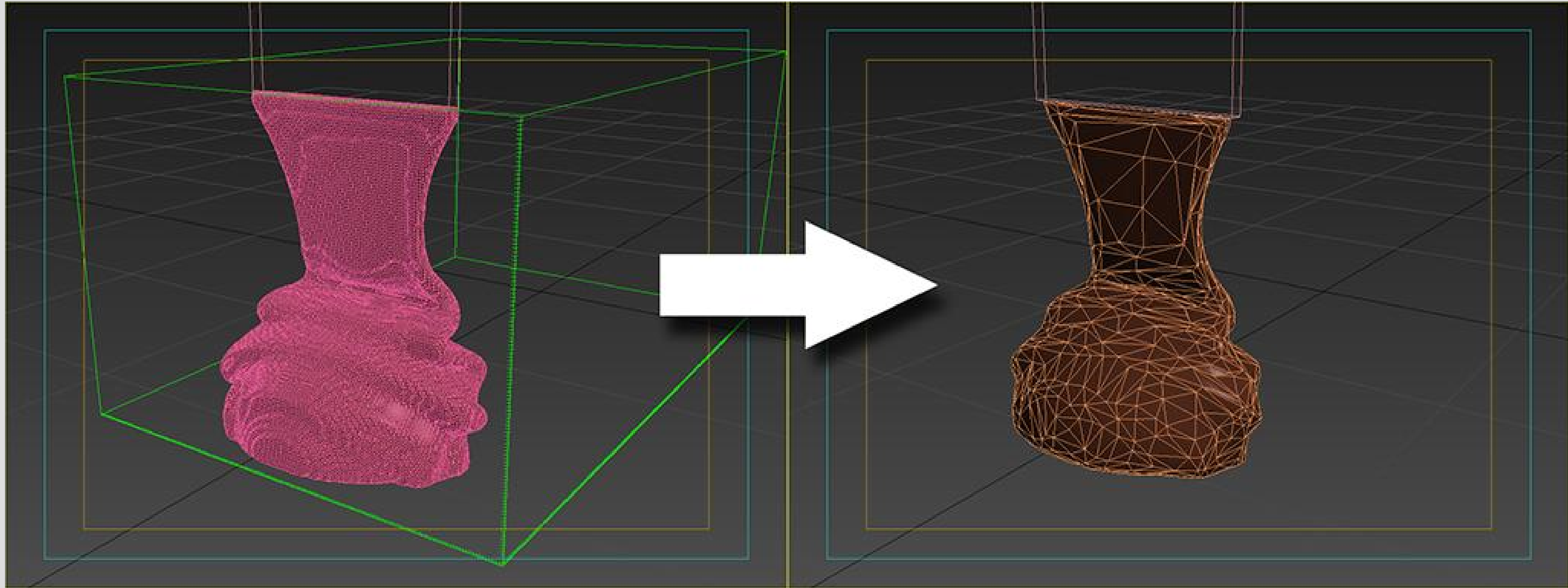
Vertex-count-agnostic Morph Targets

by Norman Schaar

The Goal



Optimize a high detailed pre-calculated fluid simulation for use in game.

The Problem:

- The calculated simulation is a sequence of meshes, each mesh differs in triangle count and vert count.
- Each frame would require the same amount of triangles AND verts for morphing to be possible.

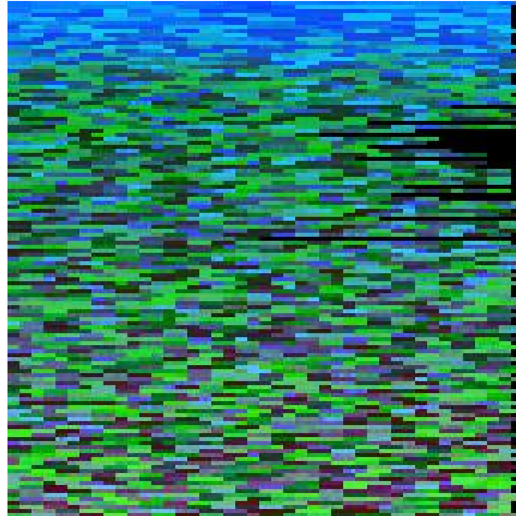
Textures in Vertex Shader:

- DirectX 11 compatible graphic cards allow textures to be used in vertex shader. In other words: we can manipulate verts with textures.

- We can store pretty much any vert attribute in a pixel

- A Look Up Table texture can be created with multiple vert positions and normals

The Texture:



This is a cropped portion of the vert position texture.

The Texture:

- Every pixel represents a vert position. Three pixels represent one triangle.
- If three pixels are black it means that that triangle is not needed for that frame and so its vert positions will be 0,0,0. The triangle will virtually cease to exist
- Each row of pixels is a morph target

Step #1: Optimize Meshes

- Decide a target face count for your meshes. The goal is to have roughly the same amount of faces for every mesh. Note: we don’t need to have the exact same face count or vert count.

- Optimizers typically don’t allow you to specify a target face count. They allow you to control vert percentage or vert count. I had to write a script that tries out different vert counts until it reaches the desired target face count.

The reason why the face count is important is because we will eventually split every geometry vert. Our final vert count = face count * 3

Step #1: Optimize Meshes (optional)

Before running the optimization through every mesh you can:

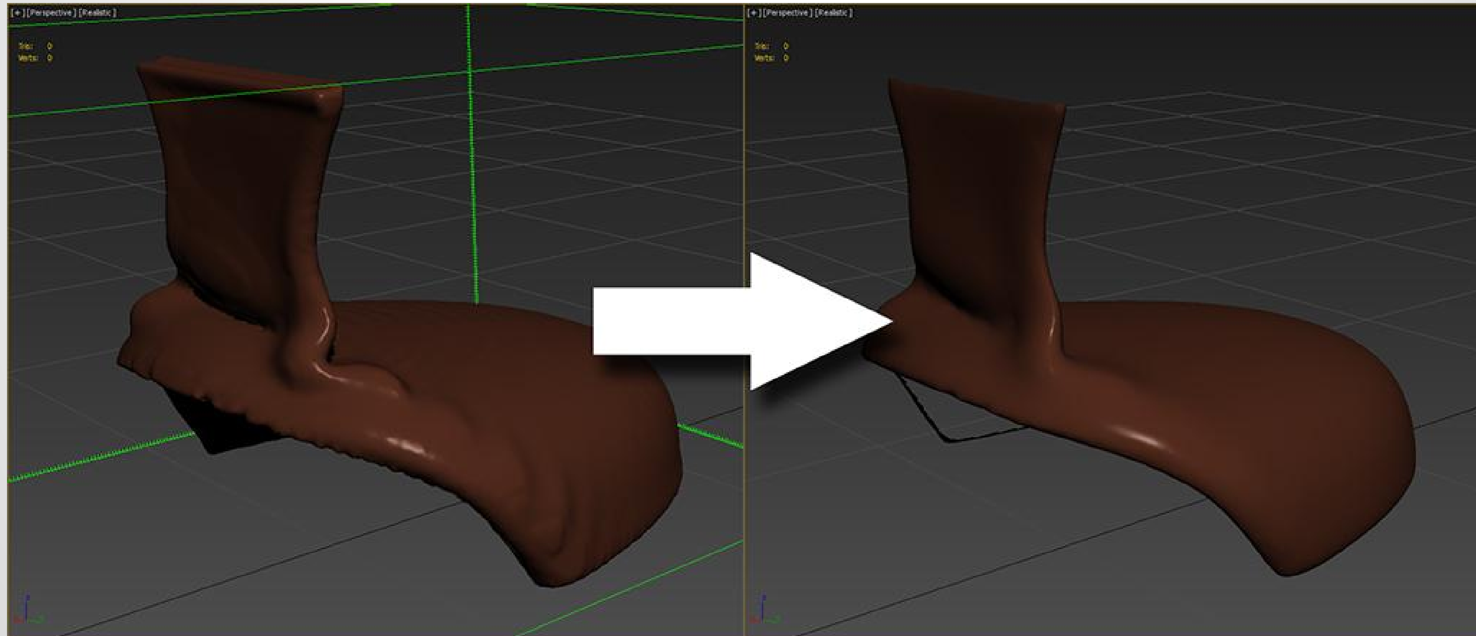
- Use Volume Select modifier in 3ds Max to select faces that are making contact with surfaces and are not seen by the camera. Add a Mesh Delete modifier to delete the selection.

- Add a relax modifier to soften the topology. In my case I created the simulation with a huge grid as I had no time to calculate a more detailed simulation.

- I also added a Relax modifier after the optimization was done. 3dsMax ProOptimizer creates very thin triangles at time which cause shading anomalies.

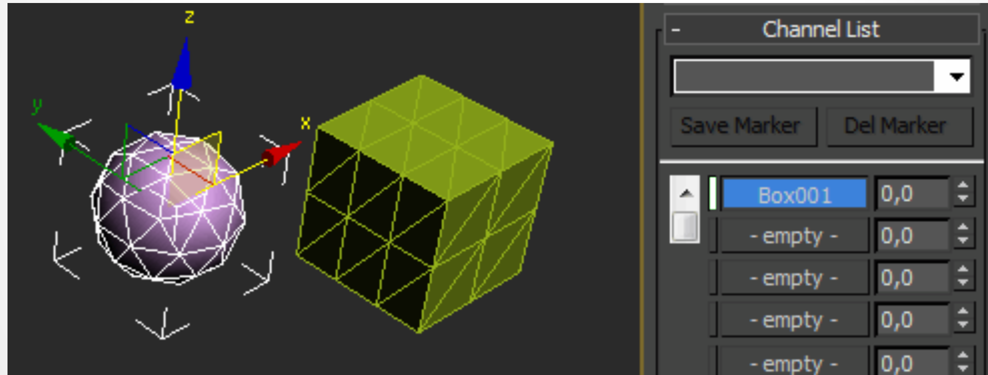
Step #1: Optimize Meshes (optional)

This is the result after Relax and Mesh Delete modifiers:



Step #2: Reorder verts

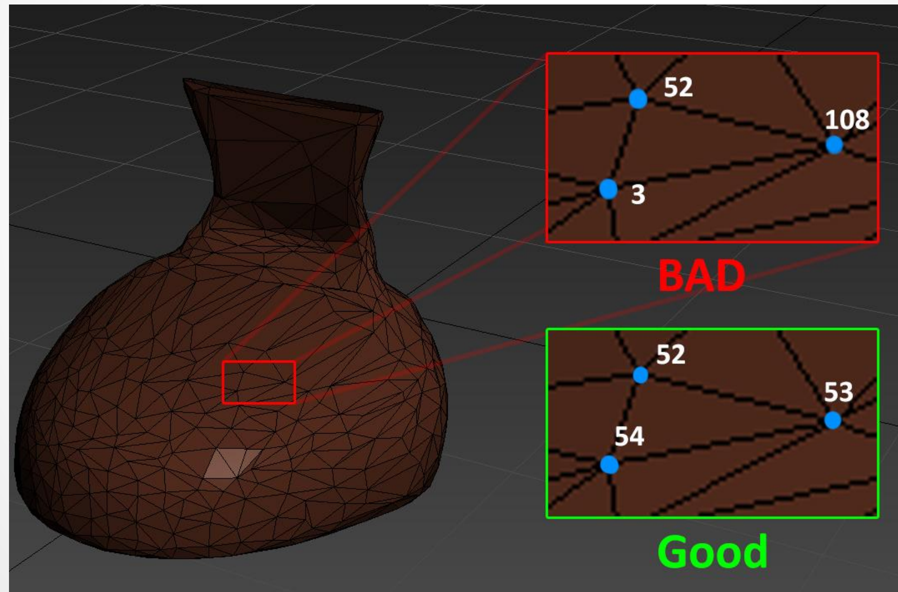
Even if your meshes had the same vert count and face count morphing would not work right of the bat. This is what will happen if you attempt to morph:



Why is this happening? Let’s take a closer look at the geometry...

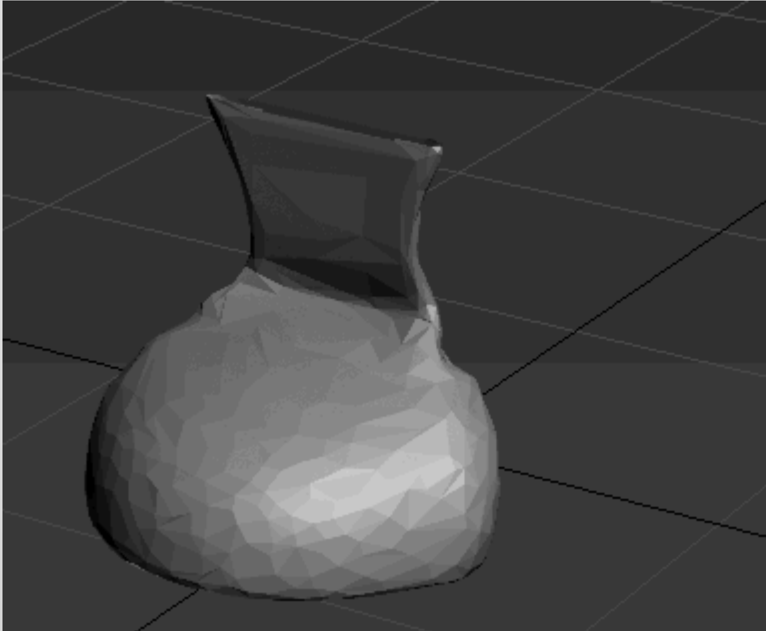
Step #2: Reorder verts

Even if your mesh has the same face count and vert count, chances are that the vert index will be all over the place.



Step #2: Reorder verts

To fix this I wrote a script that will automatically give each triangle’s verts a sequential vert ID. And this is how morphing will look:

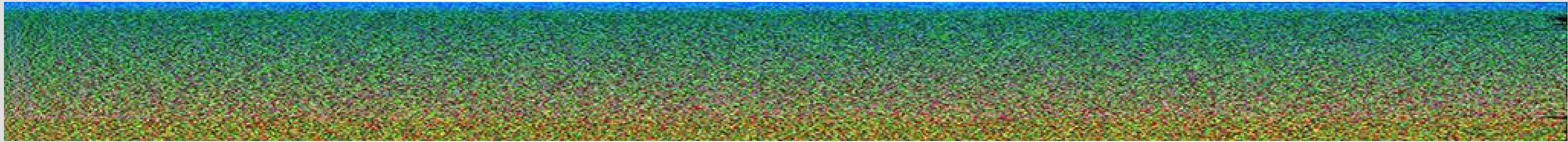


This is the expected and desired behaviour. Don’t expect a smooth transition between the frames. There is simply no data to make this a smooth transition. Think about it, if you wanted to interpolate the fluid simulation you would have to re-sim it.

One thing that I need to look into is reordering the verts based on triangle distance between frames. This could result in more organic (but still no way correct) transitions. I will try it out just to see what transitions look like with different vert ID sorting.

Step #3: Render Position Texture

- The width of the image will be determined by the maximum vert count in your meshes. Remember vert count = face count *3
- The height of the image is determined by the frames of the simulation



In this example we have a 3390 verts (1130 faces) and 300 frames. This texture has less pixels than a 1024x1024 texture.

Sadly the texture needs to be uncompressed since compression will alter the data too aggressively.

Step #3: Render Position Texture

- Keep in mind that even though the texture is uncompressed, RGB is low precision.
  Vert positions are usually highp. In order to alleviate the pain of lowering the precision, the data is packed per channel to make use of the entire 0 to 1 range.

  The script outputs two Point 3 values that you need to multiply and add to the texture in the shader to un pack the data again.

- Graphic cards nowadays support non power of two textures. But is there a penalty? Are non power of two textures harder to fit in the texture pool?
  If anyone knows please let me know.

  If it were a problem, the pixels could also be rendered in a block like fashion. Like a flipbook texture. But it would result in some unused pixels as you would rarely ever use 100% of a square texture. I’ve also created a tool that would help you decide the proper dimensions of these blocks taking into account frame and vert count.

Step #4: Render Normal Texture

The normal data is packed from -1 to 1 range to a 0 to 1 range.

Exact same process as the position texture, but instead we save the normal data for every vert.

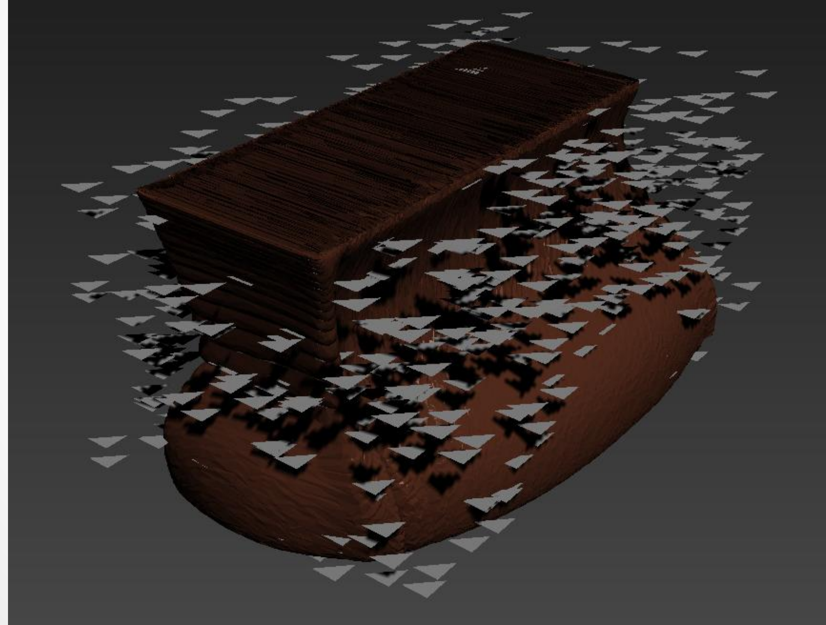
This process takes a bit more time because we can't simply use the vert's normal. Keep in mind that the verts were split in the process and so the mesh has now a faceted look and many more verts! We do not want that normal.

For a smooth finish you need to have a version of the mesh with the welded verts and a single smoothing group, so that's less verts!

The script has to go find the corresponding vert of the smoothed mesh and retrieve its normal.

Step #5: Generate Mesh

The mesh is a series of floating triangles that reside within the bounding box of the complete simulation.



Step #5: Generate Mesh

Why are they floating?

When offsetting verts in a mesh through a shader the bounding box of the object is never updated. You can essentially have geometry outside of the objects bounding box. But the CPU won't know about it.

Many engines will hide the meshes once they are outside of the camera frustum and the way they determine this is by checking against the object's bounding box.

This is why I have the triangles occupy the space of the bounding box. If I had them all with Z=0 the bounding box would be too different from the deformed mesh.

Why are they not stacked or overlapped?

If they are too close you run the risk that your engine will automatically weld the verts, like Unreal 4 does.

Step #5: Generate Mesh

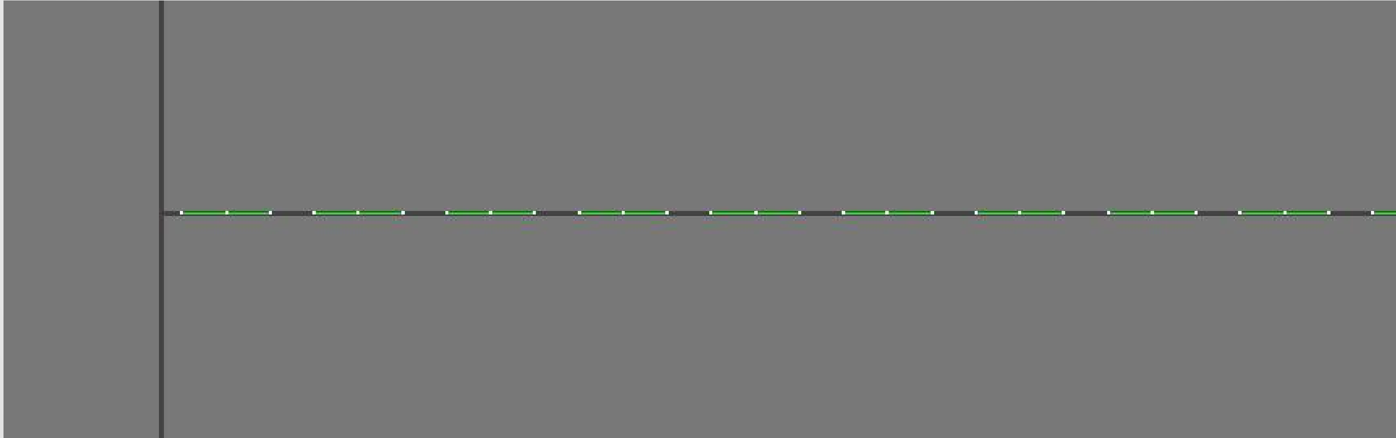
Why are they pointing up?

By having the triangles point up the normal is [0,0,1]. Which is a blank slate essentially.
If I had to deal with any other normal direction it would mean that I would have to compensate for it with my low precision normal texture with a the normal’s difference. Chances are that the low precision texture won’t be enough to compensate for the high precision normal data.

For the same reason I save the world position of the verts instead of a difference.

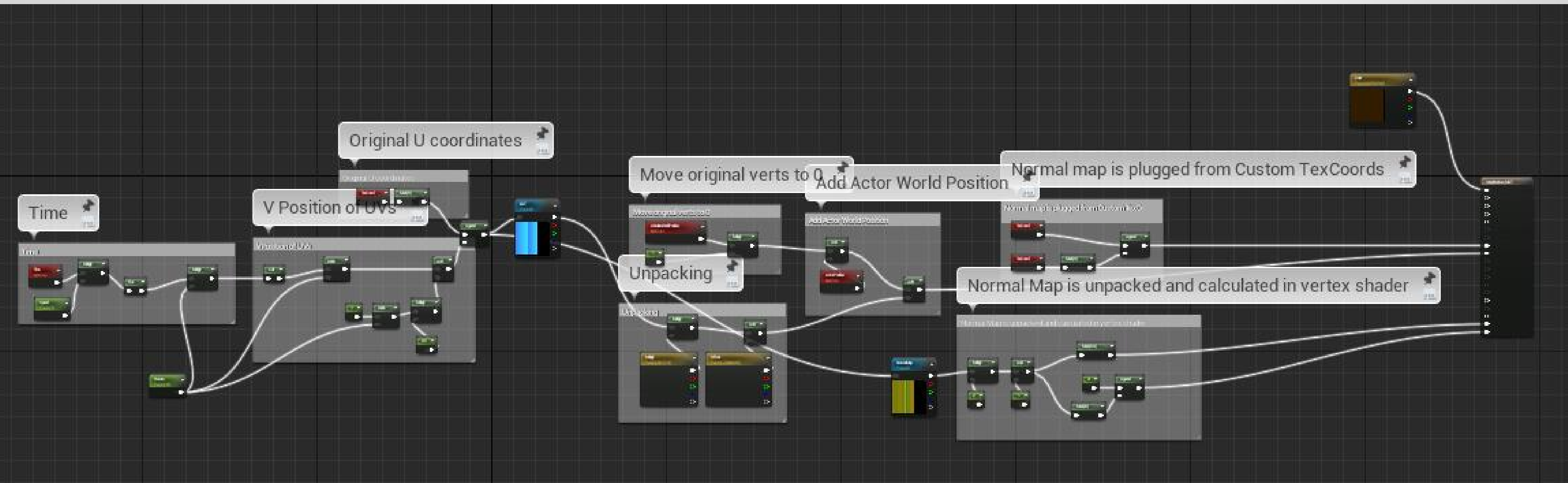
Step #5: Generate UVs

- The UVs are ordered in a row and centered in the pixel. This is all done through script as well
- The V position in this example does not matter as we will fully determine this value in the shader



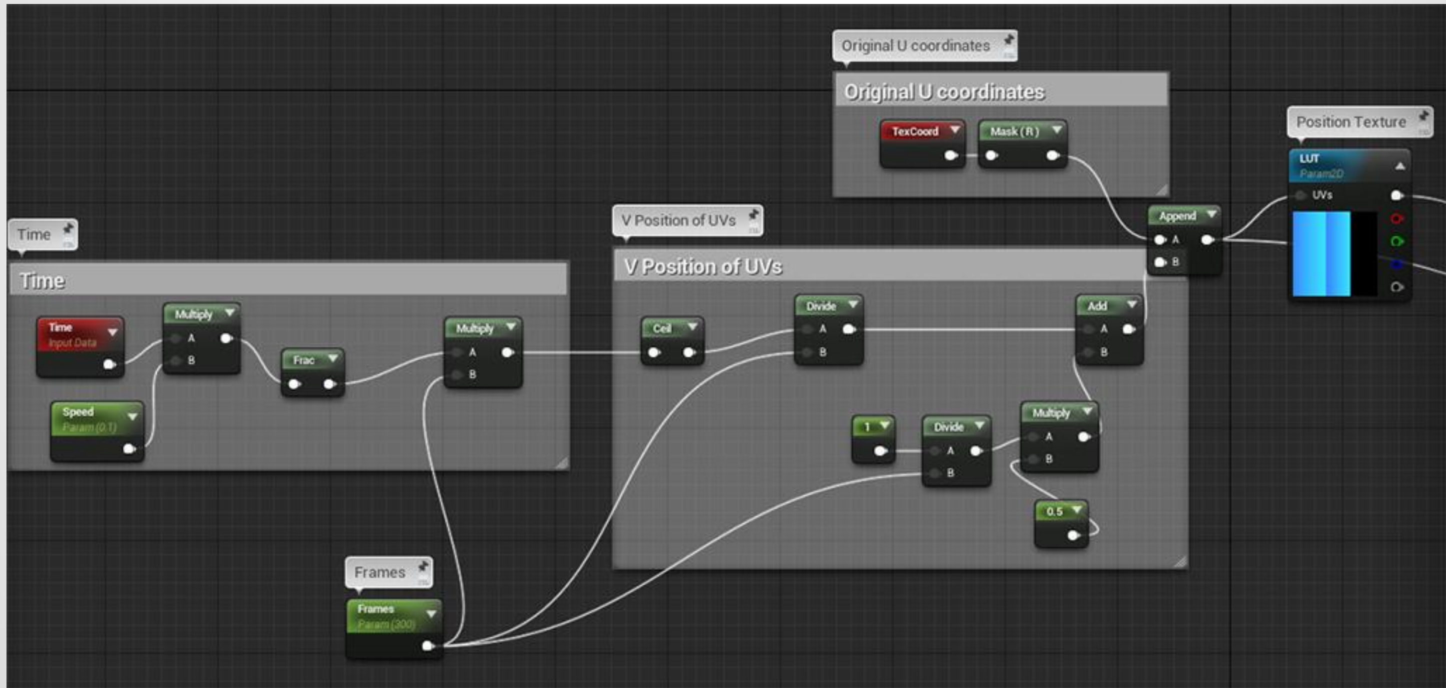
Step #6: The Material

Here is an overview:



Step #6: The Material

Here is an overview of the UVs:



Step #6: The Material

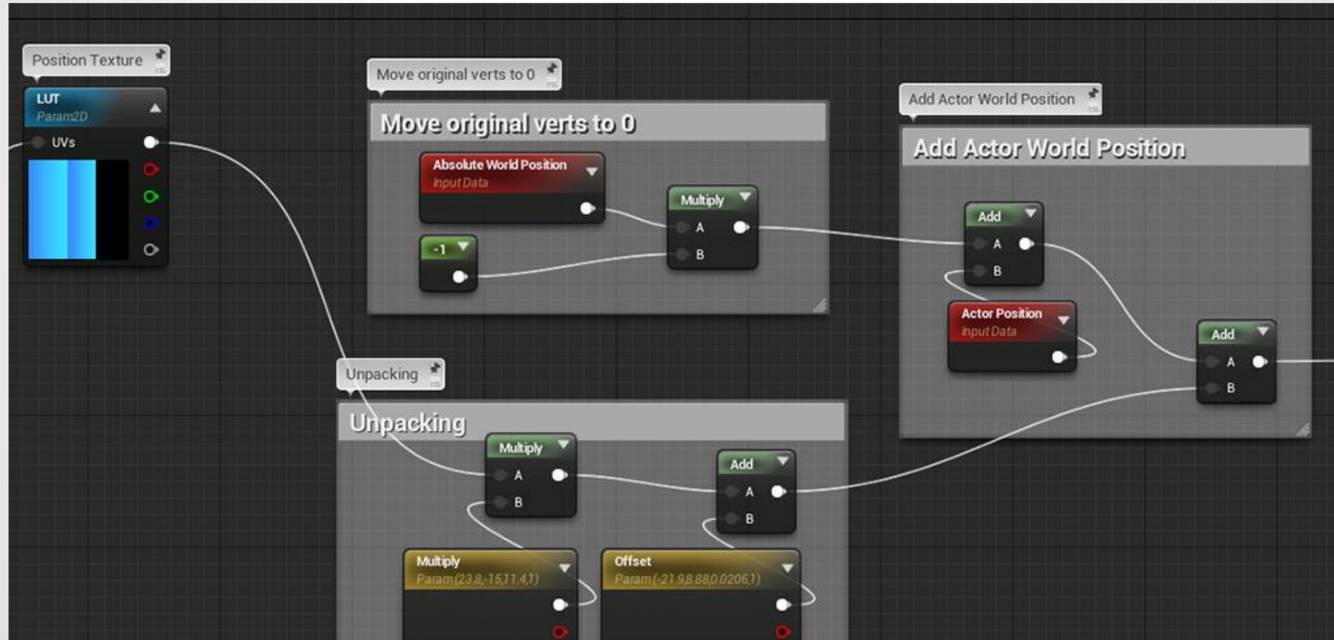
- We put the Time node through a frac and take the number of frames into account so that the UVs jump to every row of pixels instead of interpolating between them since this would generate incorrect mesh interpolation.

- The V coordinate is also placed in the row’s center.

- As you can see the U coordinates are left untouched

Step #6: The Material

Here is an overview of the Position Data:

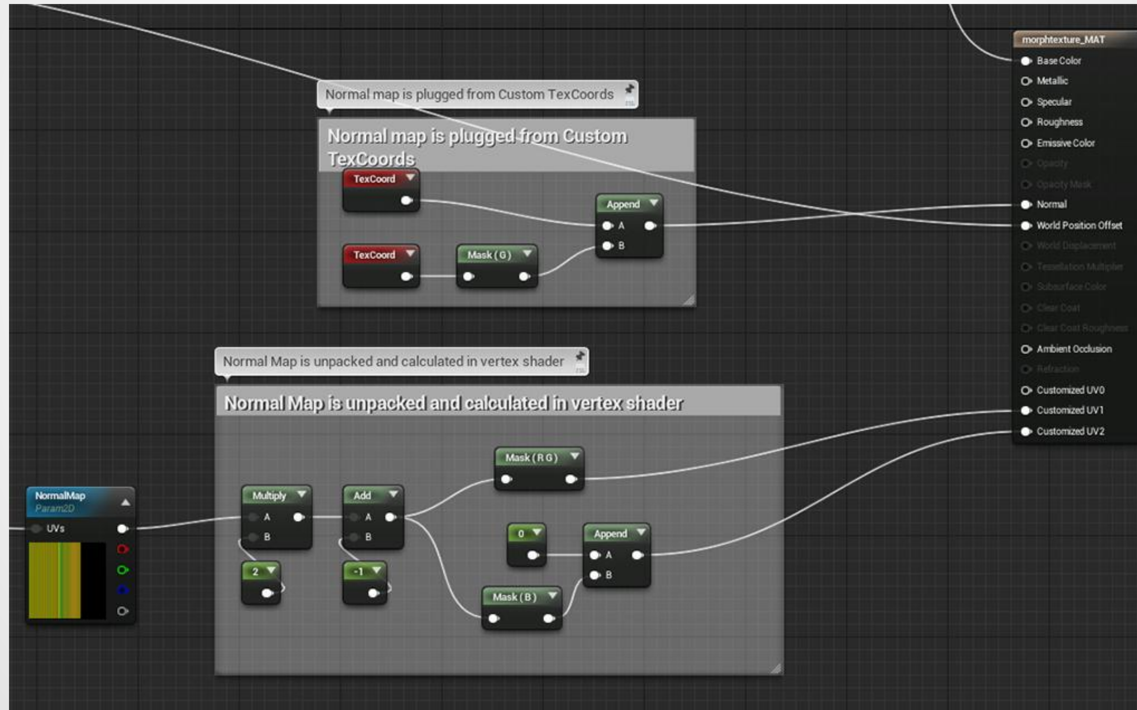


Step #6: The Material

- The verts of the mesh are moved to [0,0,0].
- Our data is unpacked with the vec3 values that the script provided.
- The actor world position is added to our data so that the geometry follows our actor around instead of sticking to [0,0,0] when the actor is moved.

Step #6: The Material

Here is an overview of the Normal Data:

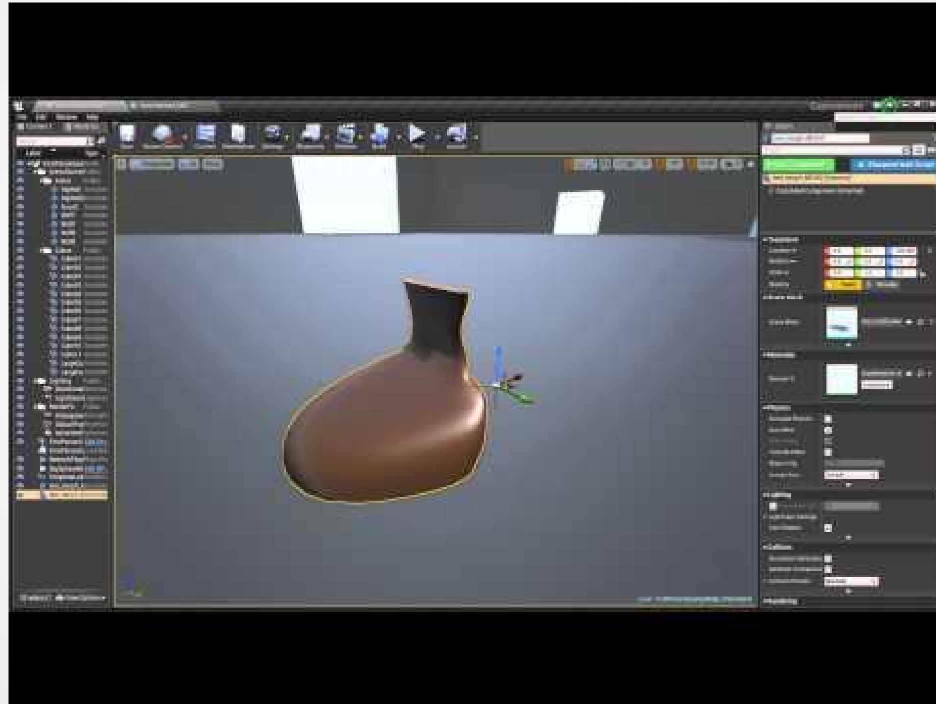


Step #6: The Material

- The normal data is unpacked.
- The normals need to be calculated in the vertex shader. Otherwise they will interpolate incorrectly.

  The only way I found to force certain calculations to happen in the vertex shader in Unreal Engine is to create “Custom UVs”. These are meant for mobile devices, where calculating UVs in the vertex shader prevents dependent texture reads.
  We have to pretend our normal data are in fact UVs. I intentionally leave Custom UV0 unplugged since UV0 needs to be left untouched. Custom UVs override regular UVs calculated in the fragment shader.

The Result



What's next?

- Save other vert attributes and simulation properties
  - UVs
  - Thickness
  - Color distribution
- Create the UI for the maxscript tools
- Research color sorting algorithms to test different interpolations between frames
- How does this apply to:
  - Cloth tearing apart
  - Particles

Special thanks to

Matt Vitalone, Francisco García-Obledo Ordóñez, Hanno Hinkelbein and the Real Time VFX Facebook group!

Contact Info:

Email: Norman3D@gmail.com

Twitter: @Norman3D