

Deformation Layering in Maya's Parallel GPU World



Charles Wardlaw

Nov 25, 2017 · 8 min read

In TD chat the other day the topic of deformation layering came up, and I realized I hadn't formalized anything I've worked on in the past year or so. This article will be about better modern practices for deformation layering in Maya 2017+. I'll be using the wonderful [Hippydrome.com mesh](http://Hippydrome.com) by Brian Tindall. Special thanks to Brian for putting that mesh out into the world.

What is Deformation Layering?

Deformation layering is an old technique; in fact, I was taught how to do it on my first job in the industry, but didn't truly internalize it for a long time. Taking two identical meshes, you design the deformation such that one deformer's output is passed downstream to a second deformer, which then deforms the modified points of the first. In most cases this means making two meshes with two separate deformers, then using a blendShape to blend the output of one deformer onto the input of the other. (More advanced users will immediately recognize that they could skip the blendShape node, opting instead for a source_meshShape.outMesh -> target_meshShapeOrig.inMesh connection, but as we'll see neither technique will work for our use case.) This is different than using a blendShape node to blend deformations all at once—the intent here is for the deformations to change the mesh *sequentially*, not in parallel.

That was a mouthful — how about some images?

I thought that a cartoon eye is the simplest way to explain the technique, so I made two layers (two meshes): one with a bone for upper and lower lids (LAYER_0), and one with a single bone for moving and scaling the eye as a unit (LAYER_1). The goal here is to chain the output of LAYER_0 so that the blink happens first, then the mover control can move, scale, or twist the eye as needed, leaving the blink operational.



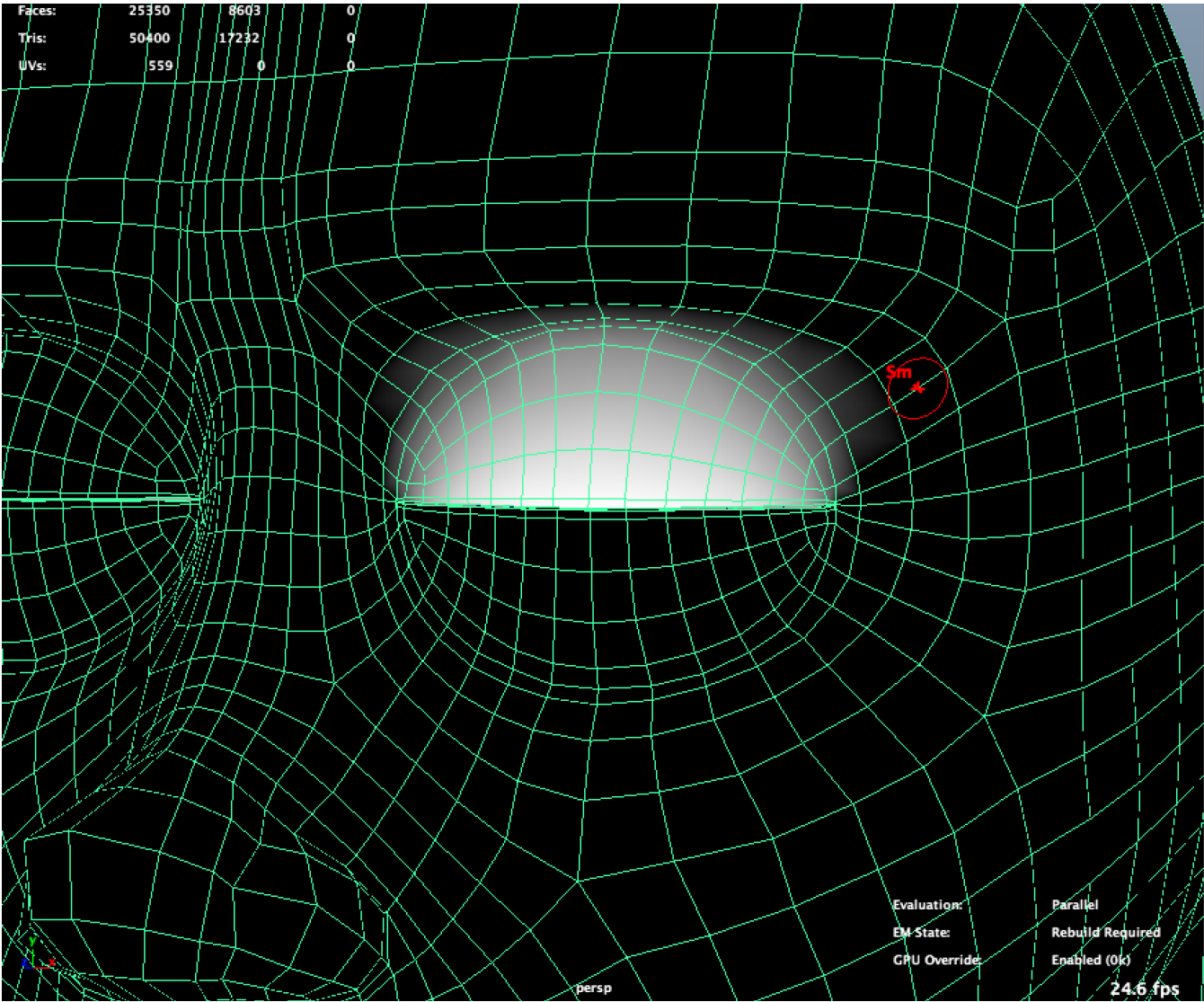
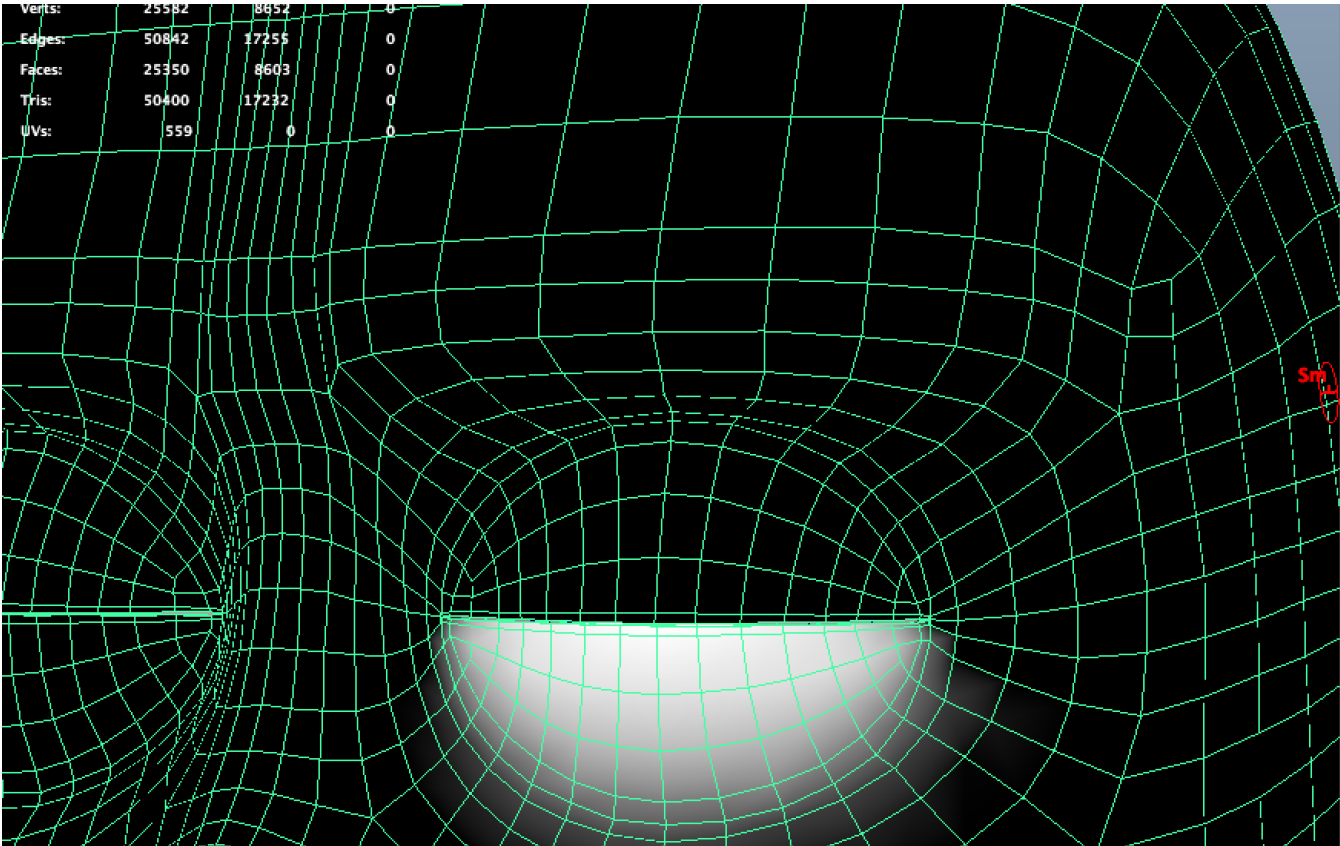


Figure 1a: lid upper weights



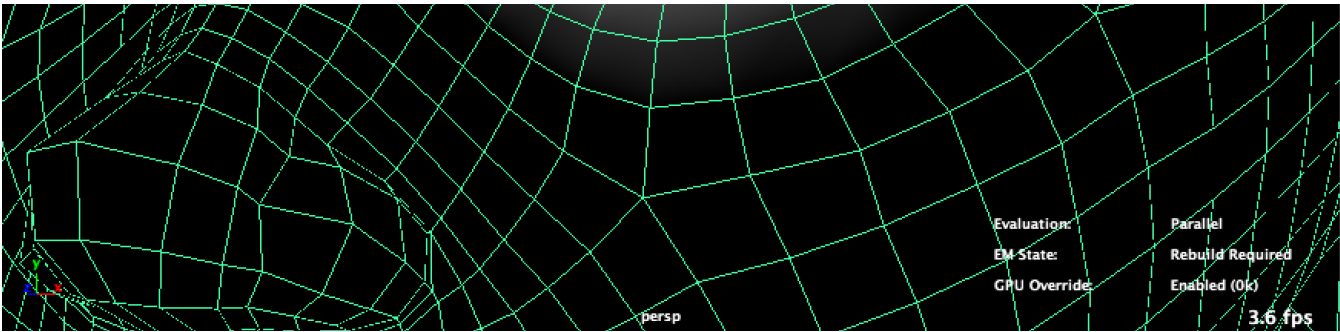


Figure 1b: lid lower weights

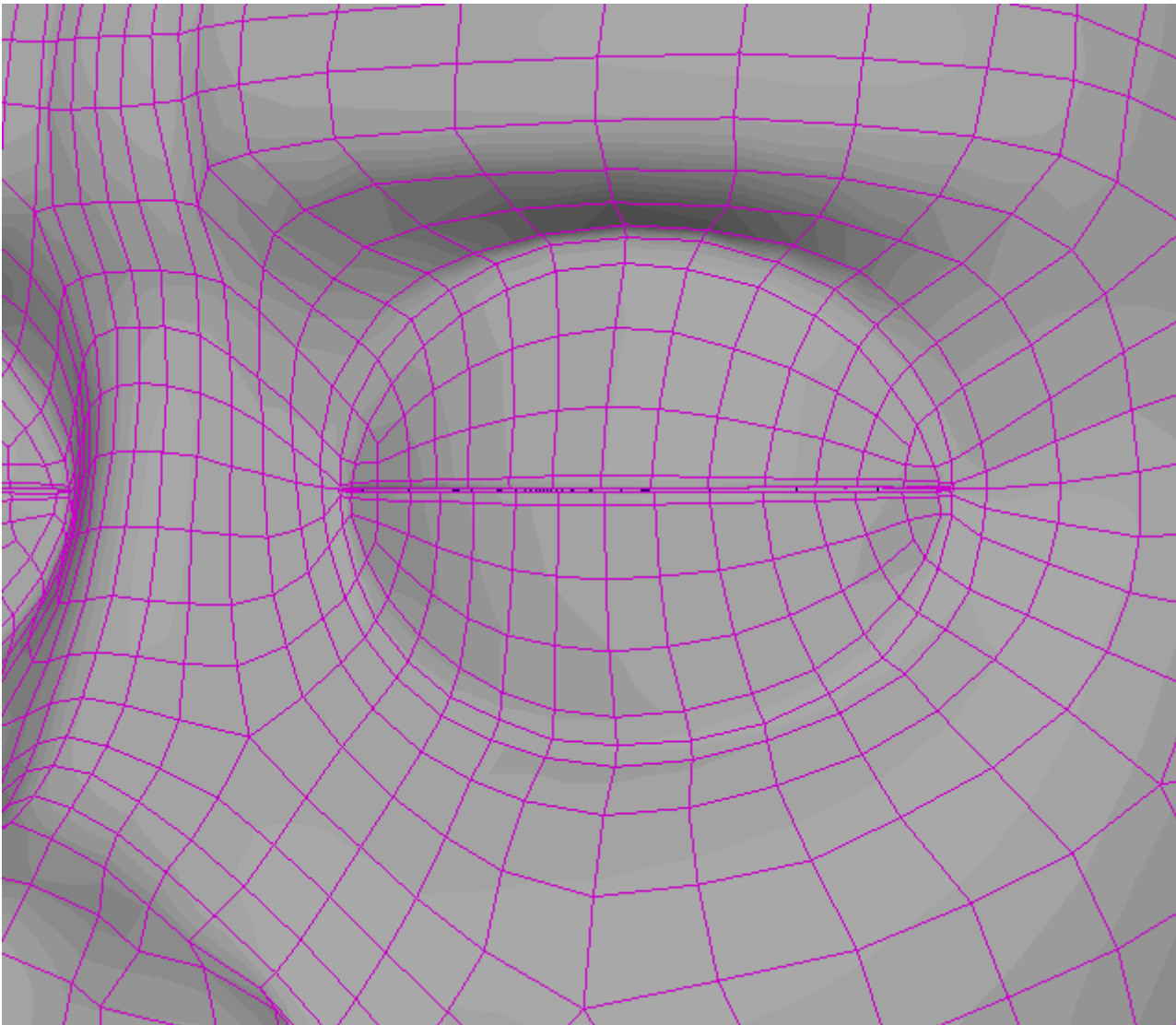
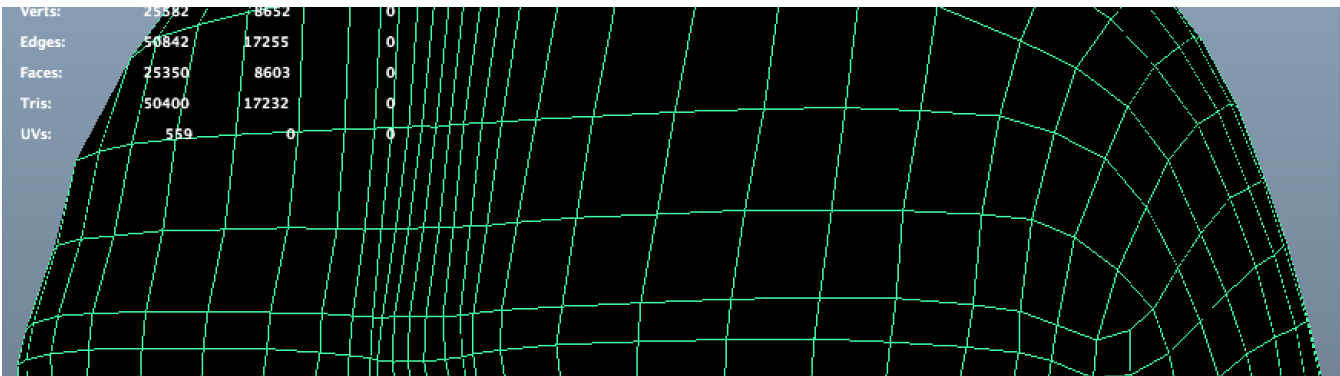


Figure 1c: eye blink (animated)



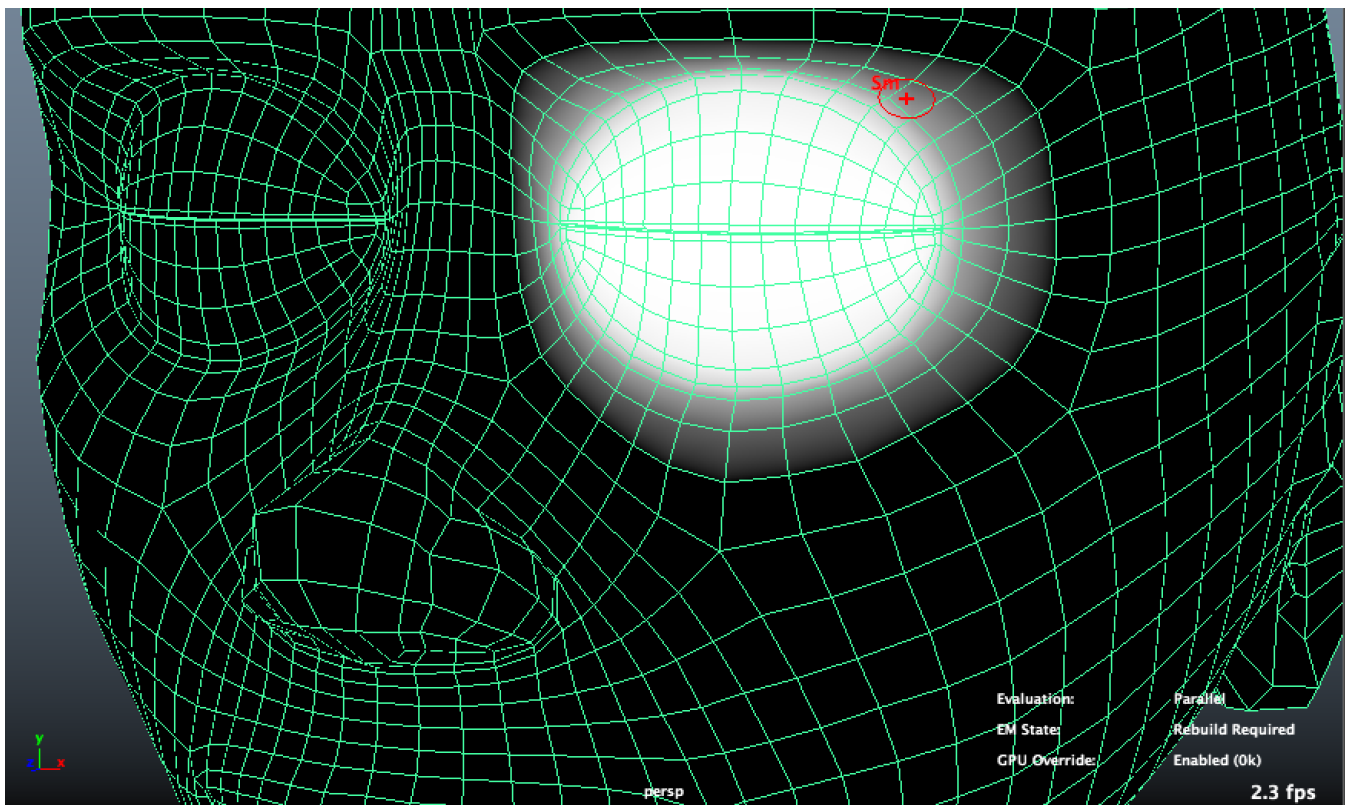


Figure 2a: eye mover weights

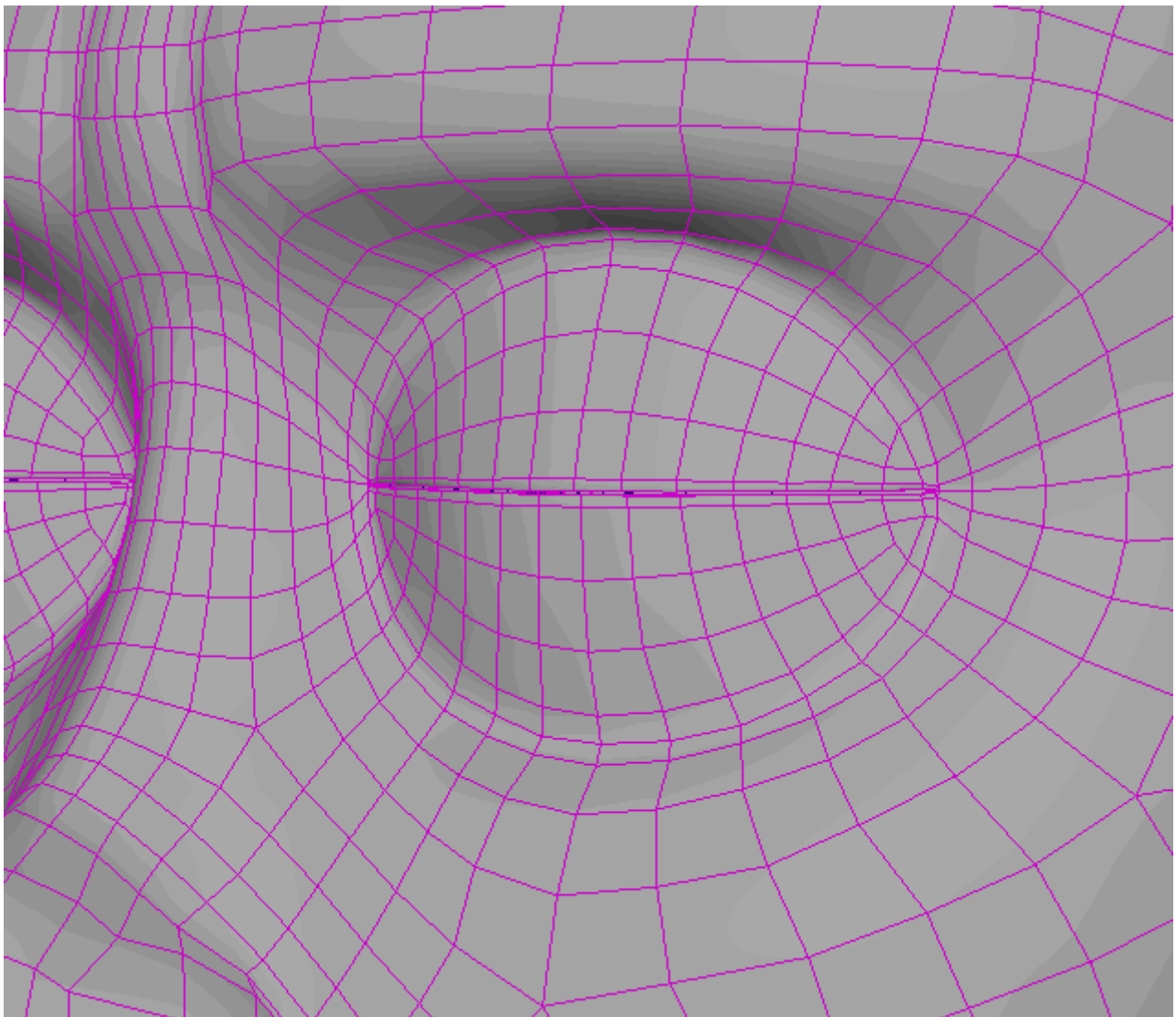


Figure 2b: eye mover gif

And for completeness, how about our end result? The weighting was a five minute job for the article, but dat animation... I think I have a career in feature.

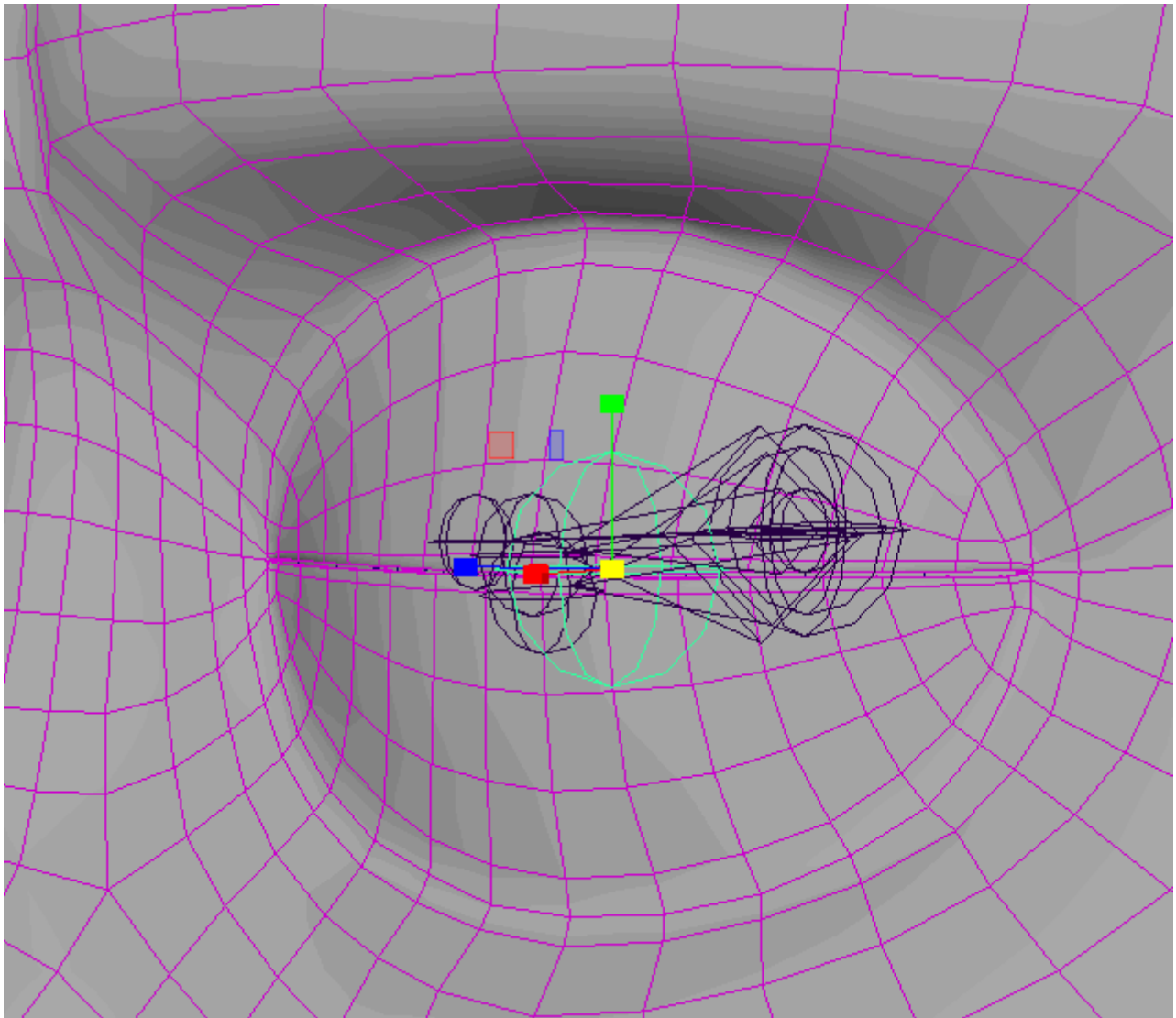


Figure 3: the complete effect of two layers working in serial.

Notice that there is no breakage as I blink through the twisted second layer.

Connecting the two meshes to achieve this can be easily done. First instinct for most would be to use the blendShape method or outMesh -> inMesh technique to do this chaining, but we hit a problem that's unique to more recent versions of Maya: those kinds of connections will stop a portion of the deform chain from being able to evaluate using GPU deformer.

Autodesk did a good job with their GPU overrides. If a deformer supports GPU evaluation and very importantly *if the deformation chain is GPU friendly*, Maya will move the mesh onto the GPU and use the GPU to do deformations, increasing your

frame rate significantly in many cases. Unfortunately, what constitutes a “GPU friendly deformation chain” isn’t readily apparent. For starters, you have to limit your rigs to the deformers with GPU overrides, and even in Maya 2018 not every deformer is GPU capable. (That’s right — No wires or wraps.) But more damning is that *you absolutely cannot chain animated meshes as sources for GPU deformers*.

What does this mean? If you have a blendShape with a live, animated target, that target mesh and the deformers that feed it will run on the CPU. If you plug the outMesh of a mesh with a skinCluster on it into the shapeOrig.inMesh of another deforming mesh, that first mesh will be CPU only.

Here’s a demonstration. In the first image the two meshes are not connected, and all 17k points are on the GPU. Connecting the outMesh of the first to the inMesh of the next, we see that number cut in half.

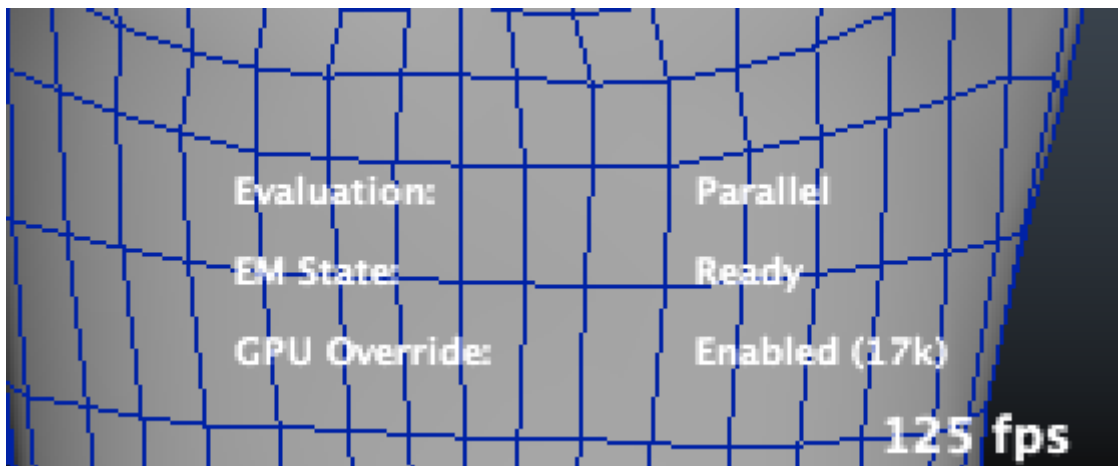


Figure 3a: GPU point count before connection

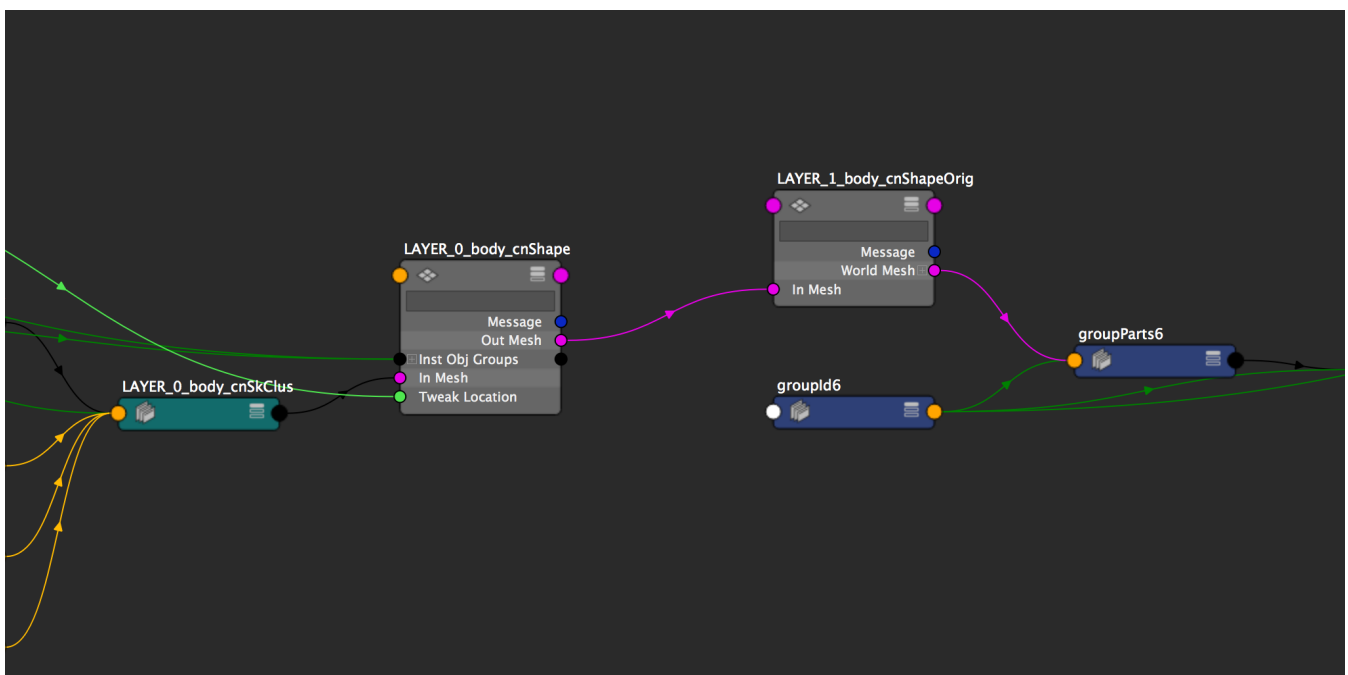


Figure 3b: outMesh -> inMesh connection



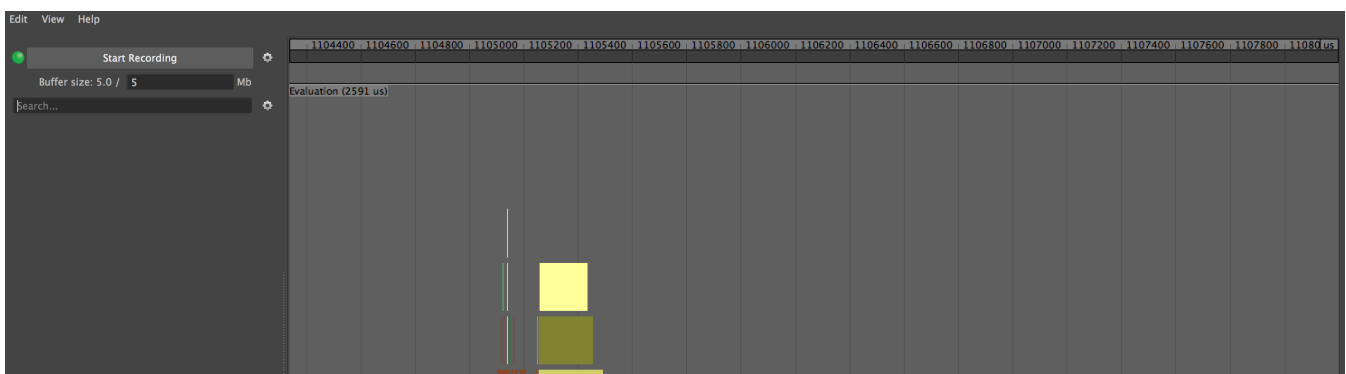
Figure 3c: GPU point count after connection

What of that second mesh? Maya may determine that the second part of the chain is GPU capable, which creates another issue: the CPU-only source mesh now has to be uploaded to the GPU in its deformed state. On my laptop this seems to cause 2ms of latency. Don't trust me on this — let's go to the Profiler.

Your Best Friend, the Profiler

Sorry, what's that? You haven't used the Profiler? Why, then you haven't lived, my friend! In the Parallel Evaluation / GPU Override utopia of the future, the Profiler is your spirit guide. There are a many changes you can make to a rig today that, while speeding up things in the old DG world, can slow you down now. ***If you change your rig, profile.*** Profile the same animation, and let it run a long time if you can (set the buffer to higher than 20mb). Check your average timings. Look at how your setup stacks together with a single animated character, then with two. You'll be surprised, I bet, that not everything is working as you believe it should.

In the profiler it's easy to see the difference between the separate meshes versus the outMesh-connected ones:



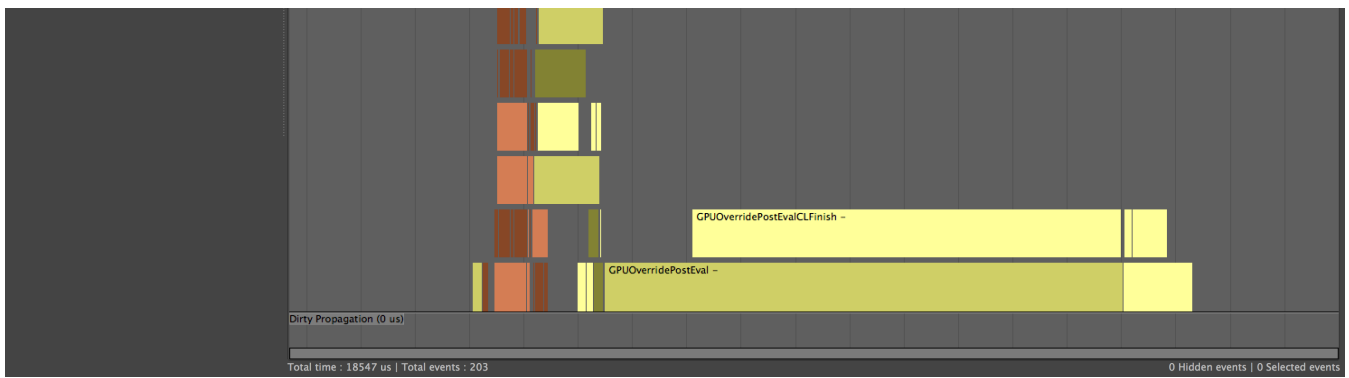


Figure 4: profile of disconnected, skinned meshes

As you can see, the disconnected meshes both have GPU-override skinClusters (those are the stacks of nodes in yellow before the long yellow bars). Because they're not connected, they stack together well and the whole scene evaluates quickly.

However, the outMesh-connected profile looks completely different:

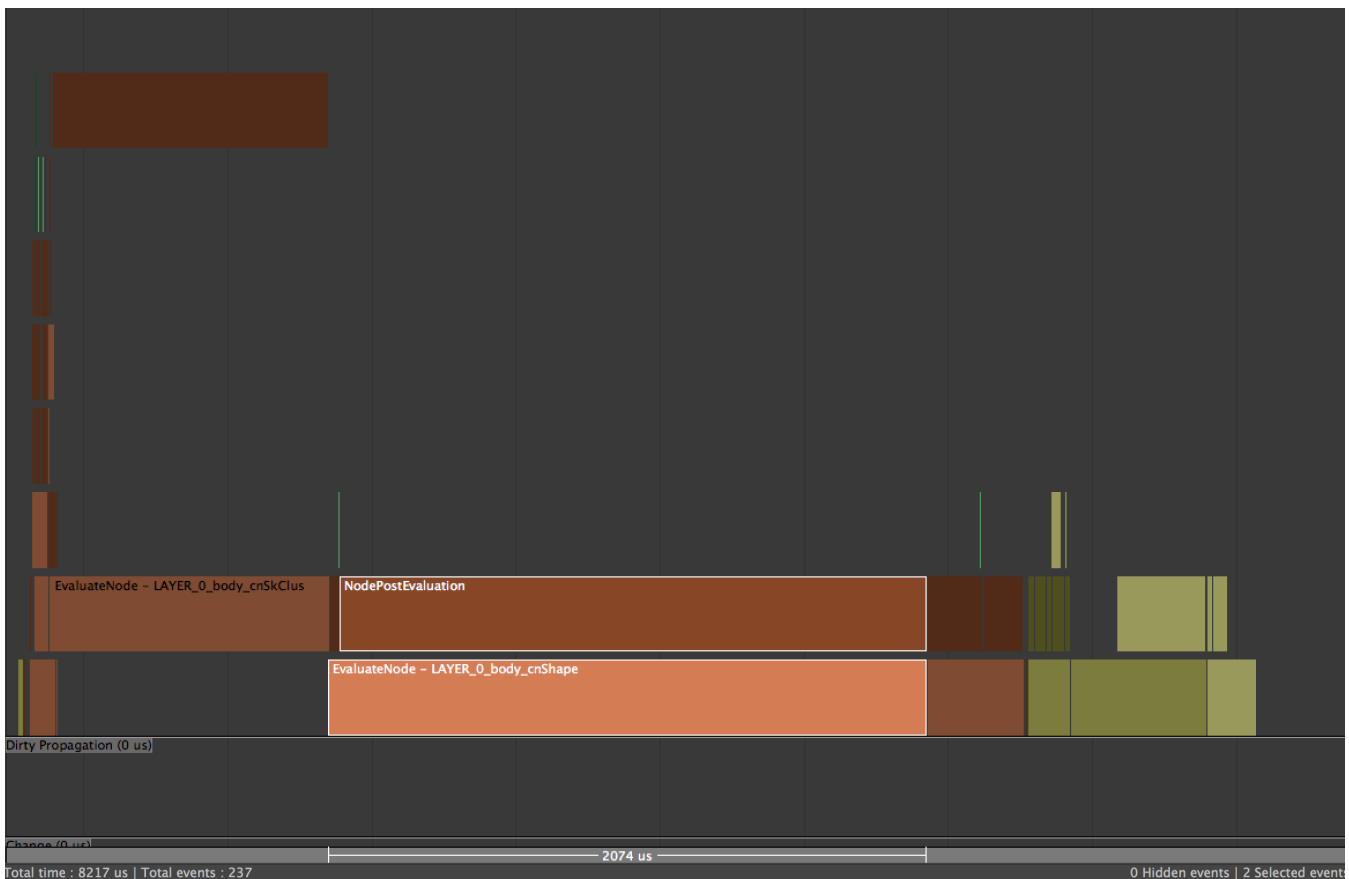


Figure 5: profile of outMesh-connected graph

In this graph, the evaluation is more stretched out. This is expected; the second skin can't fire before receiving the results of the first. However, notice that there is one less GPU override in the stack (that EvaluateNode on LAYER_0_body_cn's skin is running on CPU and taking around 900us on my machine), and also, there's a nice fat *NodePostEvaluation* sucking up a solid 2ms. If your frame budget is aiming at 24fps,

you're now down from 42 ms to 40 ms for the entire rig (less, practically, as VP2 and general Maya tax eats a lot, especially on Mac). If you're aiming at a higher interactive frame rate, say 60fps, *you've just lost an eighth of your entire frame budget.*

Obviously, we want to avoid this. Let's talk solutions.

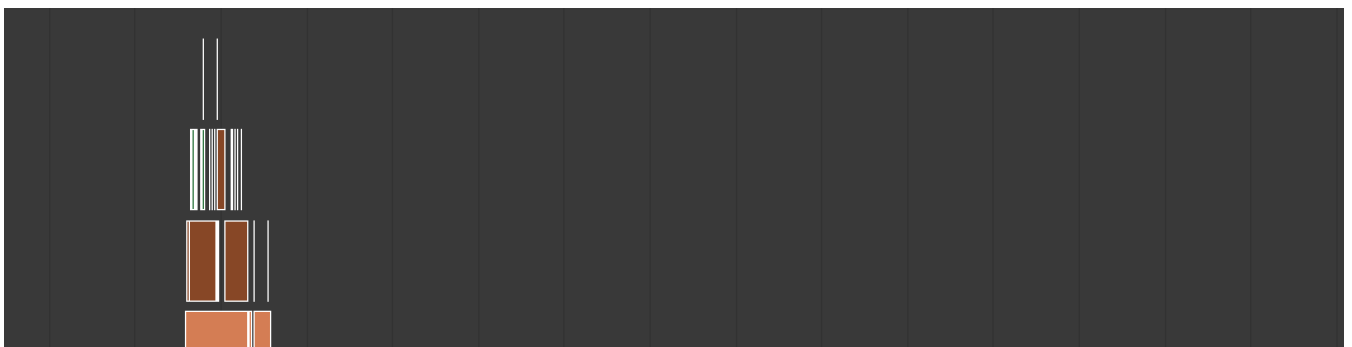
How Do We Fix This?

The best solution I've found so far is to chain skins. It's conceptually simple: instead of having skins that feed one another using blendShapes or nodal connections, you use the deformer command to add new skins to the current mesh, one literally after the other, then copy weights and influences over.

The benefits? Your mesh stays on the GPU the whole time. While you have multiple skins that run in serial on the GPU, they each run much more quickly than their CPU counterparts and you don't suffer a GPU buffer upload mid-frame. The down sides? You can get to a point where Maya tools get confused. Painting skin weights on a mesh with three skins isn't something you want to do.

Thus, this solution has two stages. In the first stage, you use the blendShape method to get your skins weighted and to test your deformation design. Once it's all working, you use a script to move the skinClusters of all the layers onto a single, final mesh that is suitable for animation.

"A script, you say?" Yes, that's the other down side. I haven't figured out how to make this trick work cleanly every time with stock Maya tools. Perhaps on reading this article someone will show me something I didn't consider, but for now this is a scripted solution. [I'm providing a working Python example to make things simpler.](#) Copy it into your script editor, select source and target meshes, and run. It will copy the skin from the source mesh onto the target as a new skinCluster. Once complete, you can delete the source blendShape chain. This results in a cleaner, faster scene with a blazing deform chain.



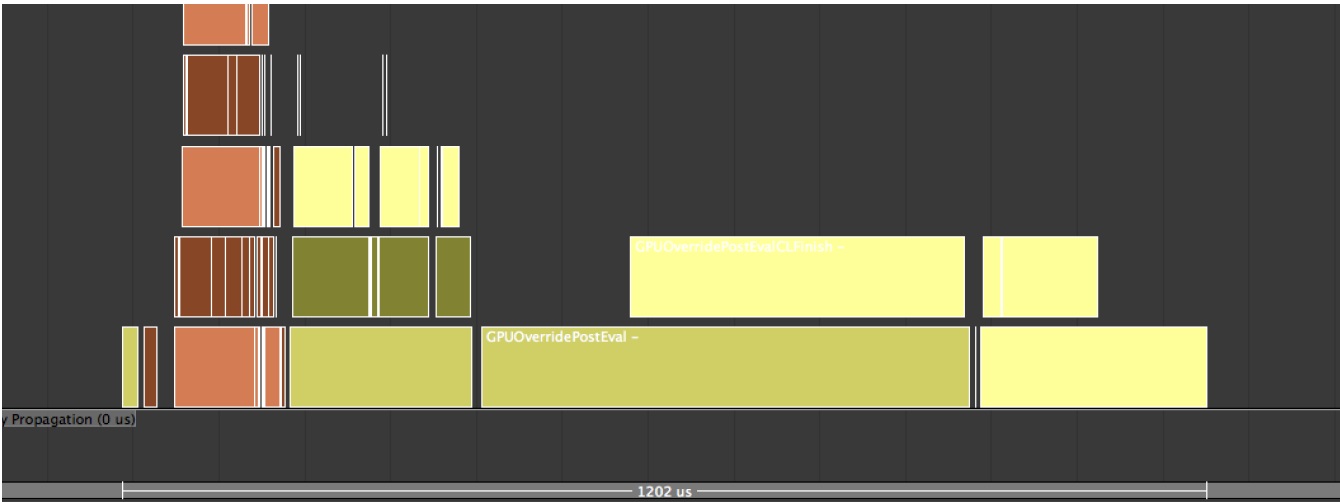
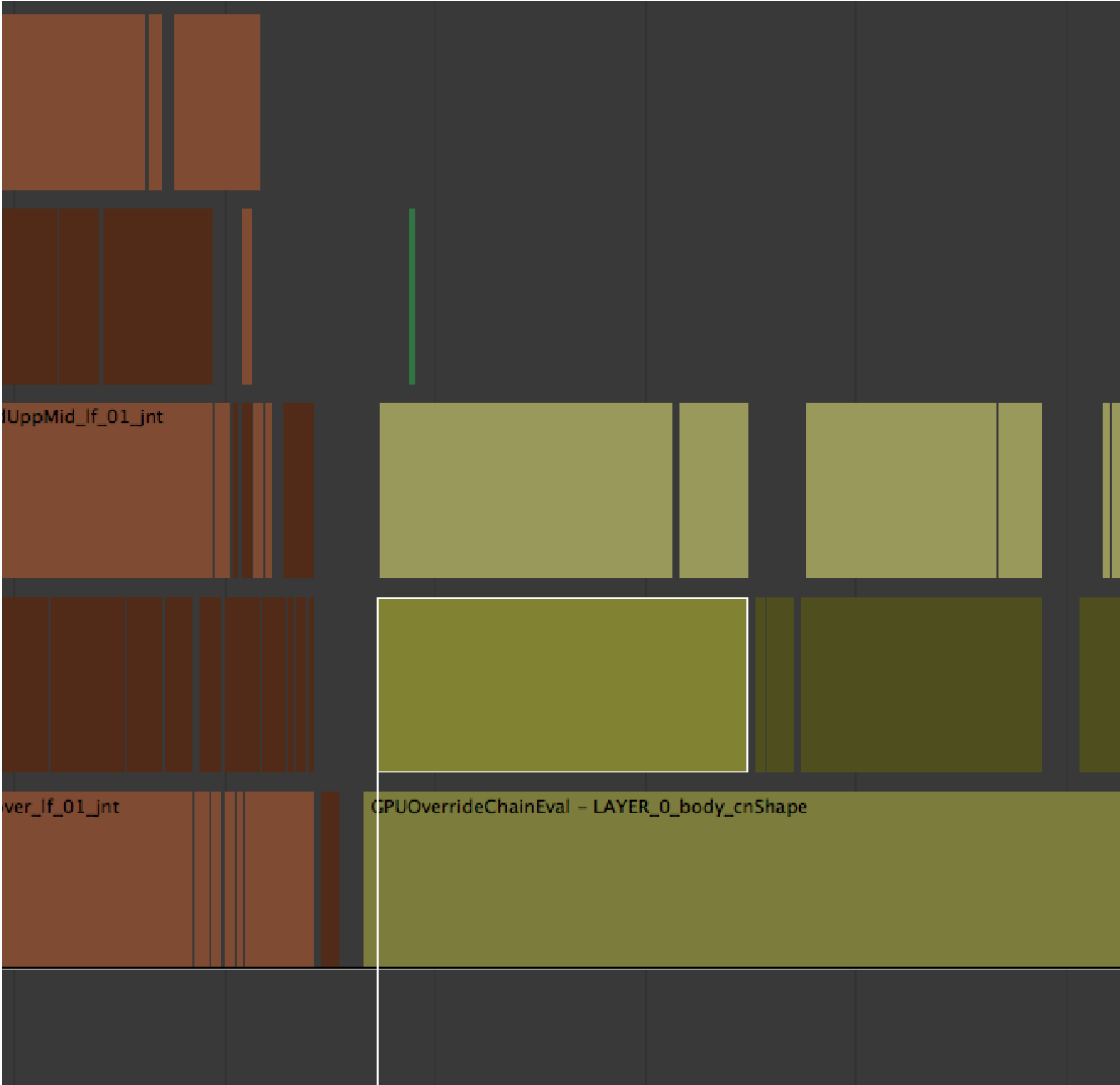


Figure 6: profile of stacked skins

NodePostEvaluation: gone. And that 900us that was being spent on LAYER_0's skin? I think we've done better there too:



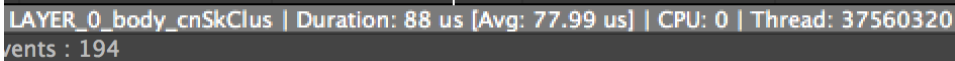


Figure 7: profile of LAYER_0 skin node

To put this into perspective, on my 2015 MacBook Pro (AMD Radeon R9 M370X), skin stacking has taken a roughly 4ms evaluation down to 1.2ms, but the deformation result is the same.

Not bad, right? Here is the complete script:

```

1
2  from maya.api import OpenMaya as om2
3  from maya.api import OpenMayaAnim as oma2
4  import pymel.core as pm
5
6
7  ## =====
8  def get_deform_shape( ob ):
9      """
10         Gets the visible geometry shape regardless of whether or not
11         the object is deformed or not.
12
13         :param ob: The object to check.
14         :returns: The object's deform shape.
15         """
16
17         ob = pm.PyNode(ob)
18         if ob.type() in ['nurbsSurface', 'mesh', 'nurbsCurve']:
19             ob = ob.getParent()
20             shapes = pm.PyNode(ob).getShapes()
21             if len(shapes) == 1:
22                 return( shapes[0] )
23             else:
24                 real_shapes = [ x for x in shapes if not x.intermediateObject.get() ]
25                 return( real_shapes[0] if len(real_shapes) else None )
26
27
28  ## -----
29  def get_skin_cluster(ob):
30      """
31         Find the first skinCluster on the object that actually effects it.
32         When multiple deformers are layered, this means that the latest skin
33         in the deform chain will be returned.
34
35         In situations where multiple skins are layered through blendshapes,
36         this function will only return the skin on the current mesh and not

```

```

36     this function will only return the skin on the current mesh and not
37     any of the blendshape sources.
38
39     :param ob: The object whose skin you wish to return.
40     :returns: A skinCluster object, or None.
41     """
42
43     shape = get_deform_shape( ob )
44     if shape is None:
45         return(None)
46     skins = pm.ls( pm.listHistory( shape ), type='skinCluster' )
47     if len( skins ):
48         for skin in skins:
49             # log( '\t+ Processing %s...' % skin )
50             groupId = skin.input[0].groupId.inputs( plugs=True )[0]
51             outputs = groupId.outputs(plugs=True)
52             for outp in outputs:
53                 node = outp.node()
54                 if node == shape or node == ob:
55                     ## force neighbors!
56                     skin.weightDistribution.set(1) ## neighbors
57                     return( skin )
58     return( None )
59
60
61     ## -----
62     def log( msg, warn=False, error=False ):
63         if warn and error:
64             raise ValueError('Please specify only one of warn / error.')
65
66         log_func = om.MGlobal.displayInfo
67         if warn:
68             log_func = om.MGlobal.displayWarning
69         elif error:
70             log_func = om.MGlobal.displayError
71
72         log_func( msg )
73
74
75     ## =====
76     ## om2 utilities
77     def get_mobject( name ):
78         sel = om2.MGlobal.getSelectionListByName( name )
79         return sel.getDependNode(0)
80
81     ## -----
82     def get_dag_path( name ):
83         sel = om2.MGlobal.getSelectionListByName( name )

```

```

84         return sel.getDagPath(0)
85
86     ## -----
87     def get_mfn_skin( skin_ob ):
88         if isinstance( skin_ob, pm.PyNode ):
89             skin_ob = get_mobject( skin_ob.longName() )
90         return oma2.MFnSkinCluster( skin_ob )
91
92     ## -----
93     def get_mfn_mesh( mesh_ob ):
94         if isinstance( mesh_ob, pm.PyNode ):
95             mesh_ob = get_mobject( mesh_ob.longName() )
96         return om2.MFnMesh( mesh_ob )
97
98     ## -----
99     def get_complete_components( mesh_ob ):
100         assert( isinstance(mesh_ob, om2.MFnMesh) )
101         comp = om2.MFnSingleIndexedComponent()
102         ob = comp.create( om2.MFn.kMeshVertComponent )
103         comp.setCompleteData( mesh_ob.numVertices )
104         return( ob )
105
106     ## =====
107     def move_skin( source, target ):
108         source_shape = get_deform_shape( source )
109         source_dp = get_dag_path( source_shape.longName() )
110         source_skin = get_skin_cluster( source )
111         source_mfn = get_mfn_skin( source_skin )
112         source_mesh = get_mfn_mesh( get_deform_shape(source) )
113         components = get_complete_components( source_mesh )
114
115         weights, influence_count = source_mfn.getWeights( source_dp, components )
116
117         pm.select( cl=True )
118         target_skin = pm.deformer( target, type='skinCluster', n='MERGED__' + source_skin.name
119
120         ## copy over input values / connections
121         bind_inputs = [ (x.inputs(plugs=True)[0] if x.isConnected() else None) for x in source
122         bind_values = [ x.get() for x in source_skin.bindPreMatrix ]
123         mat_inputs = [ (x.inputs(plugs=True)[0] if x.isConnected() else None) for x in source
124         mat_values = [ x.get() for x in source_skin.matrix ]
125
126         for index, bind_value, mat_value in zip( xrange(influence_count), bind_values, mat_val
127             target_skin.bindPreMatrix[index].set( bind_value )
128             target_skin.matrix[index].set( mat_value )
129
130         for index, bind_input, mat_input in zip( xrange(influence_count), bind_inputs, mat_inp
131             if bind input:

```

```

132         bind_input >> target_skin.bindPreMatrix[index]
133     if mat_input:
134         mat_input >> target_skin.matrix[index]
135
136     ## copy over weights
137     target_mfn = get_mfn_skin( target_skin )
138     target_mesh = get_mfn_mesh( get_deform_shape(target) )
139     target_dp = get_dag_path( get_deform_shape(target).longName() )
140     components = get_complete_components( target_mesh )
141     all_indices = om2.MIntArray( range(influence_count) )
142
143     target_mfn.setWeights( target_dp, components, all_indices, weights )
144
145
146     ## =====
147     ## main
148
149     items = pm.selected()
150
151     if len(items) == 2:
152         move_skin( items[0], items[1] )
153
154         log(
155             '+ SkinMerge: Complete. Merged skin from {} onto {}.'
156             .format( *items )
157         )
158
159     else:
160         log( "-- Please select a skinned mesh and a target mesh.", error=True )
161

```

Yes, I use tabs. Yes, I mix PyMel and OM2. I've made peace with my own weirdness.

Extensions to the Technique

I haven't provided an example, but layers past LAYER_0 may need to have bindPreMatrix connections into the skinClusters if you want to have the driving joints move with the rig. This is a simple addition to the above script. In my current facial setup I have three layers (0, 1, and 2), and all joints on the last two layers have bindPreMatrix connections.

Why am I moving joints with the rig, you may ask? Even with all my experimentation I haven't yet hit a combination where running the face in before the body (as was the best practice in years past) stacks / evaluates as well in Parallel Evaluation. I've done experiments and profiled, and for all the work involved in shuffling around transform spaces the benefits were either minimal or negative. Again, profile everything!

Now Go Rig Something

I hope this helps you move your rigs into Maya 2017 and beyond, and to really squeeze the most performance out of the tools at hand. Remember to check the script in the [Github repo](#), and send me a pull request if you come up with something better.

kattkieru/medium_def_layers

medium_def_layers - Supporting files and figures for my Medium article, Deformation Layering in...

[github.com](#)



Maya

3d

Rigging

Python

Skinning

Medium

[About](#) [Help](#) [Legal](#)