SIGGRAPH2007

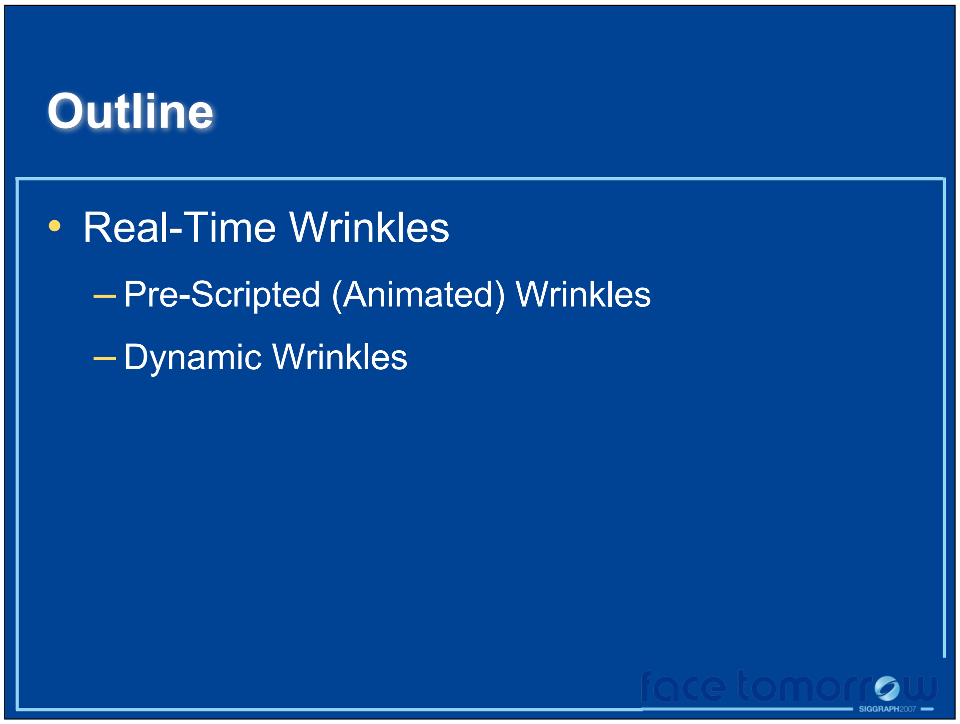
Real-Time Wrinkles

Christopher Oat

AMD, Inc.

Ruby: Whiteout Demo

Ruby: Whiteout real-time demo.



Outline

- Real-Time Wrinkles
  - Pre-Scripted (Animated) Wrinkles
  - Dynamic Wrinkles

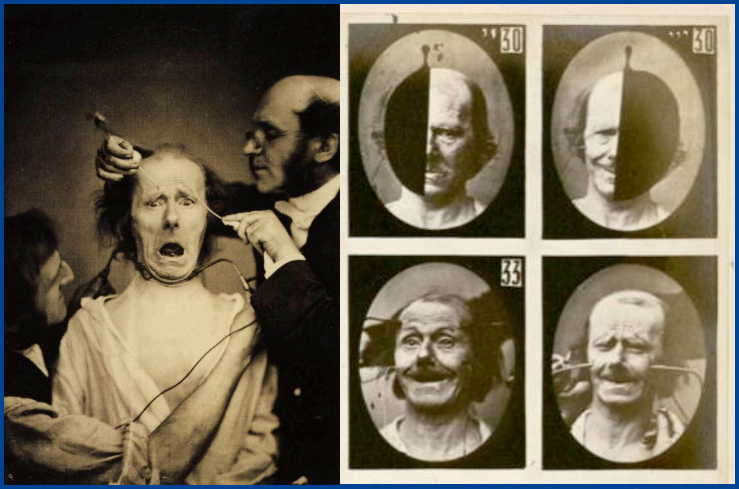
face tomorrow SIGGRAPH2007

Facial wrinkles

Ruby in 2067?

The kinds of wrinkles we're talking about today are the kind of wrinkles you get from moving your facial muscles. These are the kind of wrinkles that appear and disappear as you change facial expressions. Example: make a surprised face and your eyebrows go up and your forehead wrinkles or when you eat something very sour and you face puckers. The kinds of wrinkles I'm talking about are the kind that we associate with strong facial expression. These kinds of wrinkles, it turns out, are very important when you want to create compelling facial animation.

Facial wrinkles are important

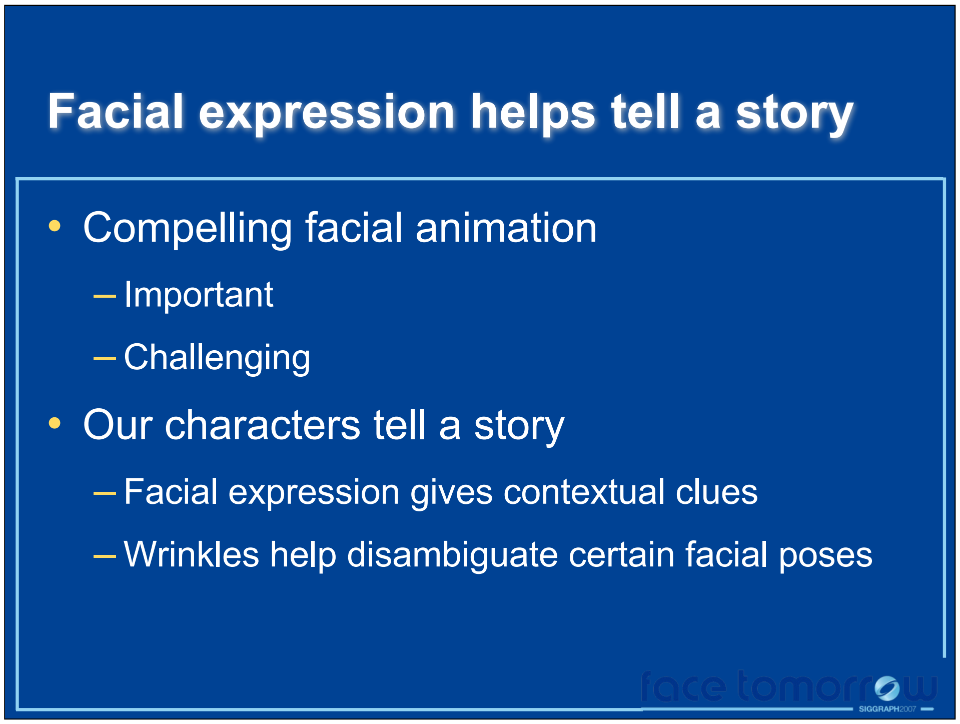


www.wikipedia.org/wiki/Guillaume_Duchenne www.wikipedia.org/wiki/Facial_expression

Facial wrinkles are important for interpreting facial expression. In the early to mid-1800s a French neurologist named Guillaume Duchenne performed experiments that involved applying electric simulation to the muscled in people's face. He was able to determine which facial muscles were used for different facial expressions. One thing he discovered is that smiles resulting from true happiness not only utilize the muscles of the mouth but also those of the eyes. These kind of "genuine" smiles are called Duchenne smiles (according to Wikipedia). What we learn from this is that facial expressions are complex and sometimes subtle but they are important. It would be difficult to come up with a automatic/dynamic method for facial wrinkles that didn't land you deep in the uncanny valley… your system would have to distinguish between such things as fake smiles and Duchenne smiles. So you really want a human sitting at the steering wheel when it comes to driving facial wrinkles as they relate to facial expressions. You really want your artists to have control over this aspect of facial animation.

Image and text taken from Wikipedia (as it appeared on July 17, 2007): www.wikipedia.org/wiki/Facial_expression

Image and text taken from Wikipedia (as it appeared on July 17, 2007): www.wikipedia.org/wiki/Guillaume_Duchenne

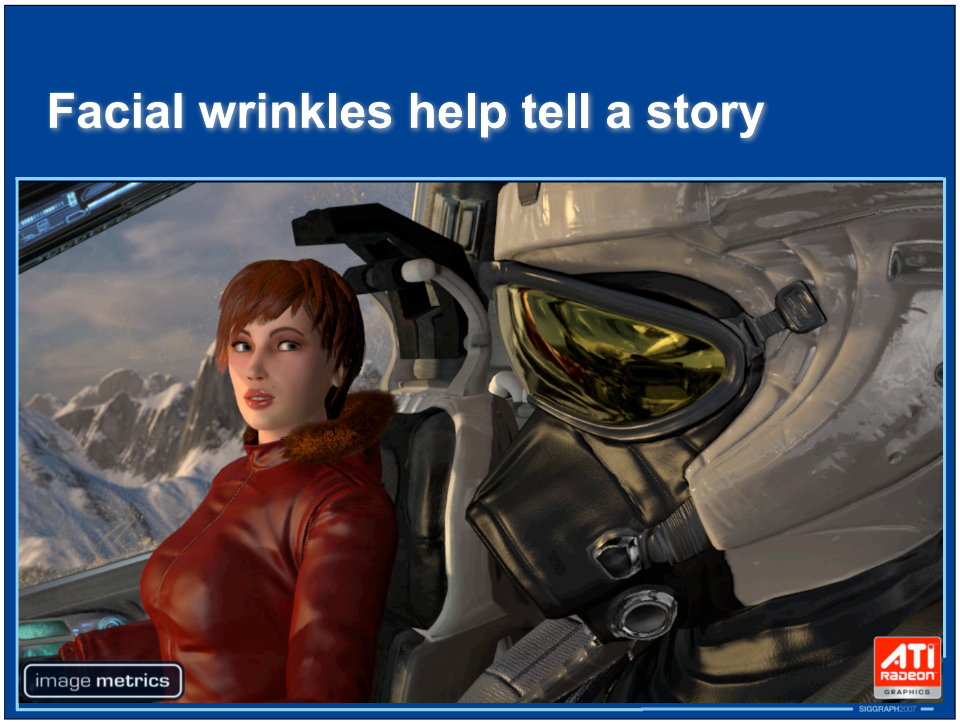


Facial expression helps tell a story

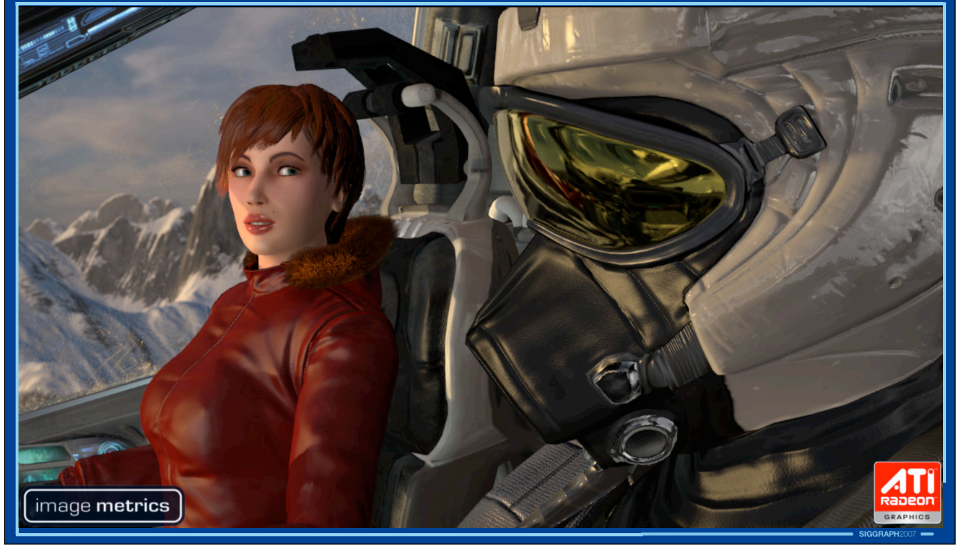
- Compelling facial animation
  - Important
  - Challenging
- Our characters tell a story
  - Facial expression gives contextual clues
  - Wrinkles help disambiguate certain facial poses

face tomorrow SIGGRAPH2007

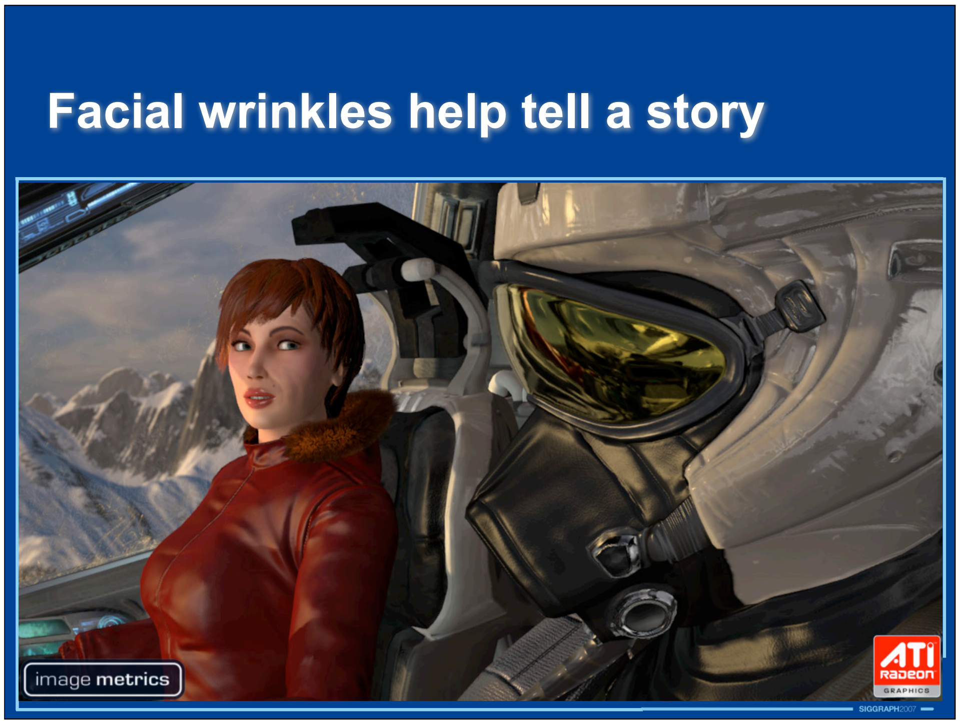
Compelling facial animation is an extremely important and challenging aspect of computer graphics. Both games and animated feature films rely on convincing characters to help tell a story and an important part of character animation is the character's ability to use facial expression. Without even realizing it, we often depend on the subtleties of facial expression to give us important contextual cues about what someone is saying, thinking, or feeling. For example a wrinkled brow can indicate surprise while a furrowed brow may indicate confusion or inquisitiveness.



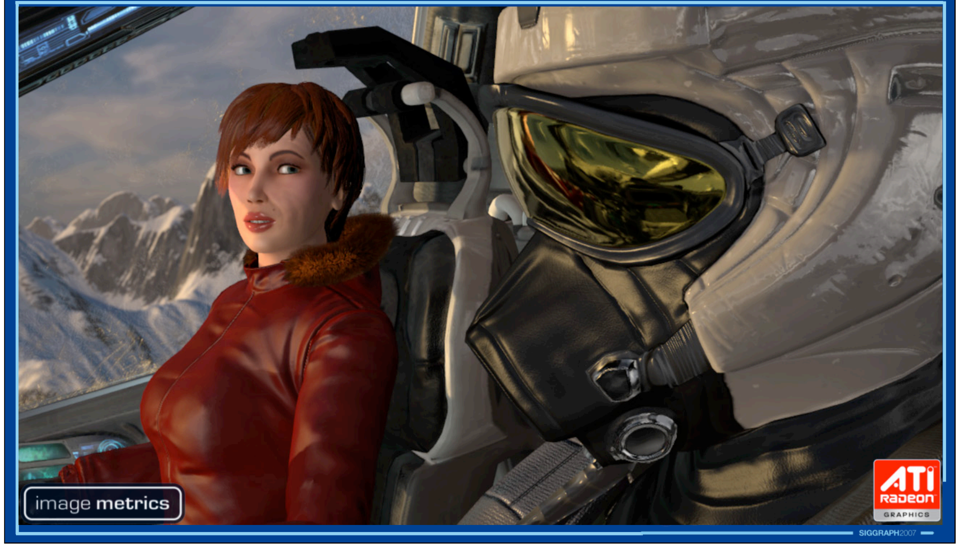
Facial wrinkles help tell a story



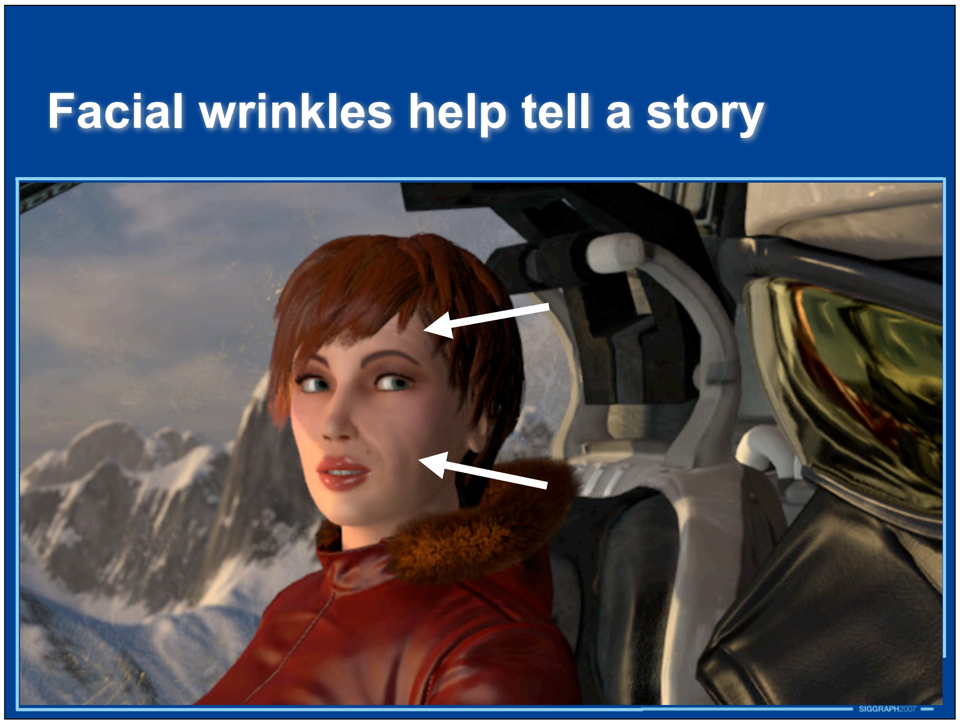
Here we see Ruby with her facial wrinkles removed. This gives her a rather expressionless look on her face and it makes it very difficult to guess what she's thinking, what kind of mood she's in. Without some facial cues we're left guessing what Ruby is thinking.



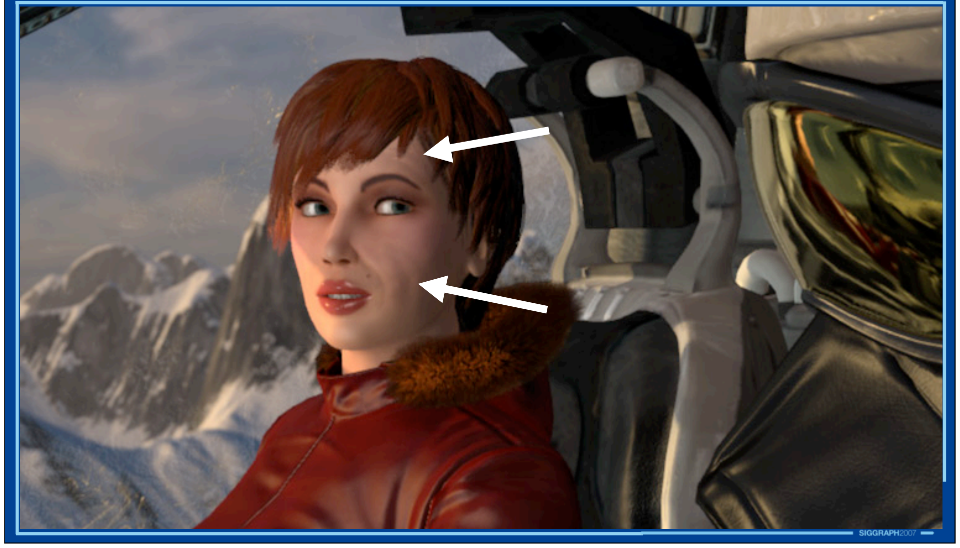
Facial wrinkles help tell a story



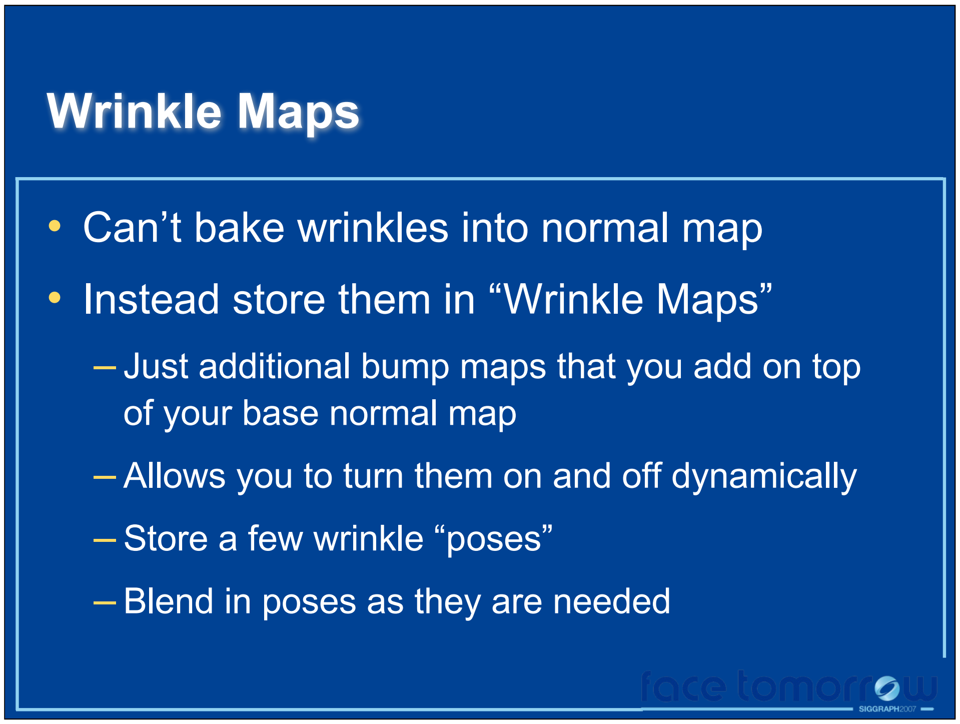
Here's the same image with the wrinkles added in. We see that Ruby's brow is wrinkled and that little spot between her eye brows is dimpled (sort of puckered?) and we know that she's thinking something; she's scheming. And she's about to say something cunning. By adding a wrinkle map we've completed the picture. Ruby's expression tells us her mood, helps us understand the kind of person she is and explains what she's doing here in this image.



Facial wrinkles help tell a story



Here's the same image with the wrinkles added in. We see that Ruby's brow is wrinkled and that little spot between her eye brows is dimpled (sort of puckered?) and we know that she's thinking something; she's scheming. And she's about to say something cunning. By adding a wrinkle map we've completed the picture. Ruby's expression tells us her mood, helps us understand the kind of person she is and explains what she's doing here in this image.

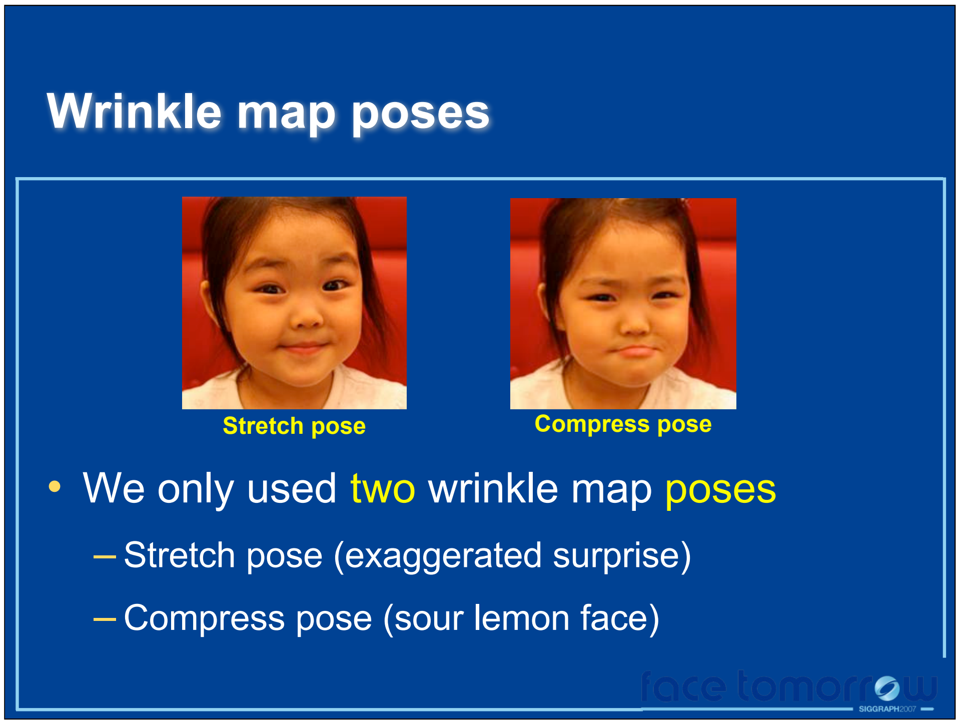


Wrinkle Maps

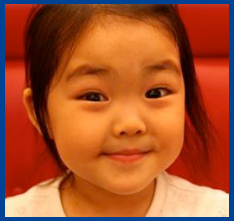
- Can’t bake wrinkles into normal map
- Instead store them in “Wrinkle Maps”
  - Just additional bump maps that you add on top of your base normal map
  - Allows you to turn them on and off dynamically
  - Store a few wrinkle “poses”
  - Blend in poses as they are needed

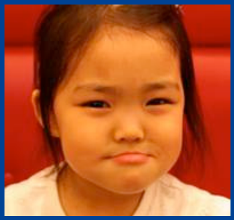
face tomorrow SIGGRAPH2007

Can’t simply paint wrinkles into our character’s normal map or we will curse them to a single facial expression forever. Instead we store our wrinkles in a separate wrinkle map. Wrinkle maps are just additional bump maps that get added on top of our base normal maps. But because they are autonomous from the normal map they can be independently turned on and off.



Wrinkle map poses

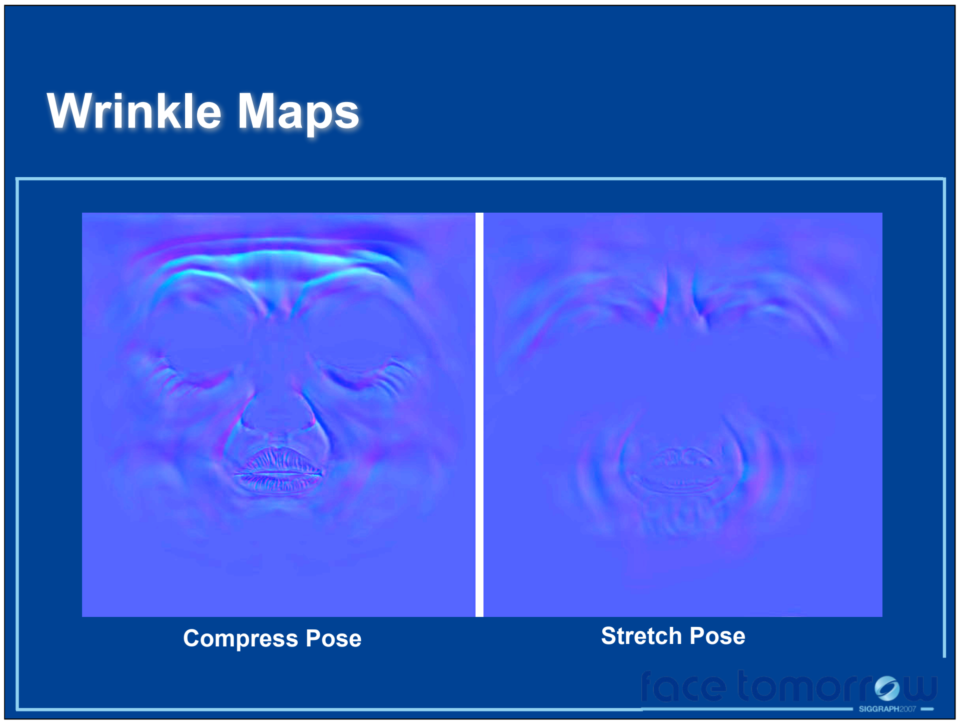

Stretch pose


Compress pose

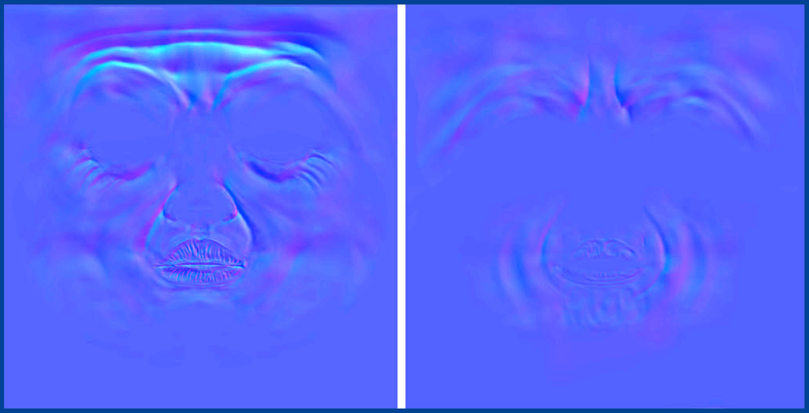
- We only used two wrinkle map poses
  - Stretch pose (exaggerated surprise)
  - Compress pose (sour lemon face)

The technique we came up with utilizes just two wrinkle map poses. We then use a set of masks and artist animated weights to allow wrinkles to be animated independently in different regions of our character's face. The end result is a wrinkled normal map that is used to while rendering our character's face.

The first wrinkle map encodes wrinkles for a stretched expression (exaggerated surprise expression: eyes wide open, eyebrows up, forehead wrinkles, mouth open) and the second wrinkle map encodes wrinkles for a compressed expression (think of sucking on a sour lemon: eyes squinting, forehead compressed down towards eyebrows, lips puckered, chin compressed and dimpled).

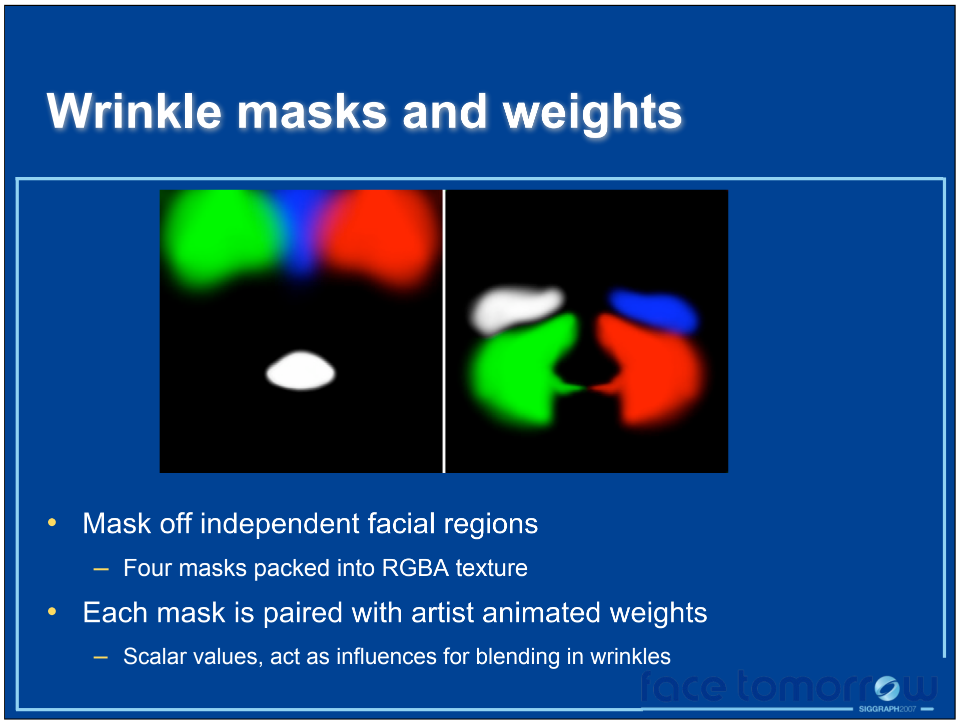


Wrinkle Maps

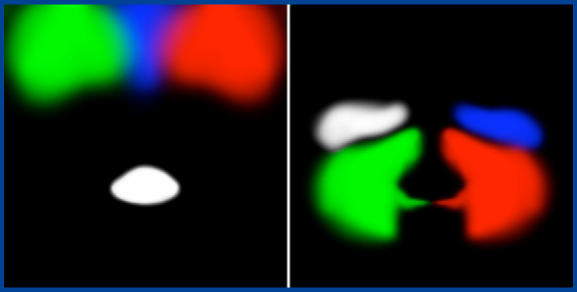


face tomorrow SIGGRAPH2007

Here are what the actual wrinkle maps look like. They’re simply tangent space normal maps.



Wrinkle masks and weights

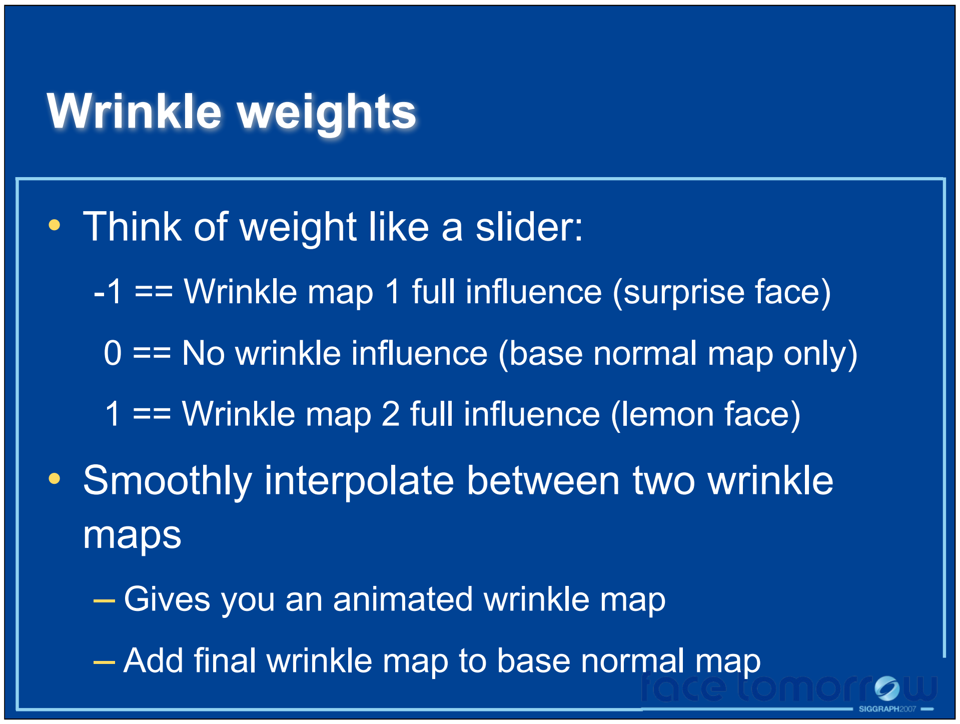


- Mask off independent facial regions
  - Four masks packed into RGBA texture
- Each mask is paired with artist animated weights
  - Scalar values, act as influences for blending in wrinkles

In order to have independently controlled wrinkles on our character's face, we must divide her face into multiple regions. Each region is specified by a mask that is stored in a texture map. Because a mask can be stored in a single color channel (it's just a scalar) of a texture, we are able to store up to four masks in a single four channel texture as shown above. Using masks allows us to store wrinkles for different parts of the face, such as the chin and forehead, in the same wrinkle map and still maintain independent control over wrinkles on different regions of our character's face.

Each mask is paired with an animated wrinkle weight. Wrinkle weights are scalar values and act as influences for blending in wrinkles from the two wrinkle maps. For example, the upper left brow (red channel of left most image) will have its own "Upper Left Brow" weight. Each weight is in the range [1,-1] and corresponds to the following wrinkle map influences:

Sample mask textures to build a vector of wrinkle masks and get the weights from constant store to build a vector of weights. Then just dot the mask vector with the weight vector to compute a wrinkle map influence.



Wrinkle weights

- Think of weight like a slider:

  -1 == Wrinkle map 1 full influence (surprise face)

  0 == No wrinkle influence (base normal map only)

  1 == Wrinkle map 2 full influence (lemon face)

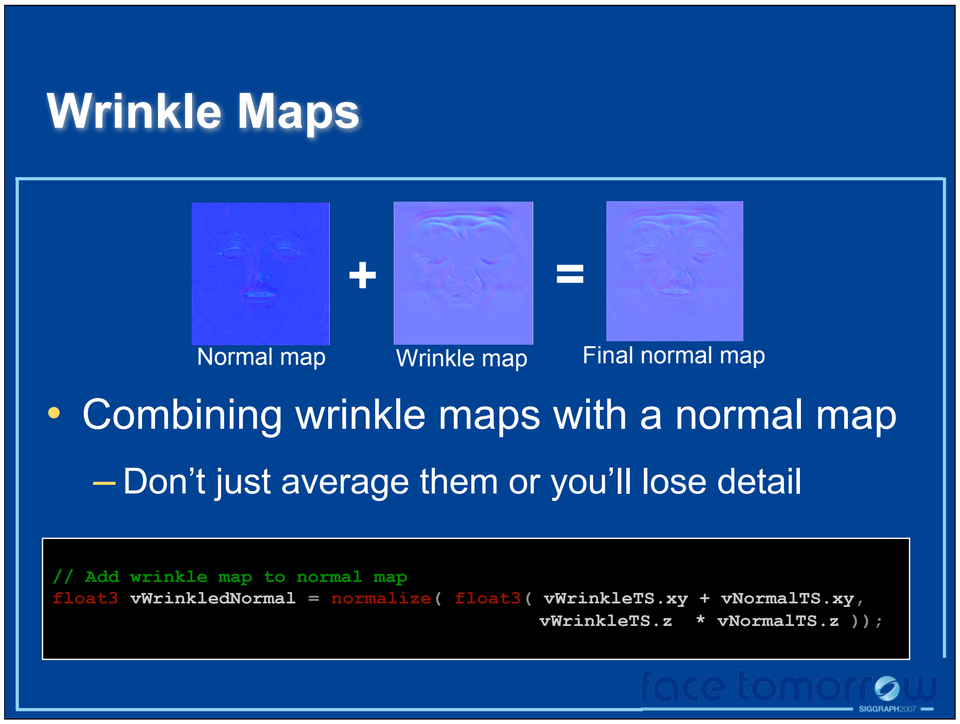
- Smoothly interpolate between two wrinkle maps
  - Gives you an animated wrinkle map
  - Add final wrinkle map to base normal map

face tomorrow
SIGGRAPH2007

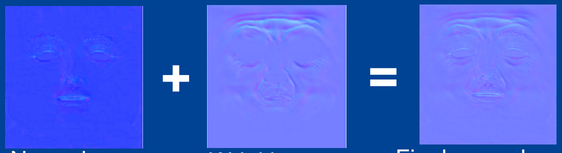
-1 == full influence from wrinkle map 1 (surprise face)

0 == no influence from either wrinkle map (base normal map only)

1 == full influence from wrinkle map 2 (“what’s that smell?” face)



Wrinkle Maps



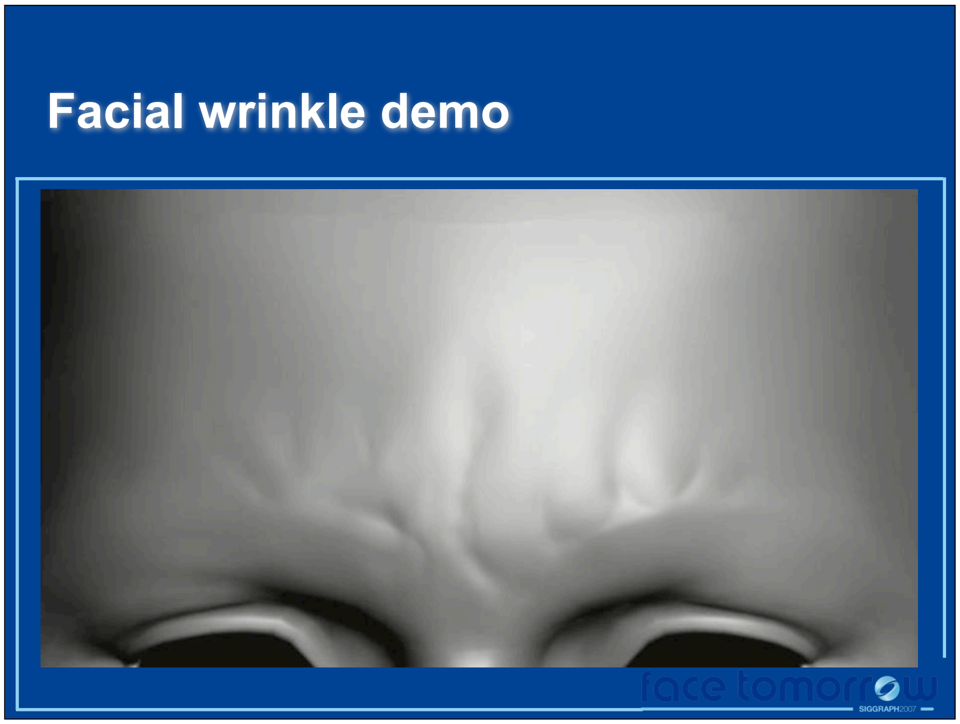
- Combining wrinkle maps with a normal map
  - Don’t just average them or you’ll lose detail

```
// Add wrinkle map to normal map
float3 vWrinkledNormal = normalize( float3( vWrinkleTS.xy + vNormalTS.xy,
                                            vWrinkleTS.z  * vNormalTS.z ));
```

As usual, the normal map encodes fine surface detail such as pores, scars, or other facial details. Thus the normal map acts as a base layer on top of which wrinkle maps are added. Because the normal map includes important detail, we do not want to simply average the normal map with the wrinkle maps or some of the normal map’s details may be lost.

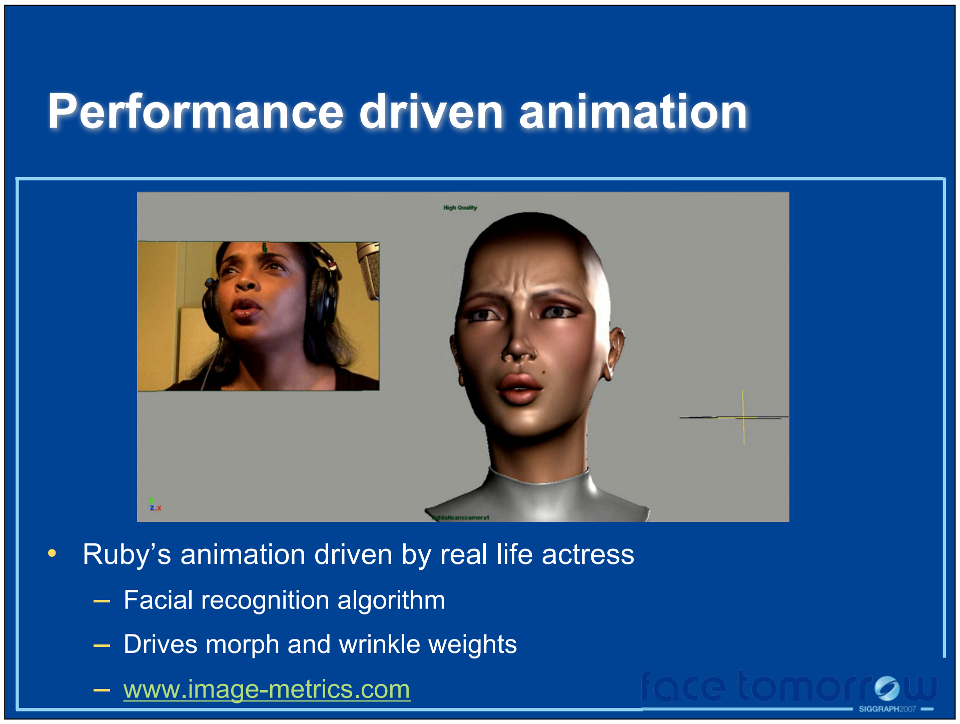
The idea is that when we composite two normal maps, any bumps that exist should end up in the final composition. So if there are pores in the normal map then we want those pores to be just as strong when we blend in a wrinkle map. If you just average the two normals (add them then renormalize) the details can start to fade away. For example, if you have one normal map with lots of high frequency detail and another normal map that is flat (all normals are <0,0,1>) after you blend them by averaging, the resultant normals are a dumbed down version of your high frequency normal map (the normals all shift towards pointing straight up).

One thing to note is that a normal is “bumpy” when the magnitude of its z component is small. The less that a normal points in the z direction, the more “bumpy” it is. The direction of the bump is all in the x and y components. So we blend the x and y components which gives a blended bump direction. For example, if you have a normal pointing “north” and a normal pointing “west” you end up with a normal pointing north-west. Now, you can’t simply blend the z components in the same way or your bumps will start to flatten out. Using multiplication gives us what we want here… for example if you are on the edge (wall?) of a pore in the normal map then your z value might be 0.75 and if you are on the edge/curve of a wrinkle in the wrinkle map then the z value might be 0.5. In order to get that pore onto the edge of the wrinkle you need a very small z value (since the wrinkle is kind of a bump itself, on that part of the wrinkle 0.5 is perpendicular to the surface… that’s why the normal points that way in the first place after all). Multiplying gives us something less than perpendicular (i.e. a bump on a curving surface) and it’s “less than perpendicular” in the right direction.



Facial wrinkle demo

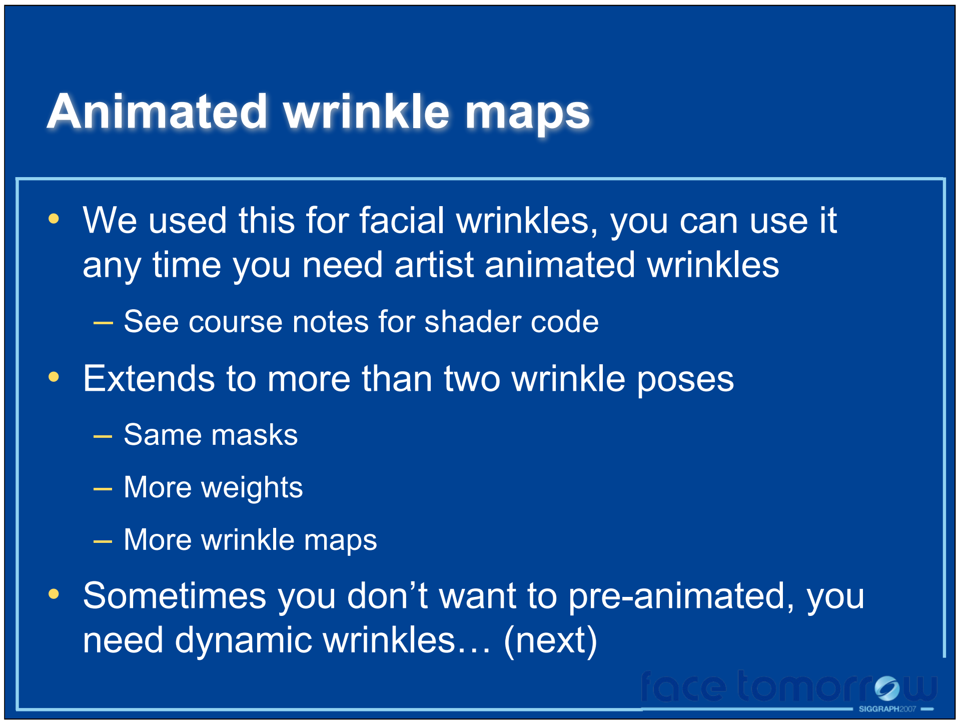
Short clip that demonstrates this technique.



Performance driven animation

- Ruby’s animation driven by real life actress
  - Facial recognition algorithm
  - Drives morph and wrinkle weights
  - www.image-metrics.com

One interesting note is that much of the facial animation in the Ruby demo, including the animate wrinkle weights, was generated using a performance driven animation technique. We worked with a company named ImageMetrics who filmed our voice actress, the woman that does Ruby’s voice, while she was performing and ran the captured frames through a facial recognition algorithm that would automatically set our facial morph blend weights (for the facial mesh) as well as our animated facial wrinkle weights.

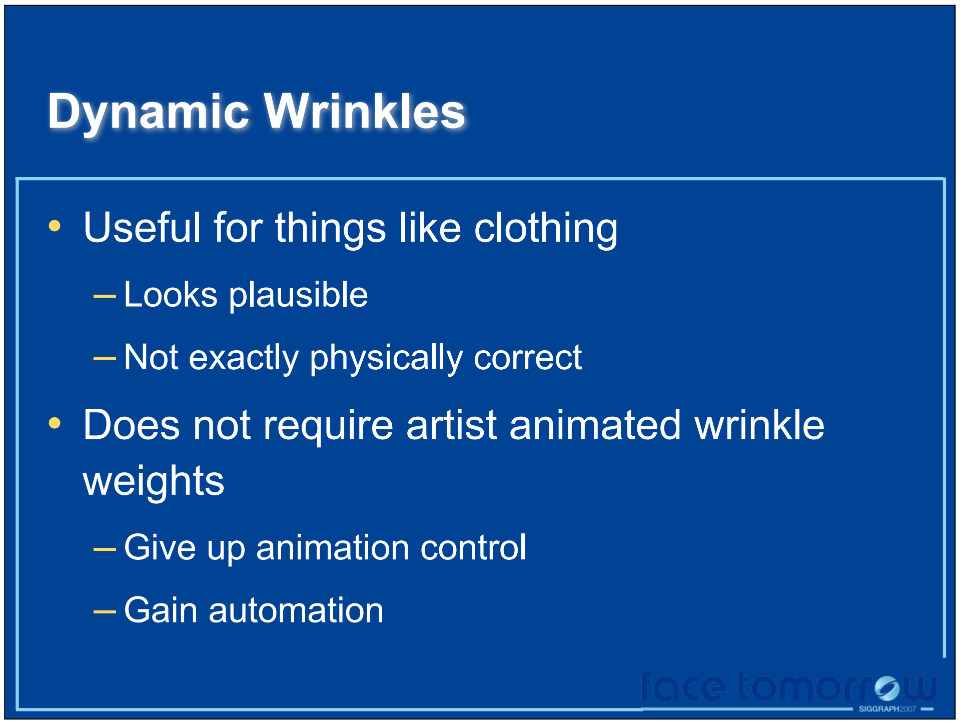


Animated wrinkle maps

- We used this for facial wrinkles, you can use it any time you need artist animated wrinkles
  – See course notes for shader code
- Extends to more than two wrinkle poses
  – Same masks
  – More weights
  – More wrinkle maps
- Sometimes you don't want to pre-animated, you need dynamic wrinkles… (next)

face tomorrow
SIGGRAPH2007

This technique is simple and efficient and allows artists to animate wrinkles on a surface. I've demonstrated the technique with two wrinkle poses but there's no reason you couldn't extend the idea to allow for many different kinds of wrinkle poses to be blended for many different regions on a character's face. The strength of this technique lies in its ability to be art directed, you get explicit control over how and when your surfaces wrinkle. This allows artists to give their character's detailed facial expressions but of course there is a time cost associated with this. Someone has to manually go through and set all the wrinkle weight key frames and there are definitely situations where you want wrinkles but you don't want to have to spend time going through this manual animation process.

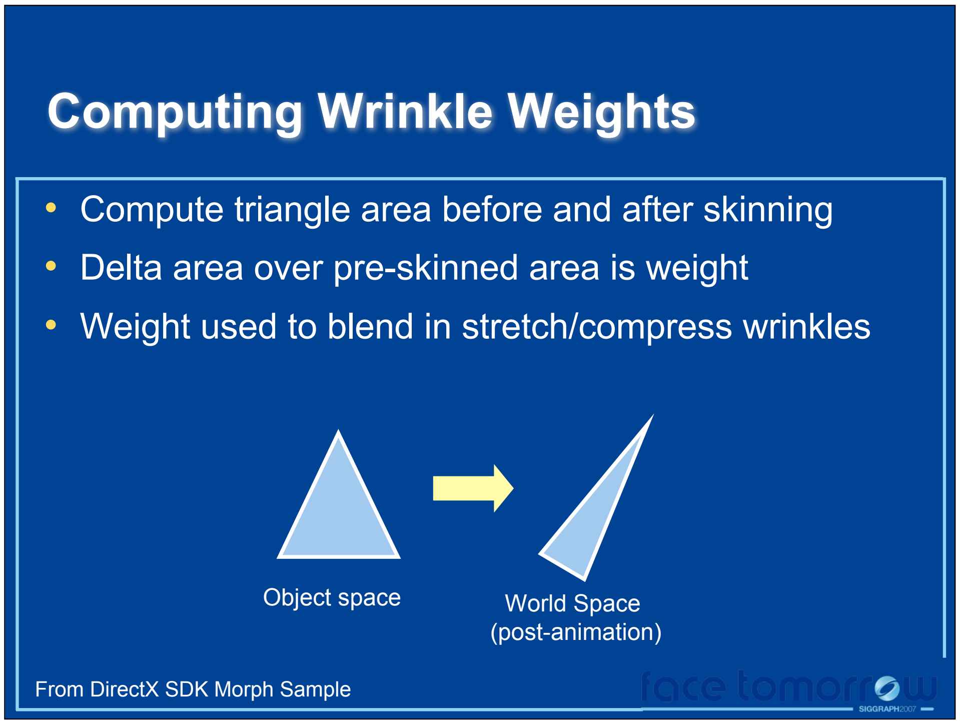


Dynamic Wrinkles

- Useful for things like clothing
  - Looks plausible
  - Not exactly physically correct
- Does not require artist animated wrinkle weights
  - Give up animation control
  - Gain automation

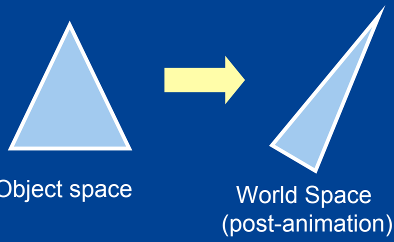
face tomorrow SIGGRAPH2007

There are applications where you want surfaces that exhibit wrinkles but you don’t want to spend the time hand animating them. A great example is clothing. Your clothing moves as you move and it stretches and compresses and creates wrinkles and folds. But there’s no “uncanny valley” associated with clothing, as viewers we don’t look to clothing wrinkles to disambiguate a character’s motivation or emotional expression. You don’t need to have super fine grain control over its animation like you’d want for a character’s face. For many applications, all that’s important is that wrinkles seem plausible. I refer to these kinds of wrinkles as “dynamic” because they don’t rely on pre-scripted animation and thus they don’t need as much artist time to setup.



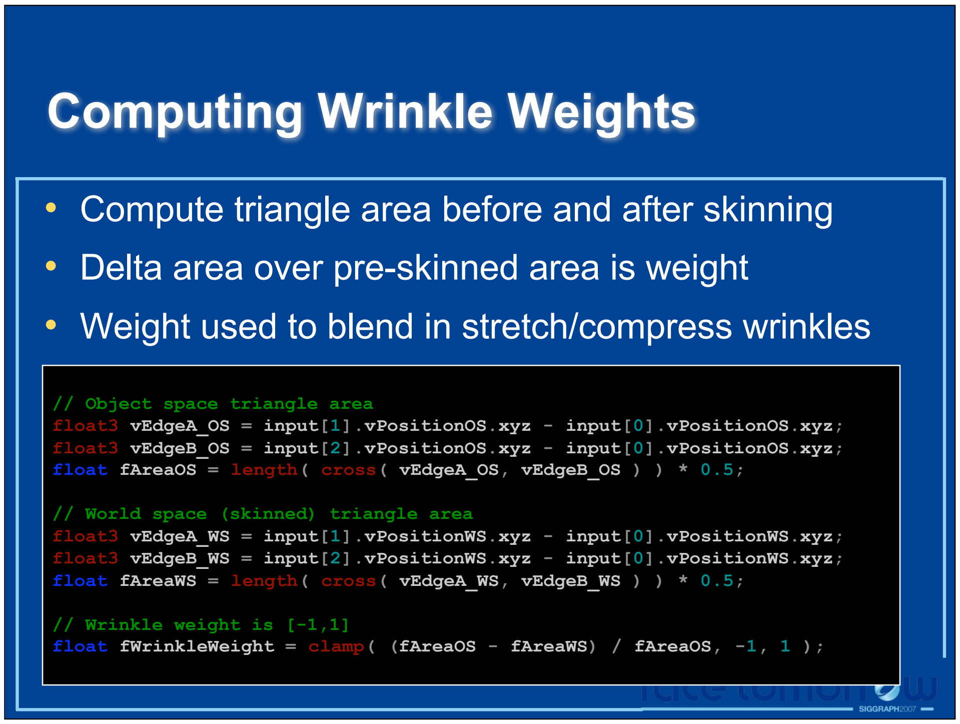
Computing Wrinkle Weights

- Compute triangle area before and after skinning
- Delta area over pre-skinned area is weight
- Weight used to blend in stretch/compress wrinkles



From DirectX SDK Morph Sample

In order to get a more automated wrinkle system, we have to find a way to dynamically compute wrinkle weights rather than relying on artists to hand animate them (as we did with facial wrinkles). One method, which I first saw in a DirectX 10 SDK demo (Sparse Morph Target demo), is to use a geometry shader to compute per-triangle wrinkle weights. The weight is computed by calculating a triangle's area before and after skinning. You can then find the delta between these two triangle areas and use it to give you a sense of whether the triangle is stretching or compressing.

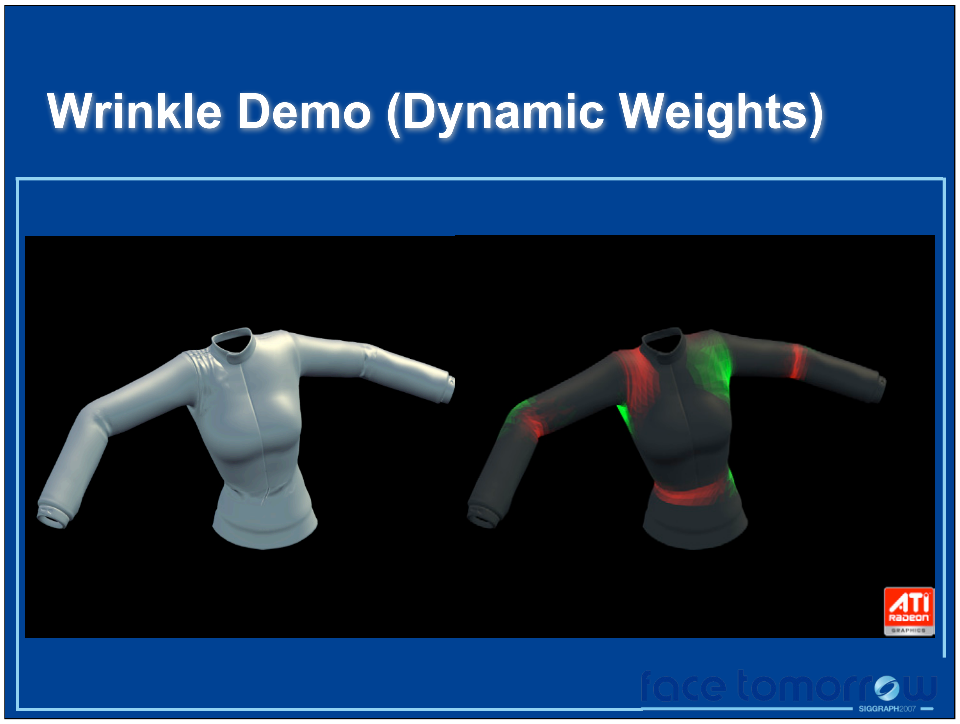


Computing Wrinkle Weights

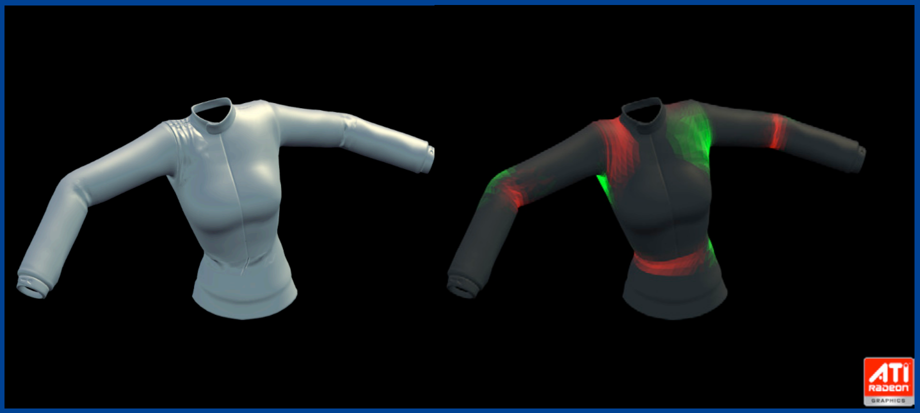
- Compute triangle area before and after skinning
- Delta area over pre-skinned area is weight
- Weight used to blend in stretch/compress wrinkles

```
// Object space triangle area
float3 vEdgeA_OS = input[1].vPositionOS.xyz - input[0].vPositionOS.xyz;
float3 vEdgeB_OS = input[2].vPositionOS.xyz - input[0].vPositionOS.xyz;
float fAreaOS = length( cross( vEdgeA_OS, vEdgeB_OS ) ) * 0.5;

// World space (skinned) triangle area
float3 vEdgeA_WS = input[1].vPositionWS.xyz - input[0].vPositionWS.xyz;
float3 vEdgeB_WS = input[2].vPositionWS.xyz - input[0].vPositionWS.xyz;
float fAreaWS = length( cross( vEdgeA_WS, vEdgeB_WS ) ) * 0.5;

// Wrinkle weight is [-1,1]
float fWrinkleWeight = clamp( (fAreaOS - fAreaWS) / fAreaOS, -1, 1 );
```

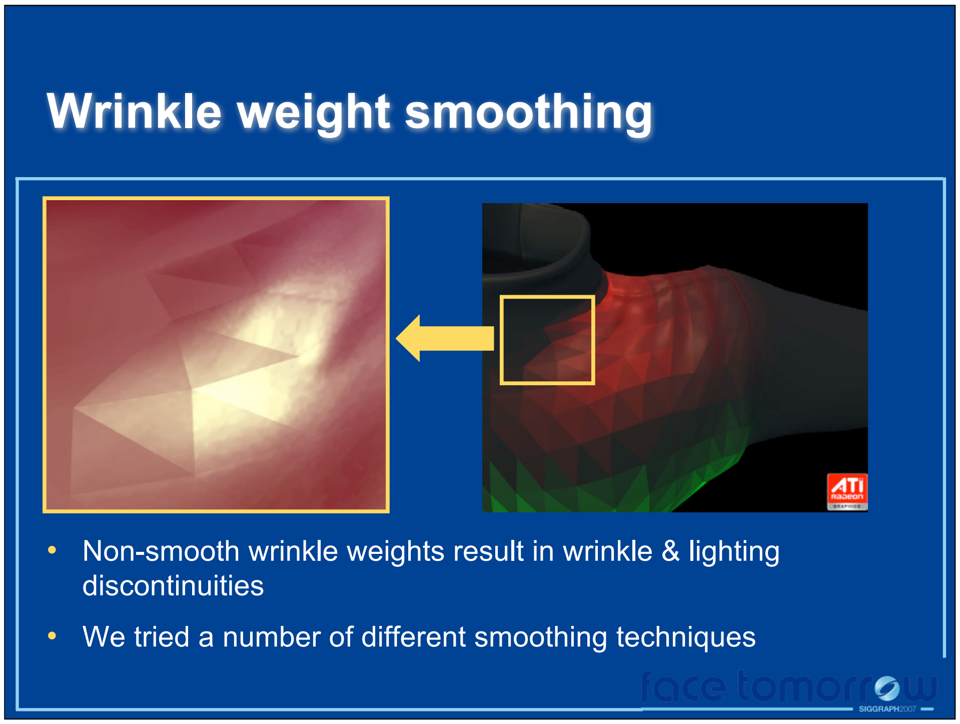
This can all be done in a geometry shader by passing the object-space and world-space vertex positions down from the vertex shader. You can then compute a cross product of two triangle edges to get the area of the triangle in object-space and world-space. Then subtract the world space area from the object space area and divide the difference by the object space area. The result is then clamped to the range -1 to 1 and this gives us our wrinkle weight. This is exactly the same kind of wrinkle weight that we used for animated facial wrinkles except this time there are no masks. The weight itself changes across the surface of the mesh in response to the mesh stretching or compressing so there's no need to have explicit wrinkle mask textures. Blending in the wrinkle maps works exactly as before. The wrinkle weight is -1 when the surface is stretching, 1 when the surface is compressing, and 0 when the area remains the same. Blending in your wrinkle maps works exactly as before. Of course this is just an approximation, it's possible that a triangle is stretching in one direction while compressing in another direction and the delta area might end up being 0 in which case you wouldn't see any wrinkles. But we aren't trying to get perfect wrinkle response here, we just want something quick and cheap that gives you an approximate, wrinkly experience.



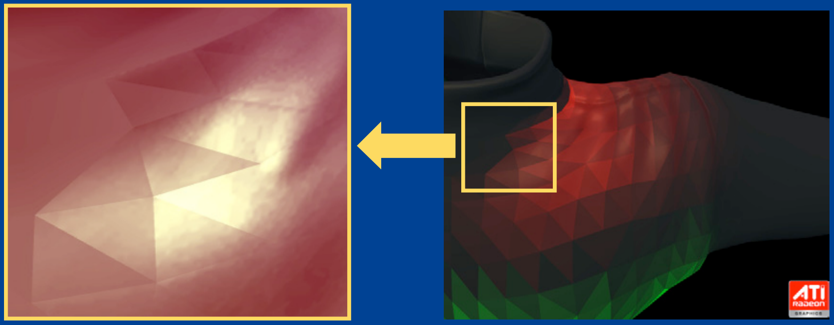
Wrinkle Demo (Dynamic Weights)



face tomorrow SIGGRAPH2007

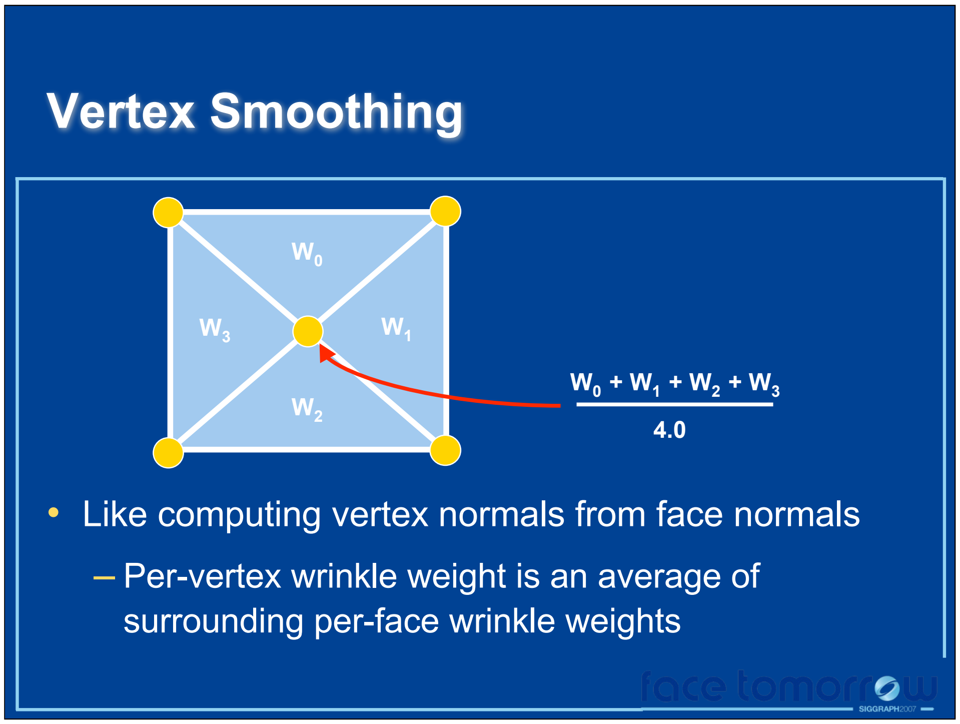


Wrinkle weight smoothing

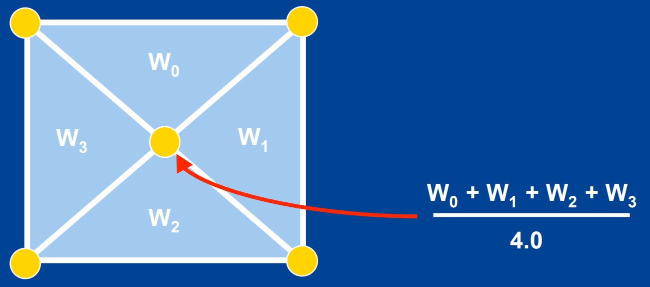


- Non-smooth wrinkle weights result in wrinkle & lighting discontinuities
- We tried a number of different smoothing techniques

The problem with the approach as I've described it so far is that the wrinkle weights we compute in the geometry shader are computed per-triangle which results in wrinkle weight discontinuities at triangle edges. So we need a way to smooth out the wrinkle weights. We tried a number of different smoothing techniques but in the end we found a vertex smoothing technique that worked the best. Note: texture smoothing techniques generally result in discontinuities as UV seams.

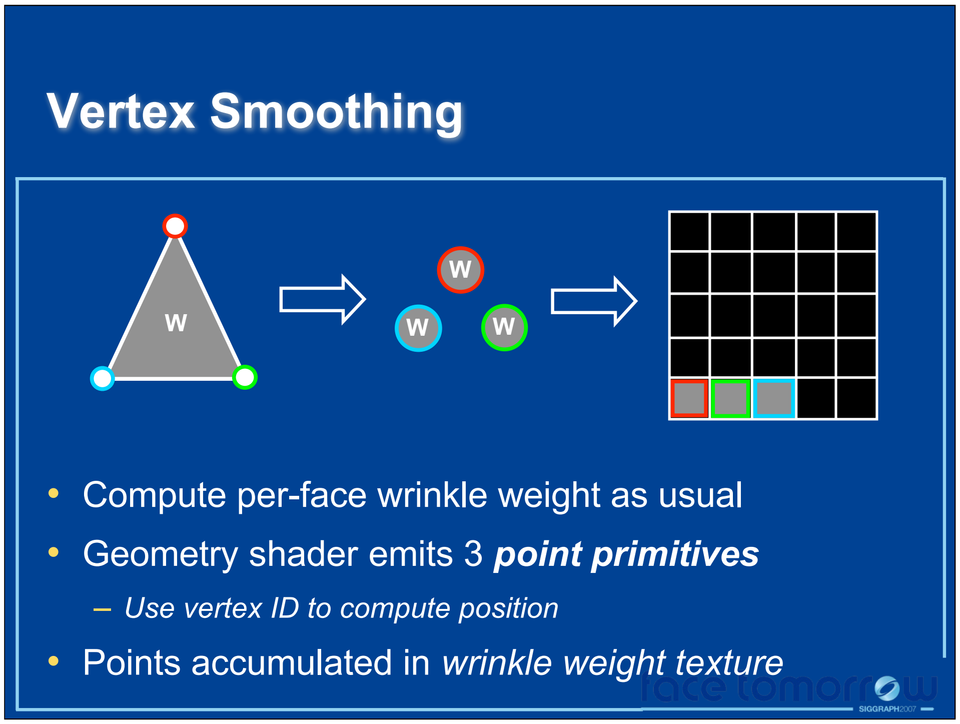


Vertex Smoothing

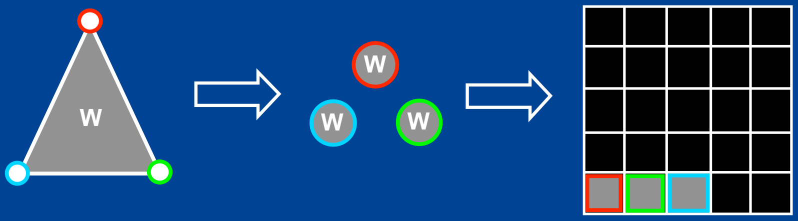


- Like computing vertex normals from face normals
  - Per-vertex wrinkle weight is an average of surrounding per-face wrinkle weights

The vertex smoothing technique works a lot like computing smooth vertex normals from triangle face normals. You basically compute a smooth per-vertex wrinkle weight by taking a weighted average of all the per-face wrinkle weights connected to that vertex. This method will be nice and smooth across UV seams (which would be problematic in a texture space smoothing approach) and can be computed completely on the GPU.

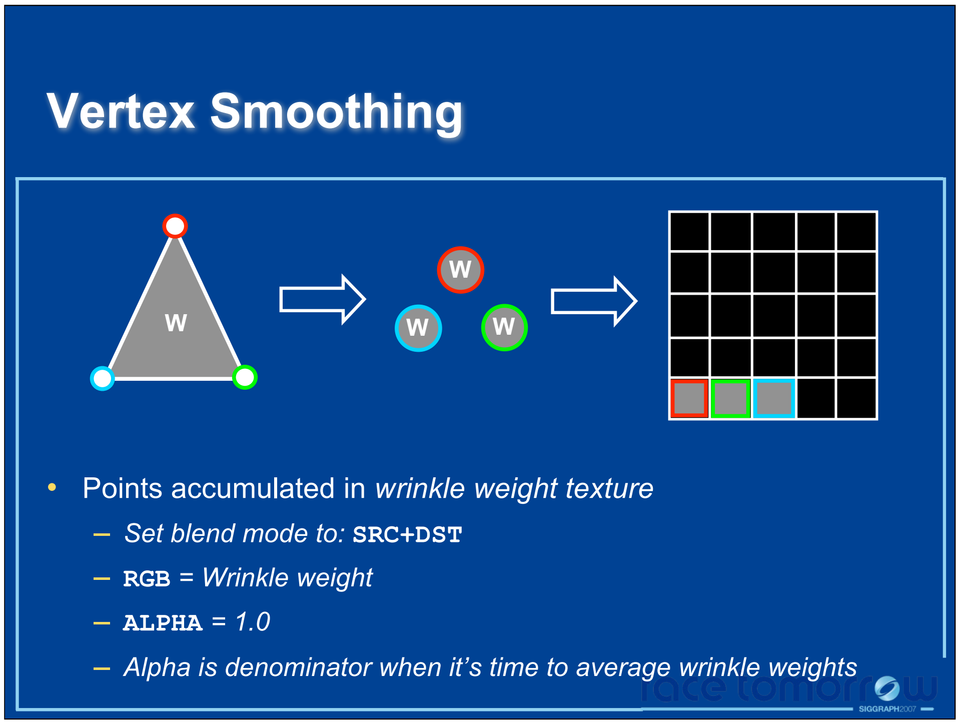


Vertex Smoothing

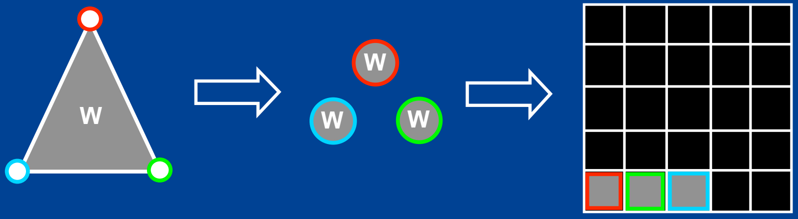


- Compute per-face wrinkle weight as usual
- Geometry shader emits 3 ***point primitives***
  - *Use vertex ID to compute position*
- Points accumulated in *wrinkle weight texture*

We start by computing a per-face wrinkle weight just as we did before by comparing the object-space and world-space triangle areas in the geometry shader. Instead of simply passing the triangle's wrinkle weight down to the pixel shader for final shading, we setup the geometry shader to emit three point primitives. There's one point primitive for each of the triangles vertices and they all get the triangle's wrinkle weight. These point primitives are 1 pixel x 1 pixel in size and we render them to an off-screen render target.

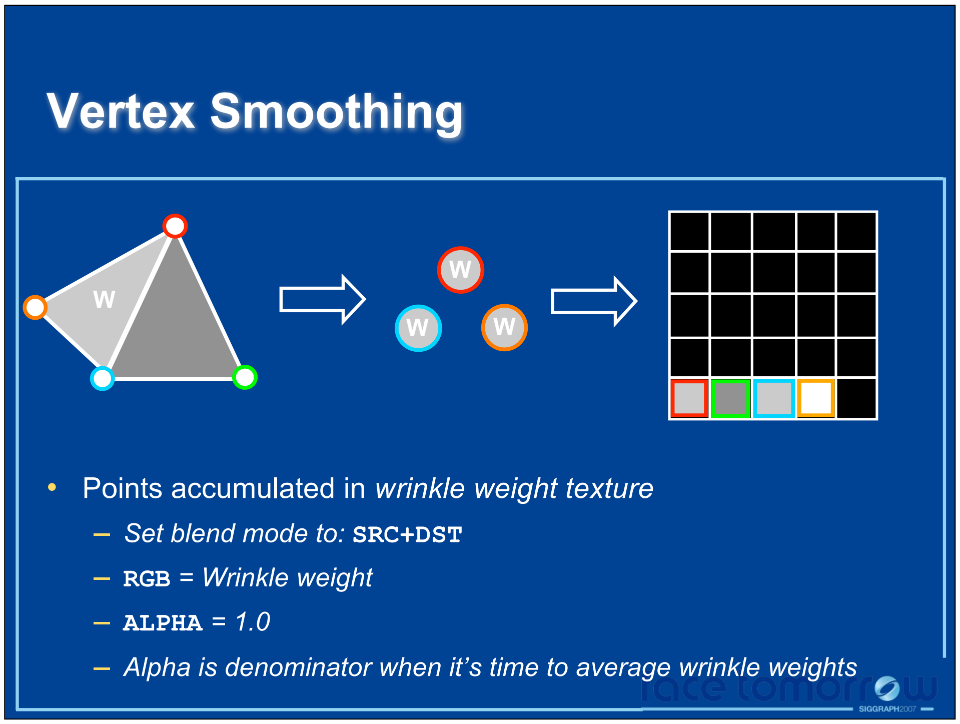


Vertex Smoothing

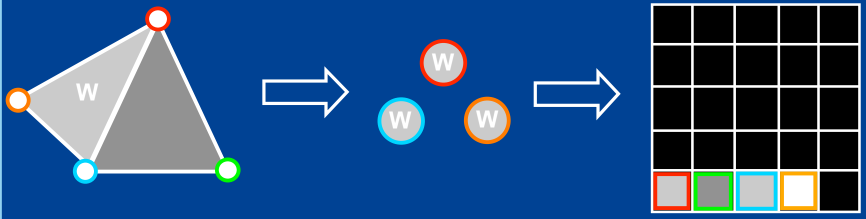


- Points accumulated in *wrinkle weight texture*
  - *Set blend mode to:* **`SRC+DST`**
  - **`RGB`** = *Wrinkle weight*
  - **`ALPHA`** = *1.0*
  - *Alpha is denominator when it's time to average wrinkle weights*

The render state is setup such that the point primitives are accumulated using the alpha blend unit. In the pixel shader, the wrinkle weight is put into the pixel's color channels and we simply put a 1 into the alpha channel. The alpha channel acts as a counter and keeps track of how many vertices fall into a given bin (aka pixel). The counter then acts as a denominator when computing an average weight.

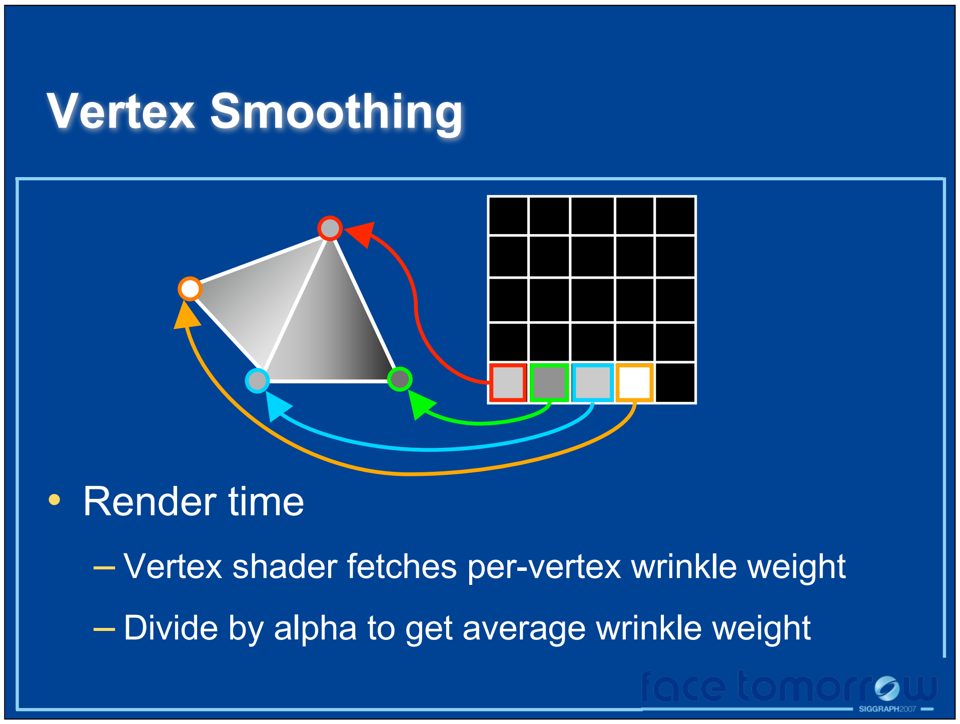


Vertex Smoothing

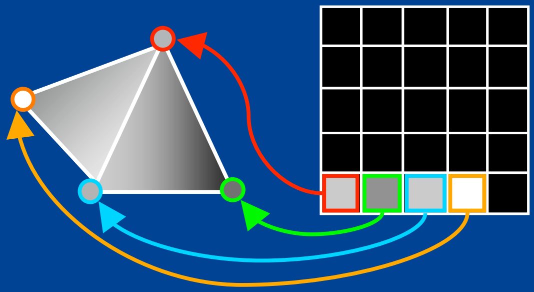


- Points accumulated in *wrinkle weight texture*
  - *Set blend mode to:* **`SRC+DST`**
  - **`RGB`** = *Wrinkle weight*
  - **`ALPHA`** = *1.0*
  - *Alpha is denominator when it's time to average wrinkle weights*

As more triangles pass through the pipeline, shared vertices (the red and blue vertices in the diagram) end up going to the same location in the off-screen render target and thus their weights all get accumulated into the same pixel location.   Because we've put weights into the color channel and 1 into the alpha channel of each point, in the end we get an accumulation of all the weights for a given vertex in the render target's color channels and the alpha channel will ultimately keep track of how many points landed on a given pixel. The average wrinkle weight for given vertex is then simply the accumulated weight (in the color channel) divided by the alpha channel.

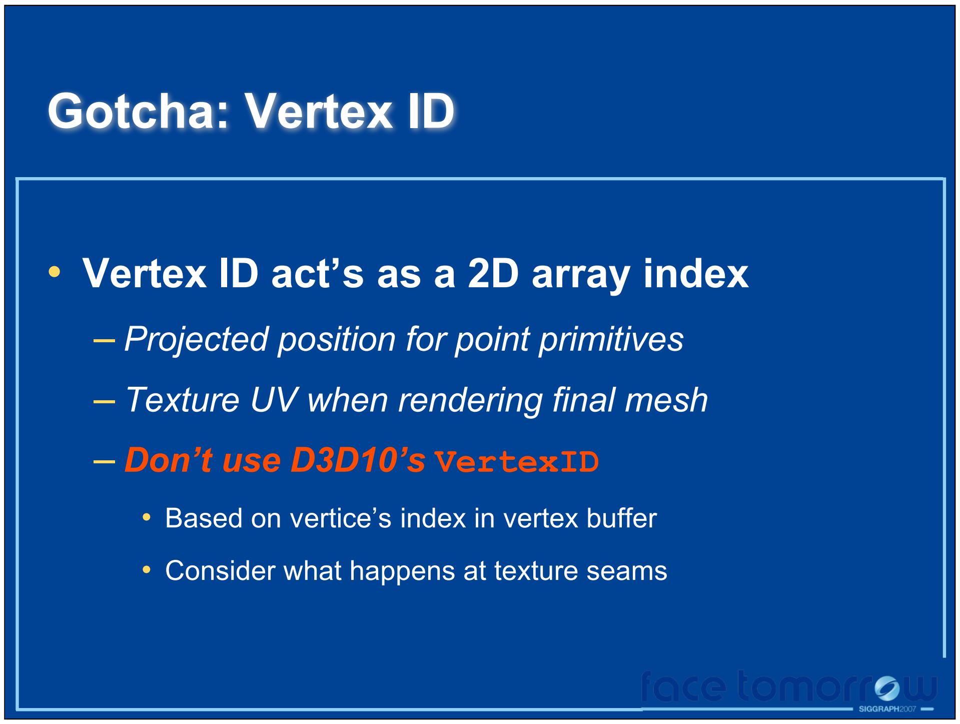


Vertex Smoothing



- Render time
  - Vertex shader fetches per-vertex wrinkle weight
  - Divide by alpha to get average wrinkle weight

When it comes time to render the actual mesh, in the vertex shader we can look up a smooth wrinkle weight for each vertex. The wrinkle weight is passed down to the pixel shader and so its get interpolated across each triangle and you end up with nice, smooth wrinkle weights across your mesh.

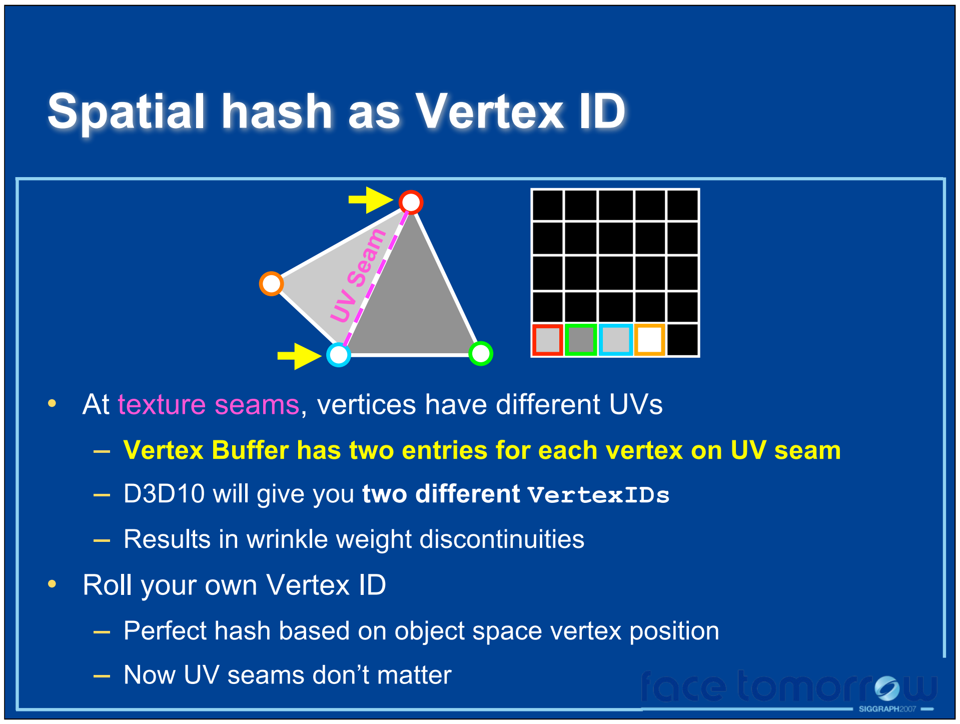


Gotcha: Vertex ID

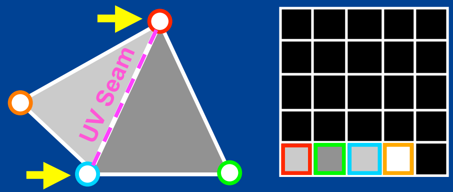
- **Vertex ID act's as a 2D array index**
  - *Projected position for point primitives*
  - *Texture UV when rendering final mesh*
  - ***Don't use D3D10's*** `VertexID`
    - Based on vertice's index in vertex buffer
    - Consider what happens at texture seams

face tomorrow
SIGGRAPH2007

Computing the point primitives's locations when rendering them to the off-screen render target: The obvious method of using D3D10's VertexID value, and doing some math based on the render-target's dimensions to turn the ID into a 2D array index (or screen space position), doesn't actually work out as you might expect. The problem is that D3D10's Vertex ID is that vertex's Vertex Buffer index. In other words, the VertexID is that vertex's position in its Vertex Buffer. This is a problem because at UV seams your Vertex Buffer must contain two separate vertices.

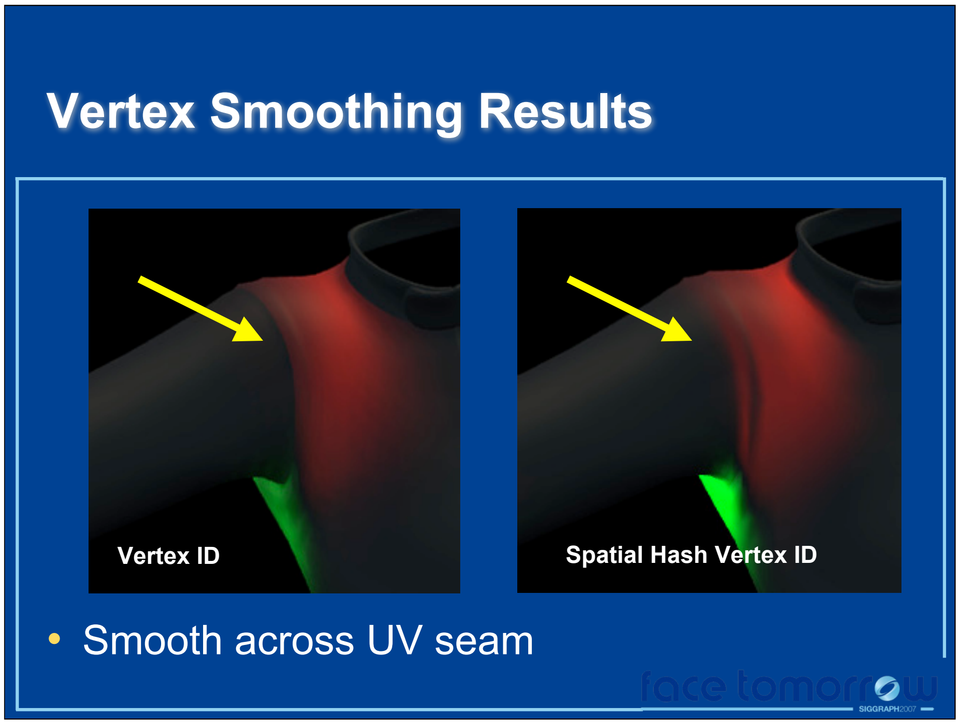


Spatial hash as Vertex ID

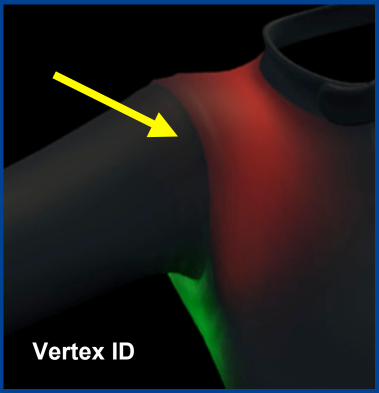


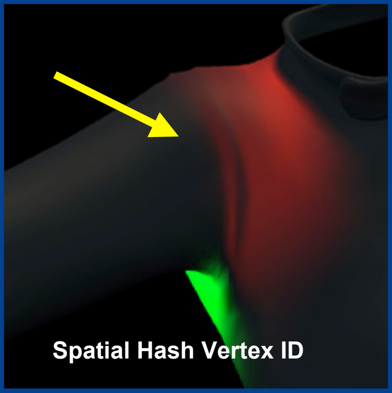
- At texture seams, vertices have different UVs
  - **Vertex Buffer has two entries for each vertex on UV seam**
  - D3D10 will give you **two different `VertexIDs`**
  - Results in wrinkle weight discontinuities
- Roll your own Vertex ID
  - Perfect hash based on object space vertex position
  - Now UV seams don't matter

For example, the vertices marked with yellow arrows lie on a UV seam. The UV coordinates are different on either side of this seam and therefore these two triangles do not actually share any vertices as far as the vertex buffer is concerned. These vertices may have the same object-space positions but because there is a UV space discontinuity here, there must be two red vertices and two blue vertices in the vertex buffer (each with different UV coordinates). Thus the VertexID that D3D10 gives you will be different and the red and blue marked vertices will not always be accumulated in the same bins. This results in the same artifact that the texture smoothing approach had, there will be wrinkle weight discontinues here because the wrinkle weights will not smooth across this boundary. The way we worked around this issue was by adding support to our mesh preprocessor for Vertex IDs based on a perfect spatial hash. We simply iterate over the vertices in our mesh and give vertices with different object space positions unique IDs without looking at any of the other vertex attributes (such as UVs). So as long as two vertices have the same object space position, they will get the same spatial hash ID. This results in the red and blue vertices always going to their respective bins regardless whether they come in as part of the light gray triangle or the dark gray triangle.

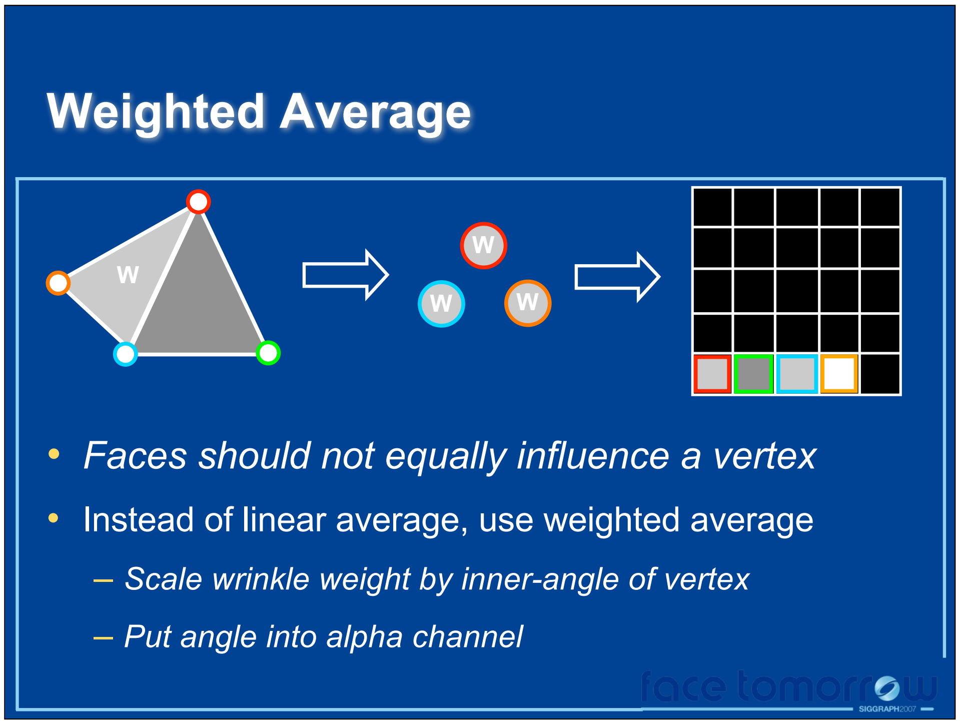


Vertex Smoothing Results

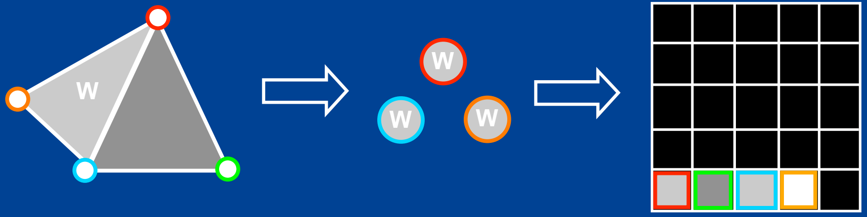




- Smooth across UV seam

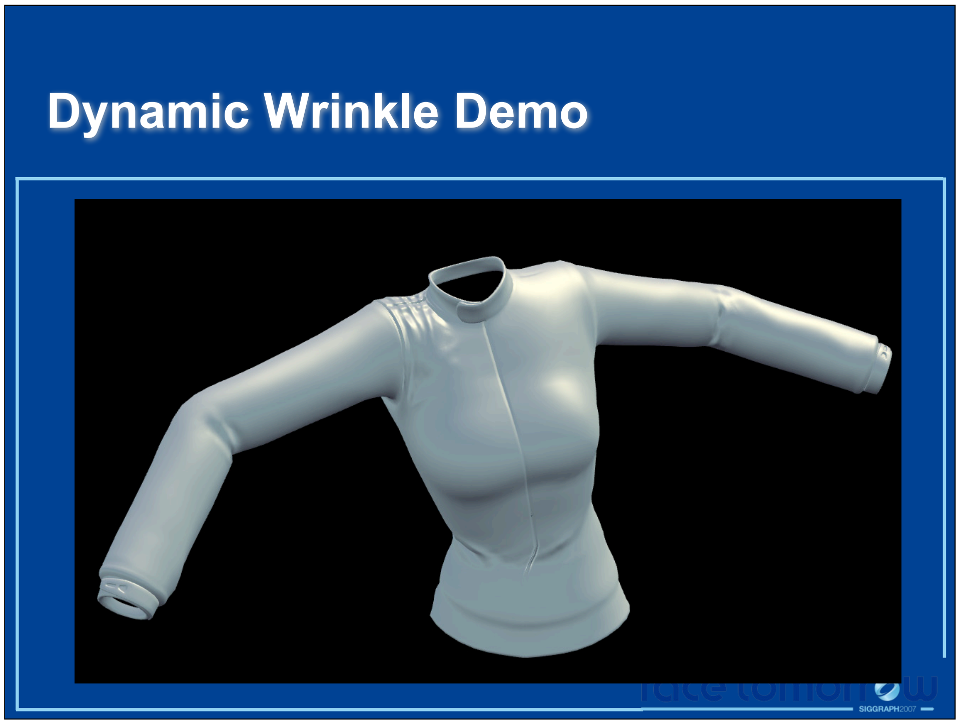


Weighted Average



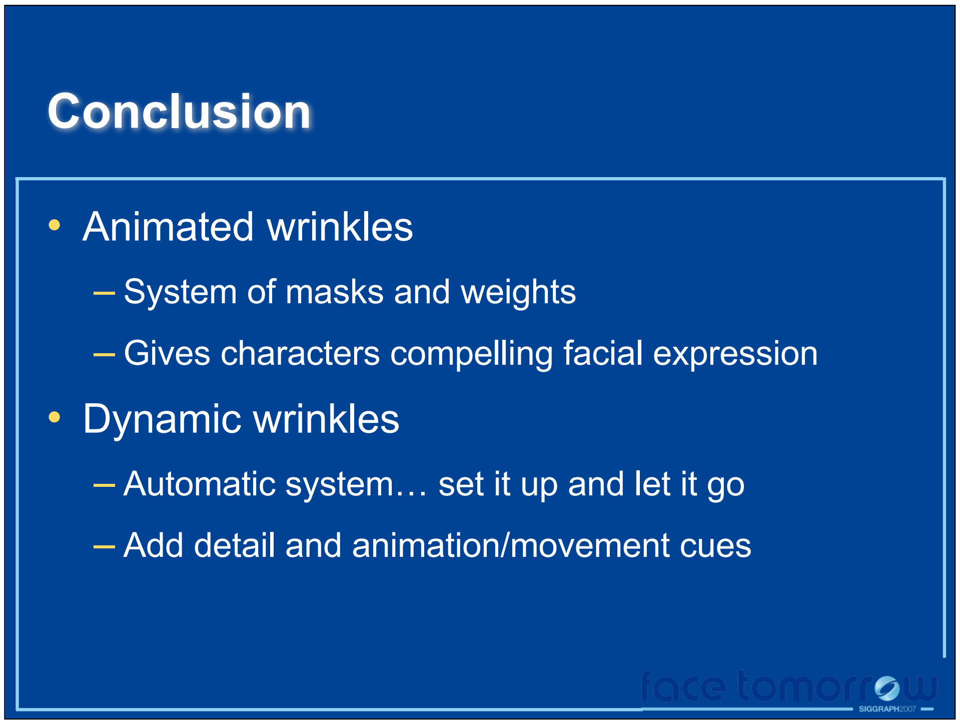
- *Faces should not equally influence a vertex*
- Instead of linear average, use weighted average
  - *Scale wrinkle weight by inner-angle of vertex*
  - *Put angle into alpha channel*

Earlier I told you that when accumulating the wrinkle weights in the off-screen render target that you should put a 1 into the alpha channel to keep track of how many vertices fell into a given bin. If you do that you will essentially compute a linear average of all the wrinkle weights that fall into a given bin. We found that we got slightly better results by weighting the wrinkle weight that goes out with each vertex (or point primitive) by the triangles inner angle at that vertex. You then put this same angle weight into the alpha channel and ultimately end up dividing the accumulated, weighted wrinkle weights by the accumulated angle weights.



Dynamic Wrinkle Demo

Demo again with vertex smoothing of wrinkle weights.



Conclusion

- Animated wrinkles
  - System of masks and weights
  - Gives characters compelling facial expression
- Dynamic wrinkles
  - Automatic system… set it up and let it go
  - Add detail and animation/movement cues

face tomorrow
SIGGRAPH2007

Questions?

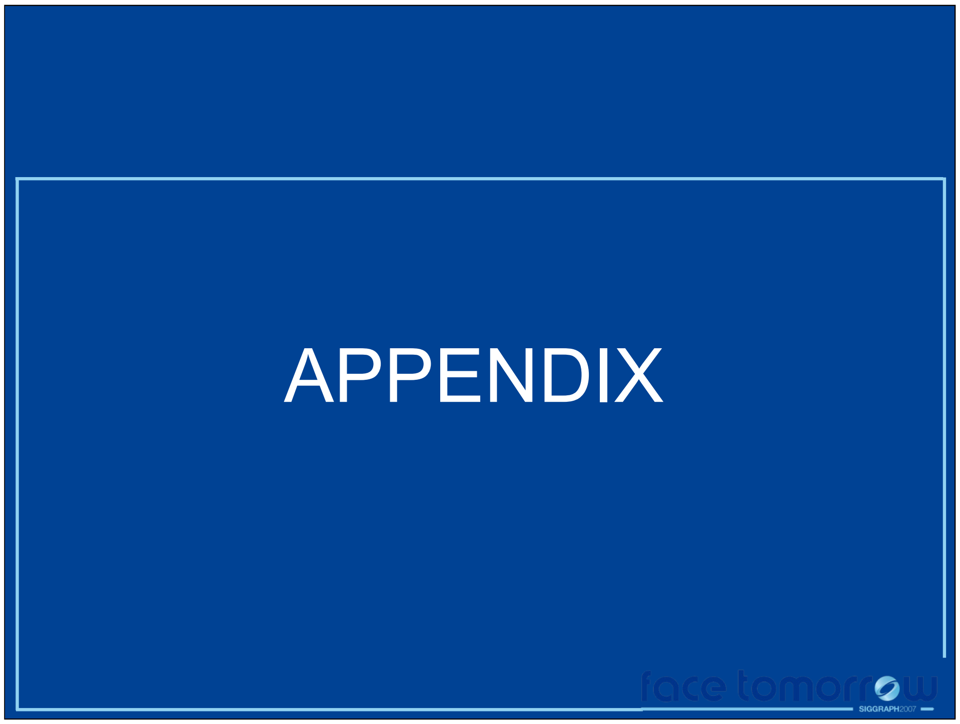
Email Me:

chris.oat@amd.com

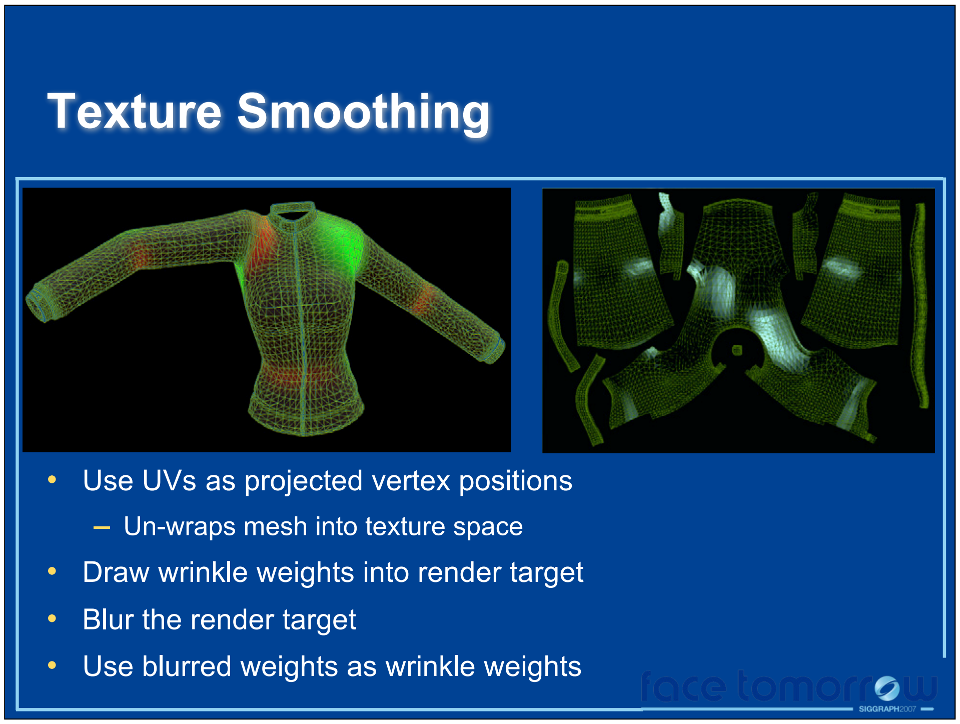
Download Slides & Course Notes:

http://www.ati.amd.com/developer

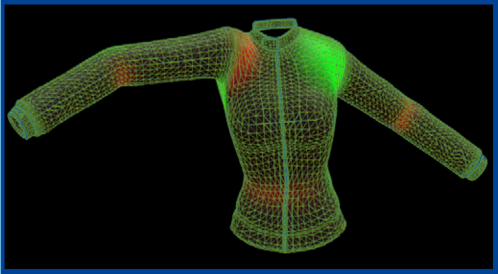
face tomorrow SIGGRAPH2007

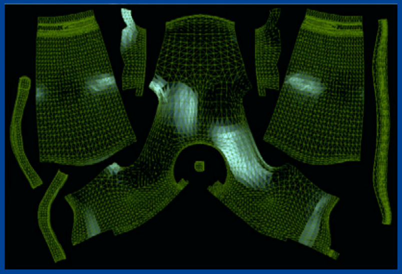


APPENDIX



Texture Smoothing

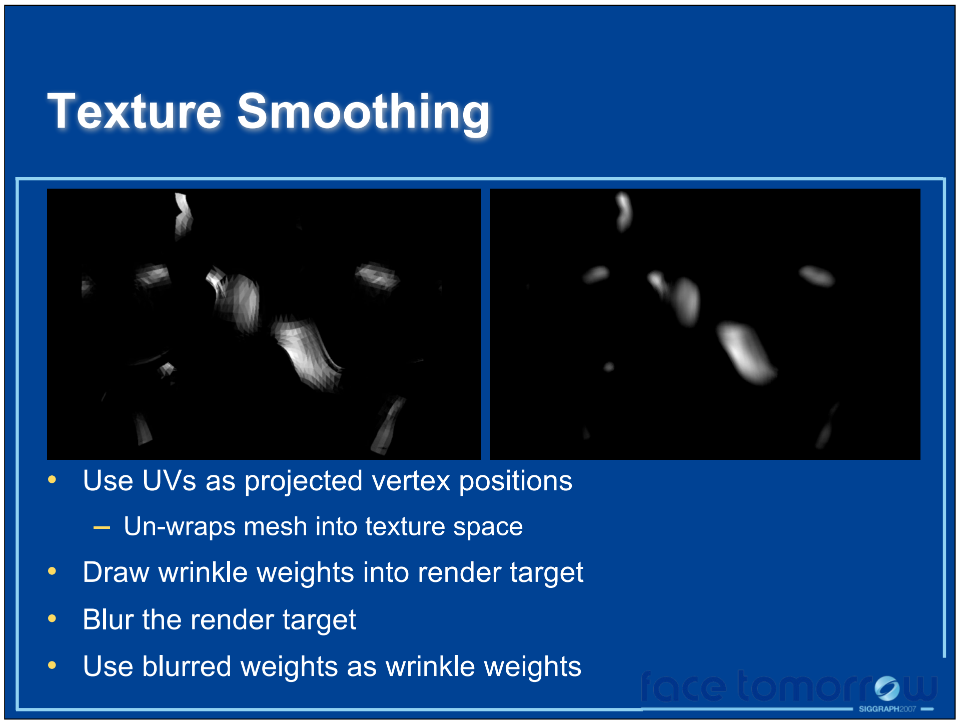




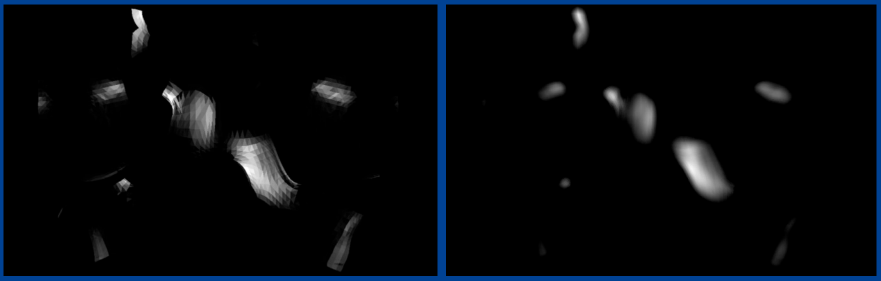
- Use UVs as projected vertex positions
  - Un-wraps mesh into texture space
- Draw wrinkle weights into render target
- Blur the render target
- Use blurred weights as wrinkle weights

face tomorrow SIGGRAPH2007

The first smoothing method we tried was a very simple, straight forward texture space smoothing approach. This smoothing technique works by un-wrapping the mesh and rendering the per-triangle wrinkle weights into the mesh’s texture space. This is pretty easy to do, you simply scale and bias your UVs so that they are in the range [-1,1] and then use them as projected vertex positions.



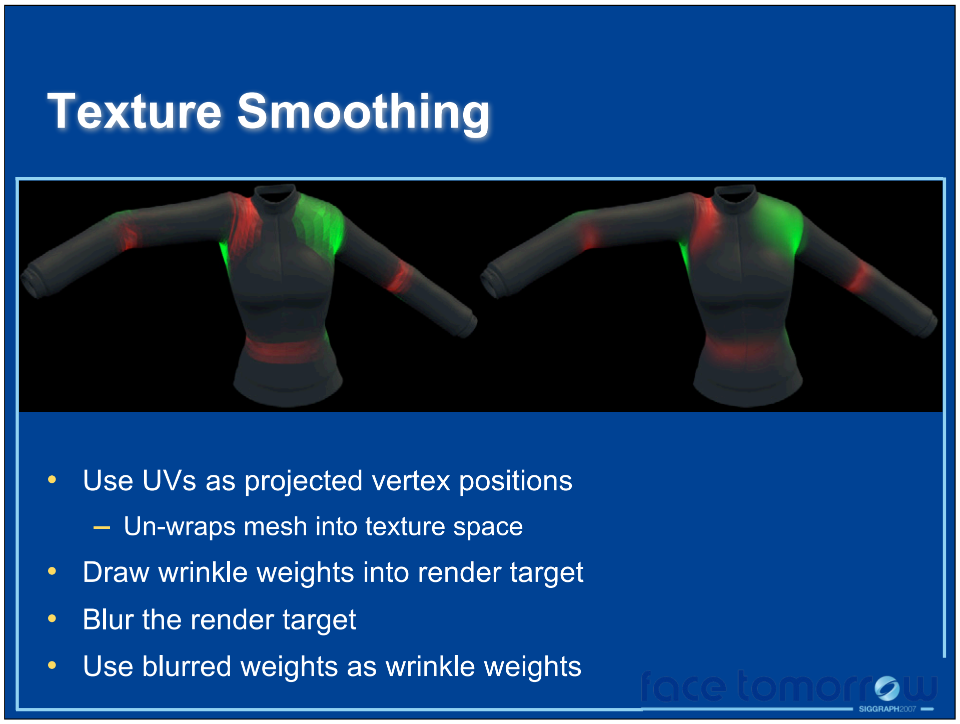
Texture Smoothing



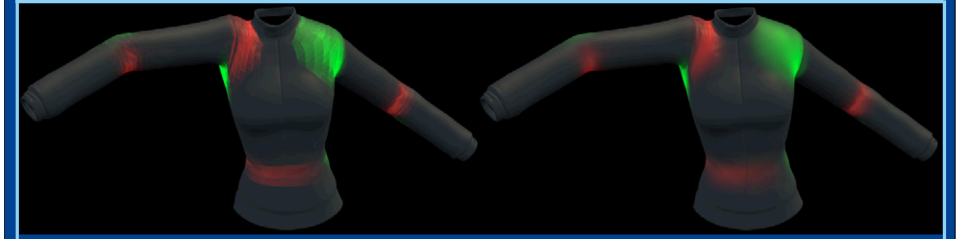
- Use UVs as projected vertex positions
  - Un-wraps mesh into texture space
- Draw wrinkle weights into render target
- Blur the render target
- Use blurred weights as wrinkle weights

face tomorrow SIGGRAPH2007

Once you have the per-triangle wrinkle weights in a texture, you can apply your favorite blurring technique to smooth out the wrinkle weights. As you can see here, we were able to eliminate all the faceting seen in the original wrinkle weight texture (on the left). The texture on the right was generated using a 13 tap Poisson disc blur kernel.



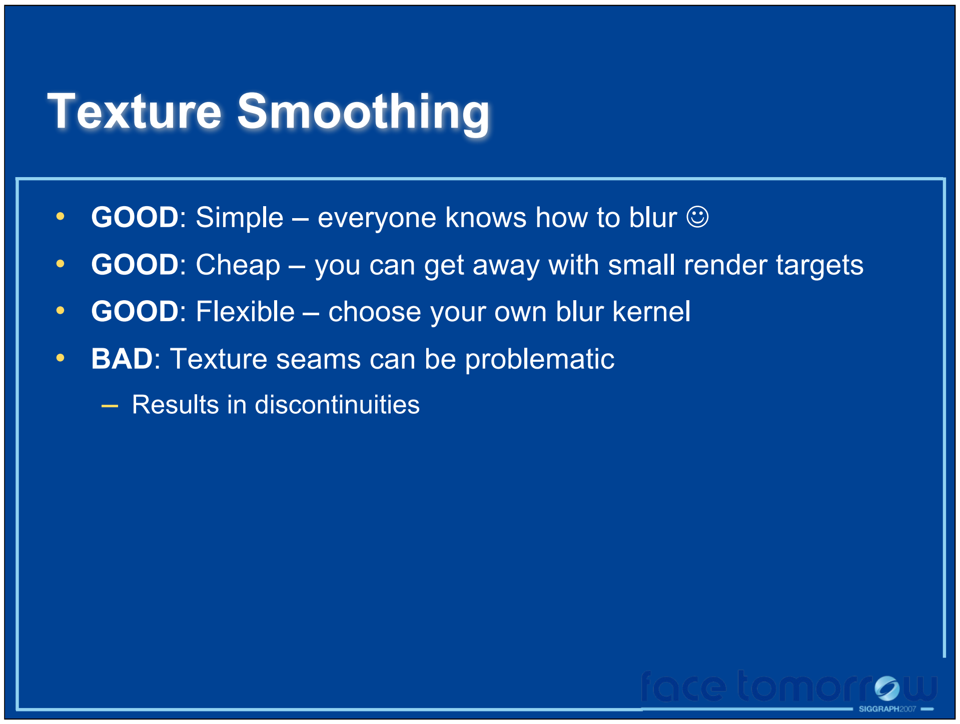
Texture Smoothing



- Use UVs as projected vertex positions
  - Un-wraps mesh into texture space
- Draw wrinkle weights into render target
- Blur the render target
- Use blurred weights as wrinkle weights

face tomorrow SIGGRAPH2007

We can then apply our blurred wrinkle map texture to mesh and you can see that the wrinkle weights are nice and smooth. You don't see any of those harsh discontinuities that you saw before.



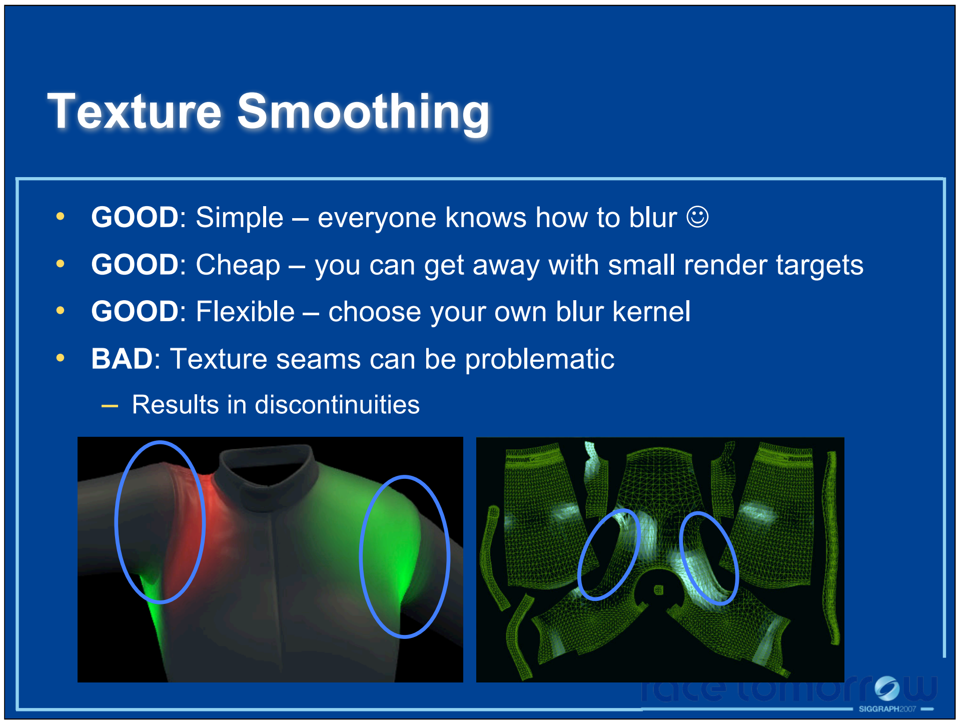
Texture Smoothing

- **GOOD**: Simple – everyone knows how to blur ☺
- **GOOD**: Cheap – you can get away with small render targets
- **GOOD**: Flexible – choose your own blur kernel
- **BAD**: Texture seams can be problematic
  - Results in discontinuities

face tomorrow SIGGRAPH2007

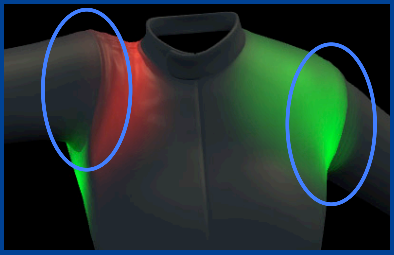
The first smoothing method we tried was a very simple, straight forward texture space smoothing approach. Once you have the geometry shader that computes the per-triangle wrinkle weights, extending your system to include a texture space smoothing step is fairly straightforward. This is also rather inexpensive addition to the basic non-smooth wrinkle shader. You can actually get away with very low resolution wrinkle weight textures so the additional cost of rendering to the renderable texture and then blurring and sampling the final smoothed wrinkle weights incurs a fairly nominal performance cost.

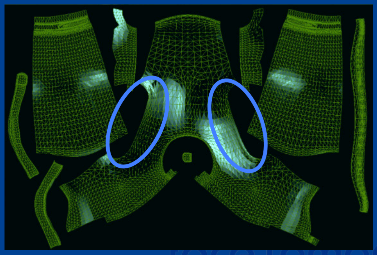
The one problem we had with this technique was that it doesn't deal with UV seams very well. On some of our meshes it worked fine (which is why I wanted to include it in my talk today) but there were some cases where UV seams did result in some visual disconinuities. The problem is that adjacent surface regions on the mesh are not necessarily adjacent in texture UV space.



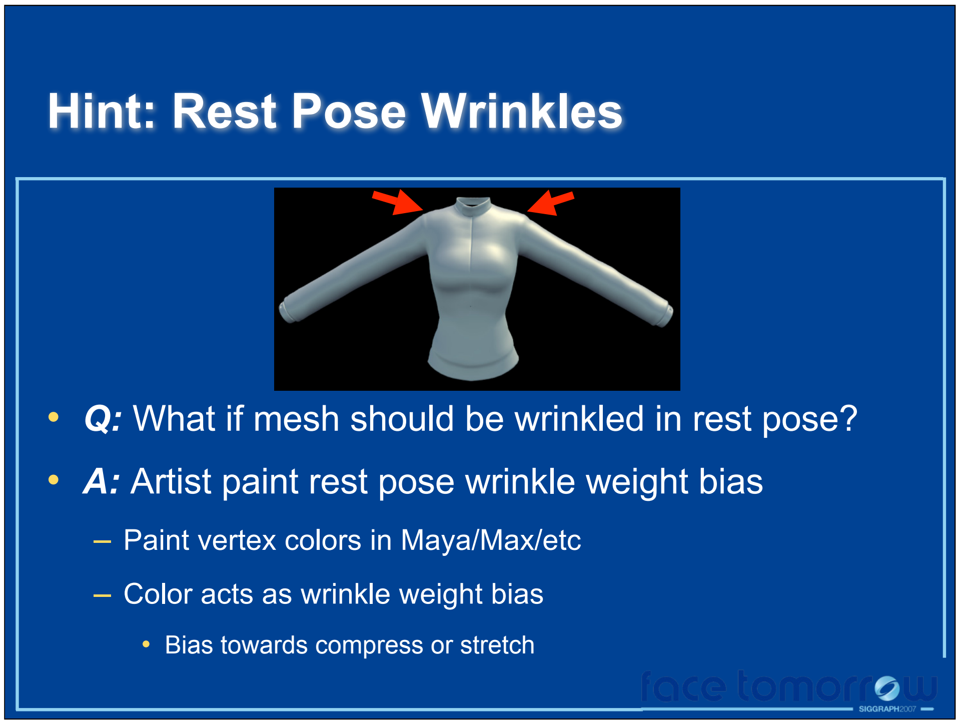
Texture Smoothing

- **GOOD**: Simple – everyone knows how to blur ☺
- **GOOD**: Cheap – you can get away with small render targets
- **GOOD**: Flexible – choose your own blur kernel
- **BAD**: Texture seams can be problematic
  - Results in discontinuities

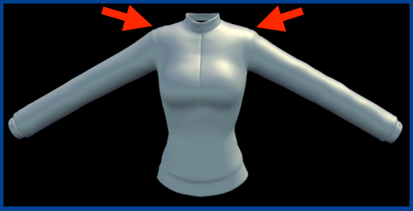




For example there are UV seams near the jacket's shoulders and so wrinkle weights don't get blurred across the shoulders and onto the upper arms of the jacket. This results in lighting discontinuities once you blend the wrinkle map into the normal map and use it for lighting and it can be a rather severe artifact in some situations. So again, if you don't have a lot of UV seams or if your UV seams are not in prominent places, this very well may not be an issue for you. If these seams do create a visual artifacts for you (as they did for us on Ruby's jacket) then you can use a vertex smoothing technique that completely ignores UVs.



Hint: Rest Pose Wrinkles



- ***Q:*** What if mesh should be wrinkled in rest pose?
- ***A:*** Artist paint rest pose wrinkle weight bias
  - Paint vertex colors in Maya/Max/etc
  - Color acts as wrinkle weight bias
    - Bias towards compress or stretch

face tomorrow SIGGRAPH2007

One last snag that we ran into, which is independent of the smoothing technique that you ultimately choose, is that the rest pose of your mesh (i.e. the object space pose of your mesh) may be such that you expect the surface to be wrinkled. For example, in the image here we see the jacket in its rest pose. Any time the mesh passes through this pose, the geometry shader will compute a wrinkle weight of 0 but in fact there are places on this mesh where we might expect there to be wrinkles even when its in this rest pose. One solution to this is to allow artists to paint vertex colors into the mesh that essentially act as wrinkle weight biases. So in this jacket mesh you might paint a wrinkle bias into the shoulder regions so even in the rest pose there will be wrinkles. Its not until the mesh is oriented such that the triangles in the shoulder regions expand in area that we get “no wrinkles” in the shoulder areas.

We actually just ended up ignoring this issue in the end but I wanted to mention it just incase it’s a significant issue for you.