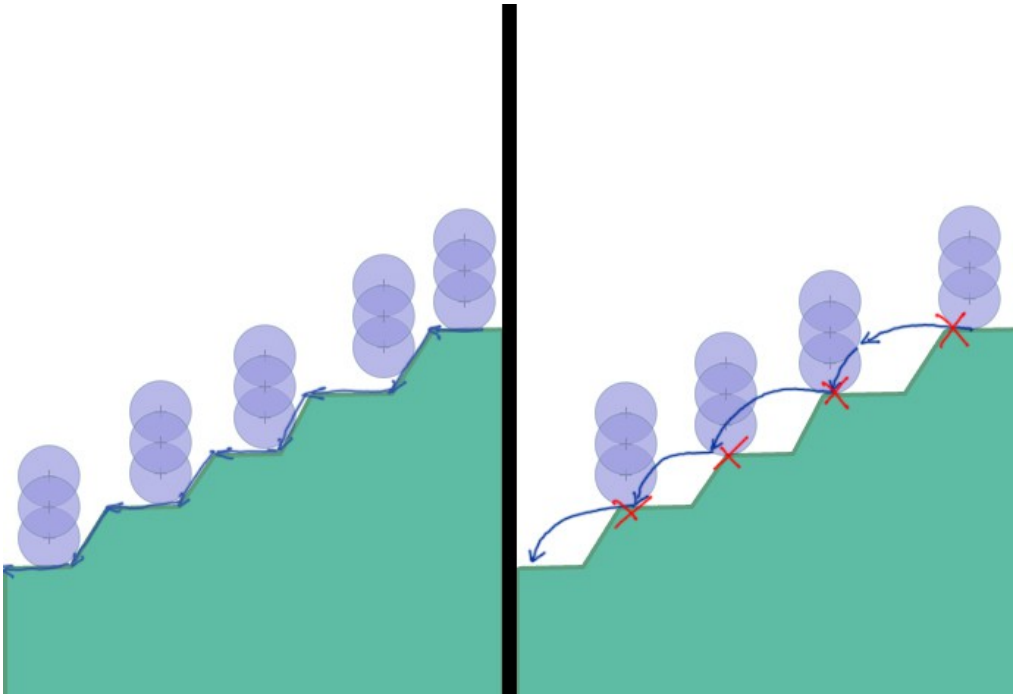
- Create a new script
- Add the “SuperUpdate” method into the script. The SuperCharacterController will call this method in it’s Update loop.
- Use the Enable and Disable clamping methods to set whether the controller should be snapped to the ground.
- Use any method to move your controller, e.g., transform.position +=, transform.Translate...
- Use any method to rotate your controller.

SuperGround and IsGrounded

Detecting ground isn't enough—you actually need to do something with the data for it to be useful.

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 distance is of our primary SphereCast, 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.

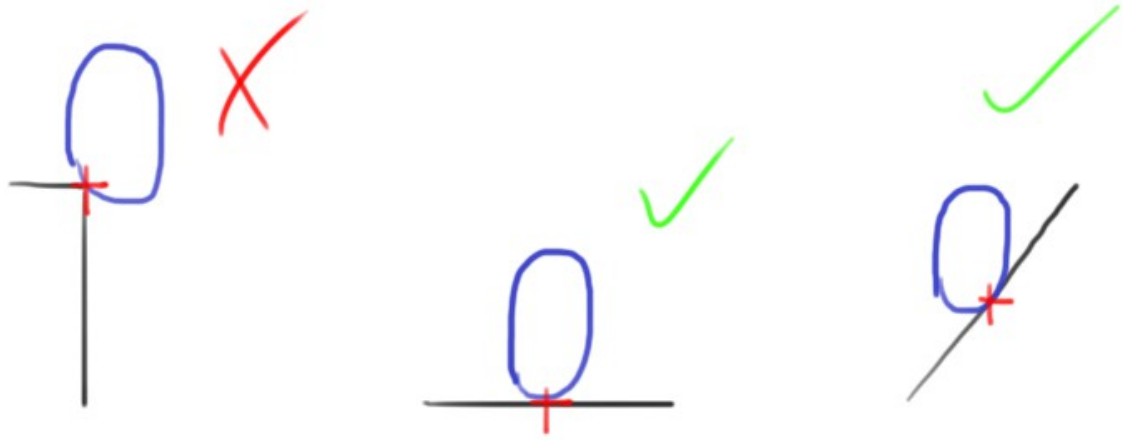


The left image shows how the character's movement follows the uneven surface by ground clamping. On the right, we see how he "bounces" across the surface when clamping is not applied. Each red "X" represent when the downward force of gravity is zeroed out

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 IsGrounded method.

MOVING PLATFORMS

There are two steps to ensuring moving platforms work properly. One is to make sure that if your character is standing on the platform and it moves, he moves with it accordingly. This is fairly easy to do and is done in the SingleUpdate.

The second step is to make sure that the SCC is capable of detecting contact with the platform when it

moves. A naive method would be to simply move the platform in Update() (using transform.position += ... or whatever), and then let the Physics engine sort it out. Unfortunately, this does not work in Unity as an object's actual position in the Physics engine is *NOT* updated immediately when it is moved, but rather it is done at the end of FixedUpdate(). To see the full listing of the order of operations in Unity, look here:

http://docs.unity3d.com/Manual/ExecutionOrder.html

The bottom line is we need to move our platforms (or any moving object you want to collide with the SCC) before the internal physics update is run, i.e., in FixedUpdate(). This works perfectly, but there is a hitch: FixedUpdate() runs irregularly: sometimes it runs once per frame, sometimes more than once, sometimes not even at all. A workaround for this is to set the Fixed Timestep (in the Time panel in Unity) to be less than your target framerate. That way, you can be sure it will run *at least* once per frame.

FUTURE WORK

Ground Detection

Ground detection can *always* be improved.

Binary Space Partitioning (BSP) Tree

BSPTree is new in V2, but it still needs some work. An important part of a BSPTree is how we select the point and normal of the partition plane to split the mesh in half (recursively). Currently, BSPTree builds an axis-oriented bounding box around all relevant vertices and then uses the center of this box as the origin of the partition plane. The normal of the plane is simply the direction facing the longest axis of the bounding box. I.e., it grabs all vertices and then neatly cuts them in half. This works and is fairly effective, but it doesn't take into account the overall "mass" of the vertex set; if vertices are not relatively evenly spaced apart, you could end up with a set of 100 vertices being split into two sets of 99 and 1 vertices. This is one improvement that should be made.

As well, since this is such an essential class of the SCC, any optimization are always useful.

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