Animation at runtime of a blob shaped character using multiple colliders to simulate a global shape

Corentin Serrie
DTU
Lyngby, Denmark
corentin.serrie@student.ecp.fr

INTRODUCTION

In the field of Video Games, characters have to be animated and react at runtime, depending on the player's inputs, and on the context of the level actually being played. For example, a humanoid character going forward might have to play a stumble animation when hitting an object on the floor. Whereas the traditional way of animating characters using state graph used to be very fixed and relied on the animators thinking of every possible outcome, and every possible transition from on state to the other, newer technologies like Motion Matching or Inverse Kinematic are making it easier and easier to have humanoid characters react to their environment at runtime with believable animations. However, for everything – character or not – that is non-humanoid, there is still a lot of work to be done, and very few researches available.

This paper will explain the process that led to a non-humanoid, blob-shaped character to be animated at runtime, using the Unity physic engine. This character should be able to move in his environment, and react like a balloon, would. This includes marrying the shapes of his environment and getting around hard edges. Additionally, this character should be able to jump and land, using the classical "squash and stretch" technic often used in 2D animation.

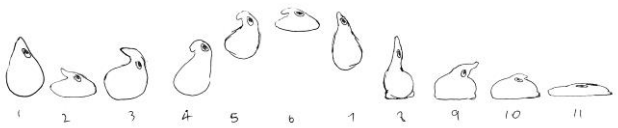

Figure 1 : example of the animation of a 2D character using squash and stretch to make it more believable.
Image: Carol Amadio on WordPress

METHOD

As I will discuss in the results section, over the course of the project, I tested two main ideas:

- Reshaping a single collider when squashing and stretching
- Using multiple smaller colliders linked together with spring joints, creating a global shape that has the desired geometry.

In hindsight, the second one is definitely the better one, and as we will see, led to better than expected results. However, I will still discuss the first one, as it was a valuable learning experience, and seemed to be the natural solution at the beginning of the project.

IMPLEMENTATION

I won't go into details regarding the actual implementation that I chose for this project, the code is available on Git [1] if needed, and the reader could get to some very similar results with a totally different implementation. I will however clarify one step of the process: the "animation". Indeed, once you have created your global shape with multiple colliders and linked them with spring joints (all of which could be done in the Unity editor without using a single line of code), you still need to modify your joints at runtime to create the desired animation. Otherwise, you are left with a spherical blob, that reacts to its environment correctly (i.e. deforms when encountering another collider and isn't perfectly spherical on the floor, due to gravity), but doesn't jump with a squash and stretch effect.

To achieve this effect, my method was to first compute where every collider should be in the void given a squash factor, and then change the joints accordingly, so that in the void, and given enough time, the overall structure would end up resting in the desired position.

To do that, I recorded the initial position P_i of every spherical collider in a referential centered around the center of the blob, and decomposed this vector into P_{up} and P_{plane}, its two component along the vertical axis, and on the horizontal plane.

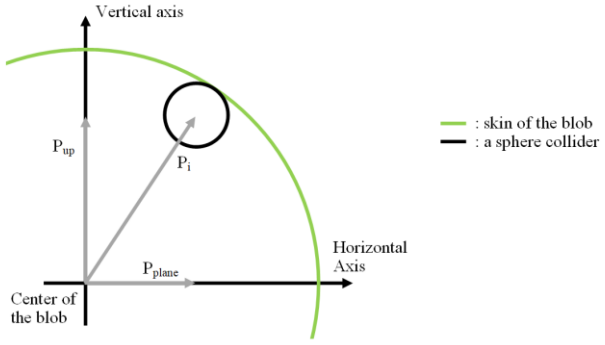

Figure 2: schema explaining the referential and the different vector used

Then, at runtime, the desired position of a sphere with initial position P_i would be

$$P_d = (1 - \text{squash}) * P_{up} + (1 + \text{squash} * 0.75) * P_{plane}$$

With squash being a ratio between 0 and 1, detailing how squashed the blob should be – 1 when it is completely squashed, and 0 when it isn't. The resulting animation feels great, but the 0.75 multiplicating factor can always be changed if you want the squashed blob to be larger.

RESULTS

This project came from the idea of having a ball as a main character in a video game, that would squash when pressing a button, and jump when releasing it. The player would then maneuver this character in a 3D environment to get to his goal at each level, in short, the game would be a 3D platformer. I was using Unity from the start, and having recently created a 2D platformer, I thought that I could use the same basis of code for the movement, extend it to have it work in a 3D environment, add a scaling method that would reduce the scale of my sphere on the Y axis while augmenting it on the X and Z axis to simulate the squashing effect, and that it would be enough for my purpose.

My first problem was that the unity default sphere collider can't be rescaled to an oval form: it always keeps a spherical shape. This lead to a desynchronization between the spherical mesh that was rendered, that was actually being reshaped, and the collider that was used to simulate the physic, that was not.

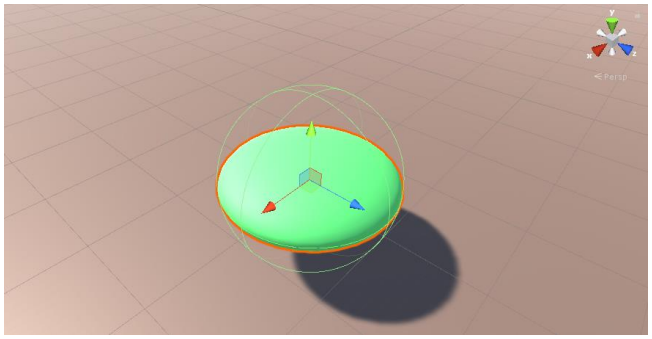

Figure 3: desynchronization between the mesh renderer (plain green), and the collider (green lines).

I solved this problem by using a custom collider, more akin to a polyhedron, that could be rescaled by the Unity engine.

Another mistake was to assume that the platformer motor (the code handling the movement) would work using the same principle in 2D and 3D: whereas it is mostly recognized that you shouldn't use the Unity physic engine to create a 2D platformer motor [2], there is no real consensus when it comes to 3D, and using the same Raycast approach I had used for my previous 2D platformer lead to some issues, notably the character would get stuck when encountering a stair less than $1/10^{th}$ its height, and treat it like a solid wall, which feels bad for the player.

By changing the characters collider, I solved more and more of these small issues, but the more time went on, the more it seemed apparent that I was losing lots of time treating edge cases, and that whatever the collider I ended up choosing, and the code I would end up writing to prevent these edge cases, I couldn't think about all of them, and a player would inevitably find one that I wouldn't have thought of.

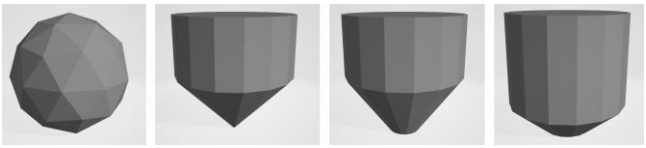

Figure 4: different collider used to solve more and more edge cases.

During this iterative process, I also changed back and forth between a platformer motor using the built-in physic engine, and the original one using Raycasting, each time improving it a bit, but never really got to a clean version that would have no known bug.

As it turns out there is a reason why the defaults colliders can't get rescaled: rescaling a collider is actually not recommended, and can lead, at best to objects being slightly teleported when rescaling a collider leads to a collision, and at worse to them clipping through some walls, or even the floor, and falling indefinitely.

Around that point, I came upon a video explaining how the developers of Snake Pass made their snake's movement feel realistic using multiple sphere collider linked together with joints [3].

Figure 5: the snake's body using multiple sphere colliders in Snake Pass
Image: How Snake Pass Works | Game Maker's Toolkit

This totally changed my perspective on things, and I instantly started working on a version where the blob would actually be composed of 18 sphere colliders linked together with spring joints.

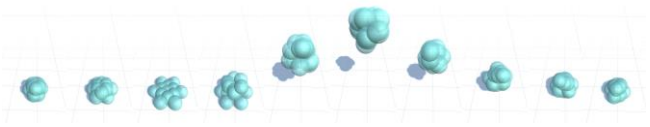

Figure 6: Example of a jump of a blob made of 18 spheres linked with spring joints

The results were promising, so I kept on fiddling with the different joints available in Unity, switching to an

Icosahedron shape, and eventually making it a modulable parameter where you could change the number of sphere per row, and the number of rows.

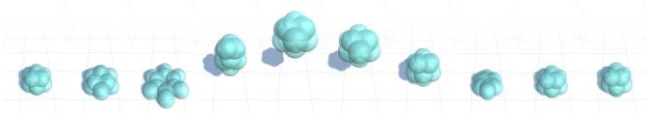

Figure 7: jump of a blob with an icosahedron structure

Each sphere would have a joint linking it to its following neighbor in the same row, and to one or two spheres in the row above. This allowed me to test how many spheres it was possible to simulate before experiencing to much lag. To my great satisfaction, I was able to simulate 15x15 spheres on a phone without experiencing any lag.

The last important change regarding the physics was to add a central sphere that is connected to every other sphere collider with a spring joint, and that has a mass equal to the sum of the mass of every other sphere. This helps stabilize the blob around a center, and because we apply the forces used to make the blob move only to this "core" sphere, the other one act as a skin that will protect the core. Moreover, because this core sphere is nearly never in direct contact with any collider (all these spheres don't collide with each other), when it actually is, you know that an exceptional event is happening: either the blob just landed from a huge fall and should take damage, or it is being crushed by a platform coming from above…

Finally, and because the player wants to play a coherent character, and not a heap of spheres, a mesh is created and updated at runtime to encapsulate the resulting general shape, simply by having one vertex for each sphere composing the "skin" of the blob (i.e. every collider except the core).

The resulting blob is way better than expected, because on top of having a jump that feels very good to play with, using the squash and stretch principles discussed in introduction, the resulting shape adapts to basically every geometry in the level.

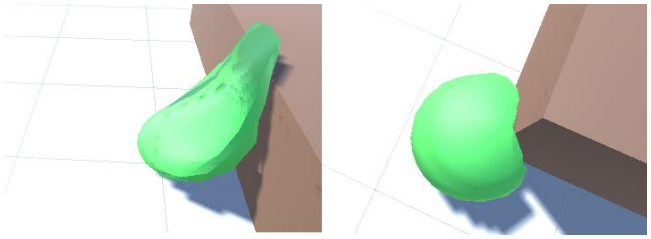

Figure 8: Example of the blob adapting to the geometry of its environment

CONCLUSION

To sum it up, the method that brought the most success used the physic engine of Unity, and can be boiled down to four main points:

- Using multiple colliders to create a global shape looking like the one you desire
- Linking these colliders together with spring joints, whose force you have to adjust to get a good feeling
- Changing the position and anchor points of these joints a runtime to create, in a void, the animation that you want to play, and let the environment interaction and the gravity interact with the resulting mass of colliders.
- Creating a mesh that moves according to the position of the different colliders.

This process has some similarities with the classical process of animating 3D character, mainly the last part. Indeed, traditionally, you would first create a 3D mesh of your character, then rig it, i.e. associate each point of the mesh to some bones, and then animate it by making only the bones move. The rigging process is here to make sure that the different vertices will react in a good way when the bones are moved, and to make the animation process easier, since you now only have a few bones (traditionally 17 for a humanoid character) to move, instead of the thousands of vertices that compose a 3D model. Regarding our process, we could go so far as to call the last point rigging, and we could therefore think of using this process to animate more complicated characters than a simple blob.

I think that by creating a body made of one collider per bone, and using hinge joints at each articulation, it might be possible to animate a humanoid character in the same way, and have it interact realistically with its environment. The resulting character would be animated in the void using the classical data from the animators, processed to get the angle to apply to the hinge joint, but the fact that these joint have a spring factor would make every contact with any other object affect the character, in a way that is close to the original animation, but still account for the environment.

But before even going there, this model can already be used in this form, as the main character of a video game, as is my intent, just by texturing the mesh. Other uses could include simulating bubbles, an object that is really often used in water levels, or balloons, or even a regular ball by increasing the force in every spring of the blob, although this would be very overkill for an object that stays mostly a sphere.

REFERENCES

1. https://github.com/Aberduc/Blob
2. http://www.gamasutra.com/blogs/YoannPignole/20131010/202080/The_hobbyist_coder_1_2D_platformer_controller.php
3. https://www.youtube.com/watch?v=4NNPr2Ay4OM