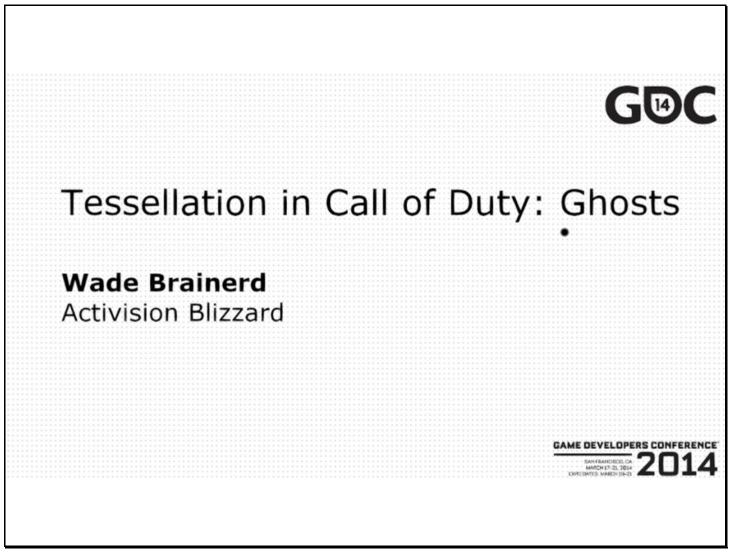
Slide 1



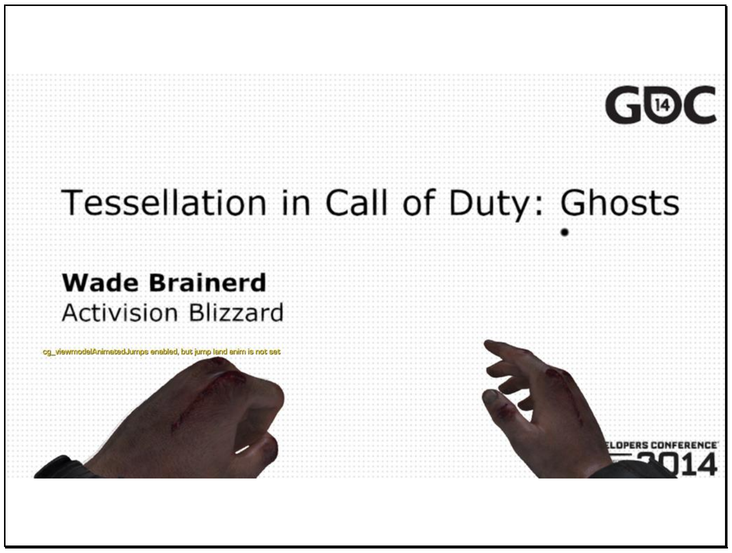
Tessellation in Call of Duty: Ghosts

Wade Brainerd
Activision Blizzard

GAME DEVELOPERS CONFERENCE
2014

This talk was presented at GDC 2014 as part of the Advanced Visual Effects with DirectX 11 tutorial.

Slide 2



Tessellation in Call of Duty: Ghosts

Wade Brainerd
Activision Blizzard

```
cg_viewmodelAnimatedJumps enabled, but jump land anim is not set
```

Note that the entire presentation was given inside the game engine, so these slides are simply a series of screenshots.

The information conveyed here is not as good as what was given live, so if you have Vault access, I encourage you to watch the video instead.

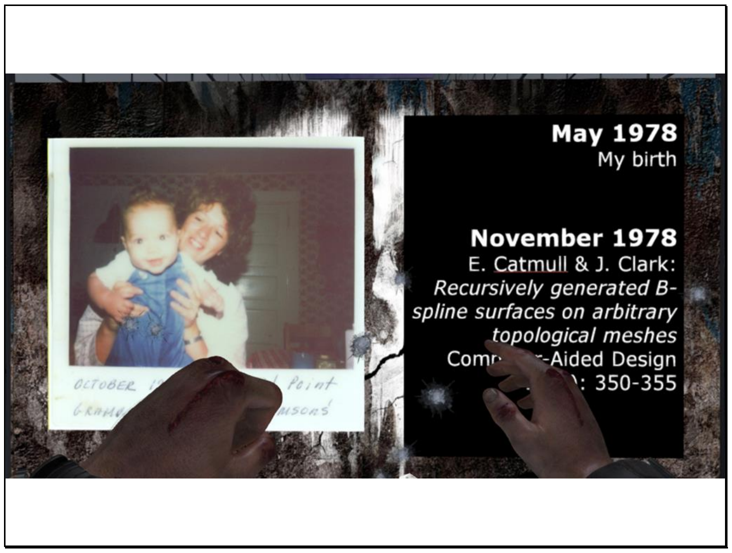
Slide 3

Call of Duty: Ghosts was a launch title for the new generation of consoles, and was also Infinity Ward’s first DirectX 11 title.

We used tessellation in two different parts of our graphics pipeline: On models, Catmull Clark Subdivision Surfaces, and on terrain, Displacement maps.

The first part of the presentation covers Catmull Clark Subdivision Surfaces in detail. The second part is a brief overview of our terrain displacement maps.

Slide 4

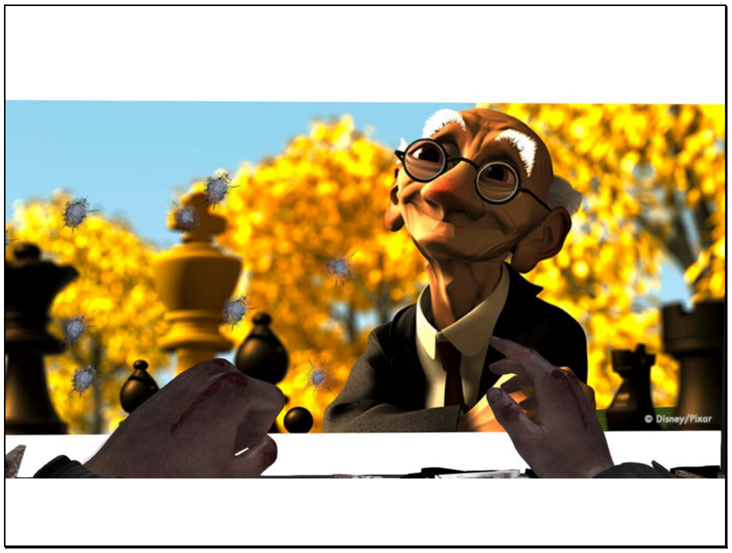


May 1978
My birth

November 1978
E. Catmull & J. Clark:
Recursively generated B-spline surfaces on arbitrary topological meshes
Comp[illegible]r-Aided Design
[illegible]: 350-355

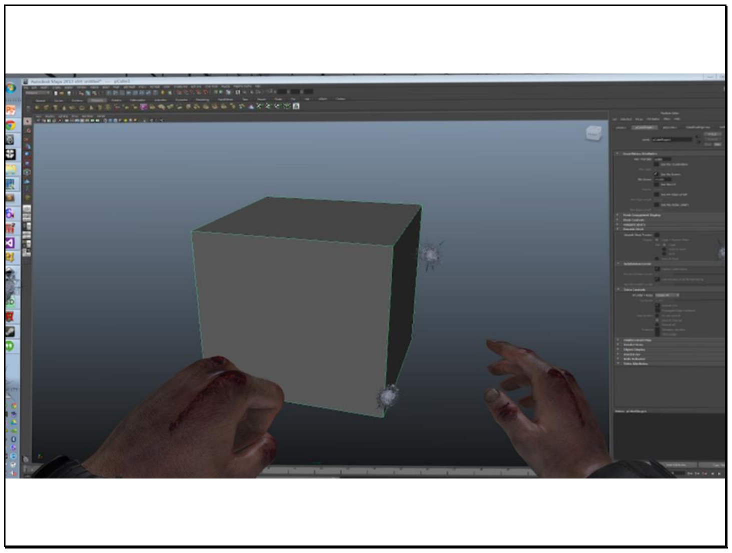
Catmull Clark Subdivision Surfaces, or SubDs for short, were invented by Ed Catmull and Jim Clark and published in 1978. I was about 6 months old at the time.

Slide 5



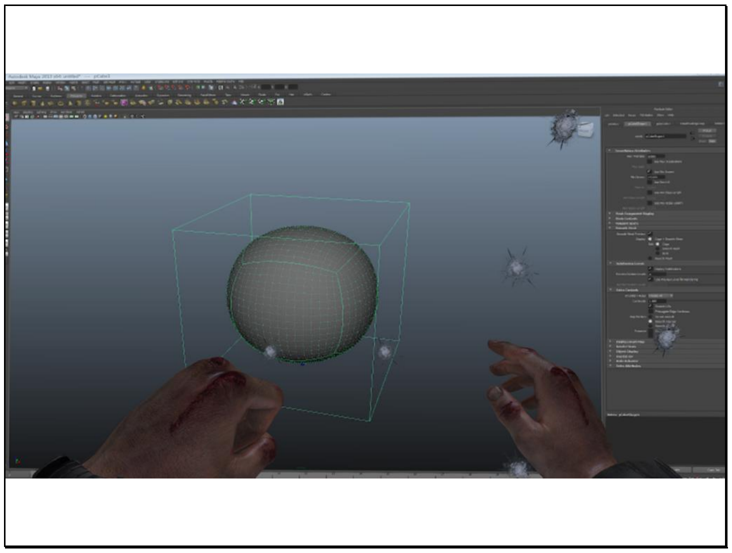
Subdivision surfaces are popular in the film industry for their ease of authoring and low memory cost. For example, many of Pixar's characters and props are modeled as subdivision surfaces.

Slide 6



A subdivision surface is defined by a control mesh. The control mesh can have faces with any number of vertices, though quads are the most natural. Only vertex positions and the control mesh topology contribute to the final surface.

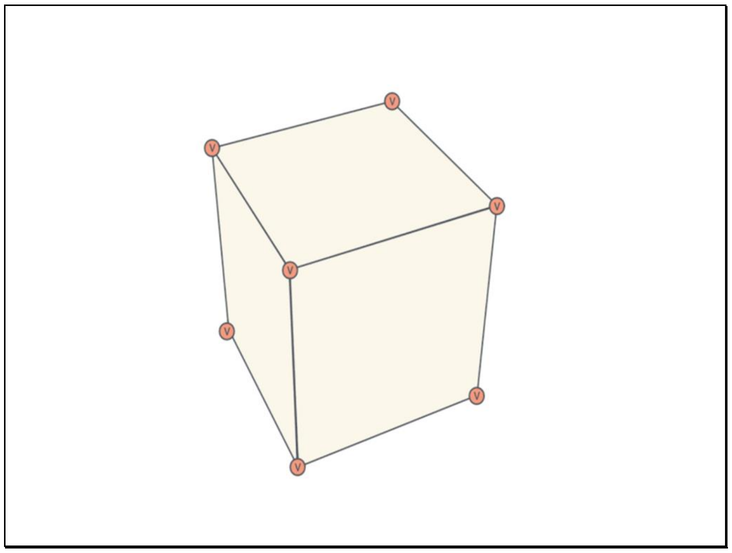
Slide 7



One way to make a subdivision surface is to click on an object in Maya and press the number 2 on your keyboard. Whatever you have selected will be smoothed using Maya's version of the SubD rules. You may then edit the control mesh and watch Maya's viewport interactively update the result.

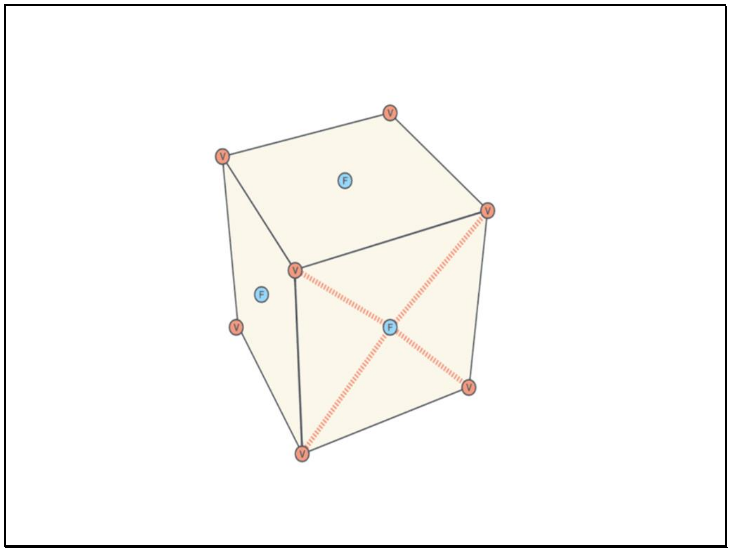
We implemented SubD support on Ghosts at the request of our art lead, Jake Rowell. We agreed up front that our pipeline would be based around Maya, that our implementation should match theirs as closely as possible.

Slide 8



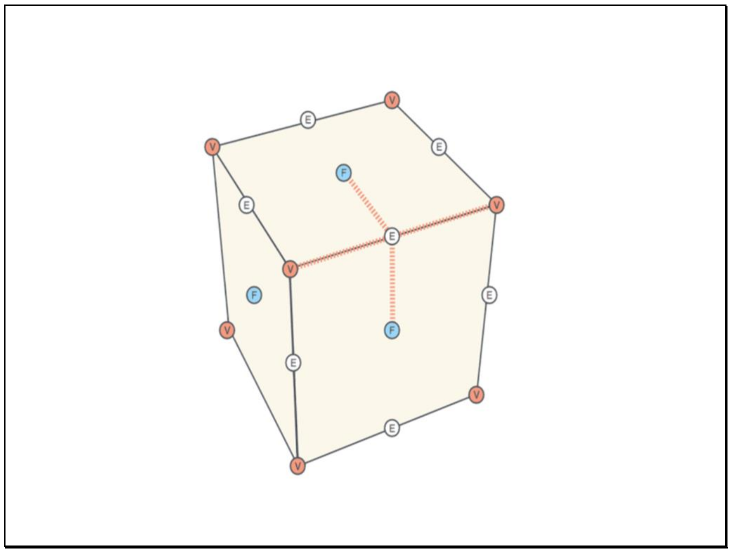
To evaluate a SubD, you repeatedly apply a series of simple rules to the control mesh. This is called global subdivision.

Slide 9



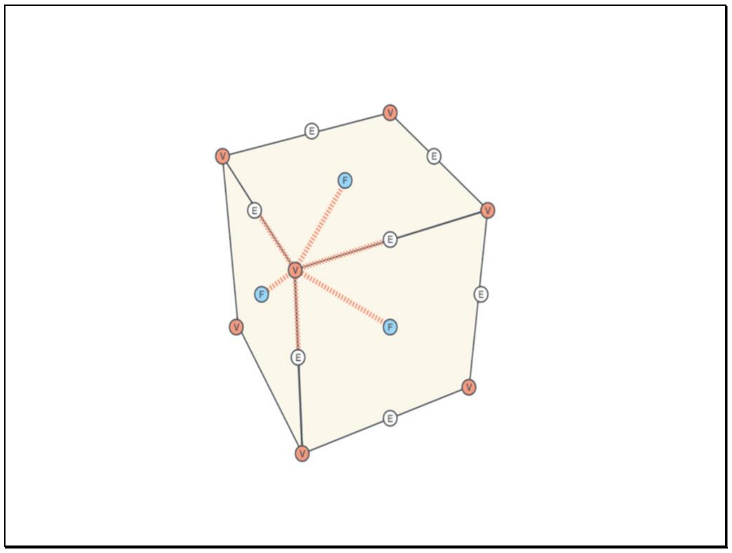
First, you insert a new vertex at the center of each face, called a face point. These points are the average of all the vertices in the face.

Slide 10



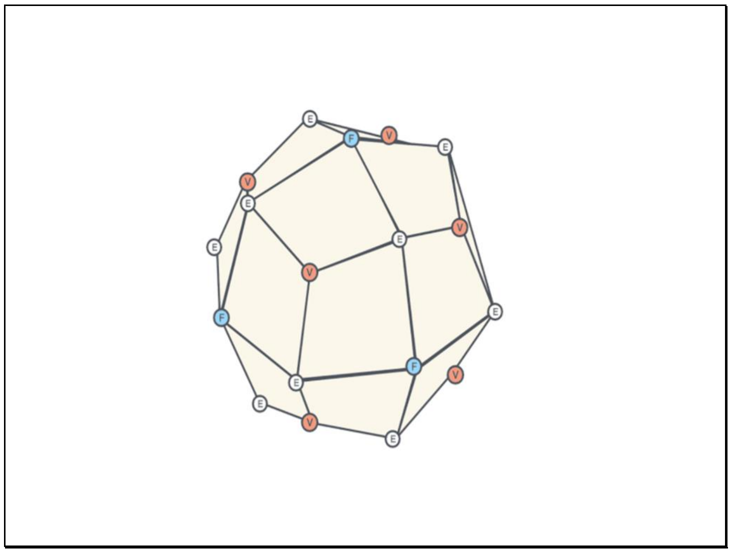
Then you insert one along each edge, weighted by the edge's vertex endpoints and neighboring face points; these are called edge points.

Slide 11



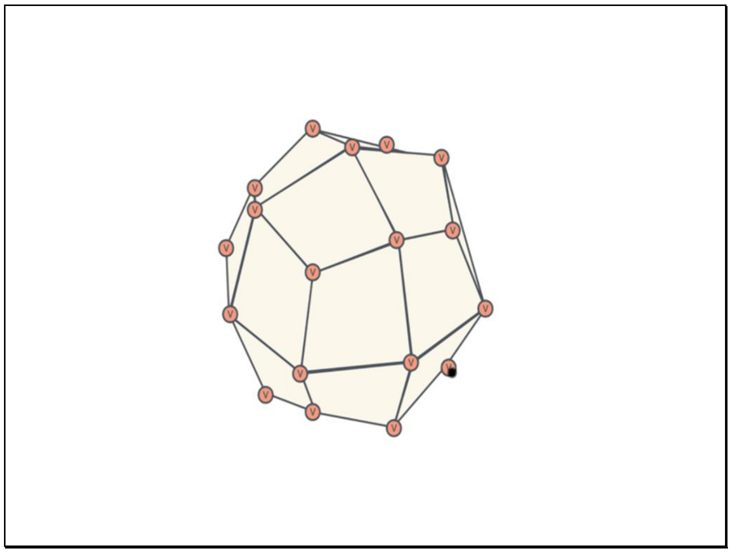
Last, you move the original vertices toward their adjacent edge endpoints and face point neighbors. These are now called vertex points.

Slide 12



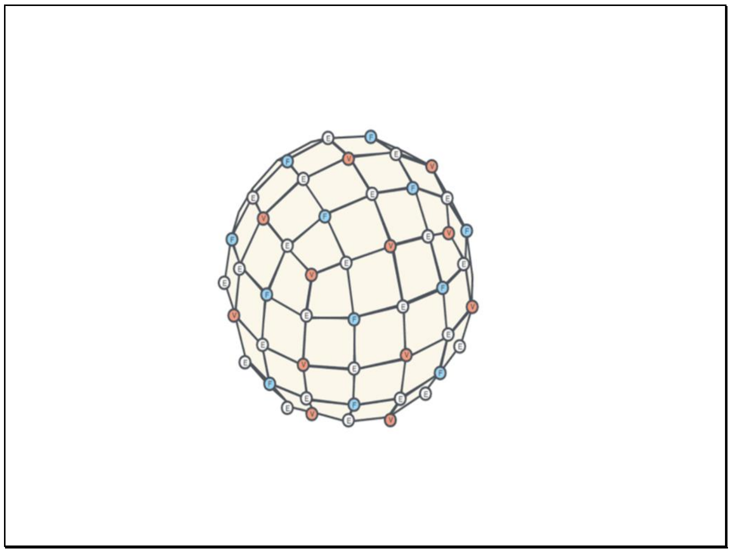
Once all the new vertices have been created, they are stitched together as quads, starting with the face points and walking around the connected edges. It's interesting to note that after the first global subdivision step, all faces become quads, regardless of the input, though these quads are not generally planar.

Slide 13



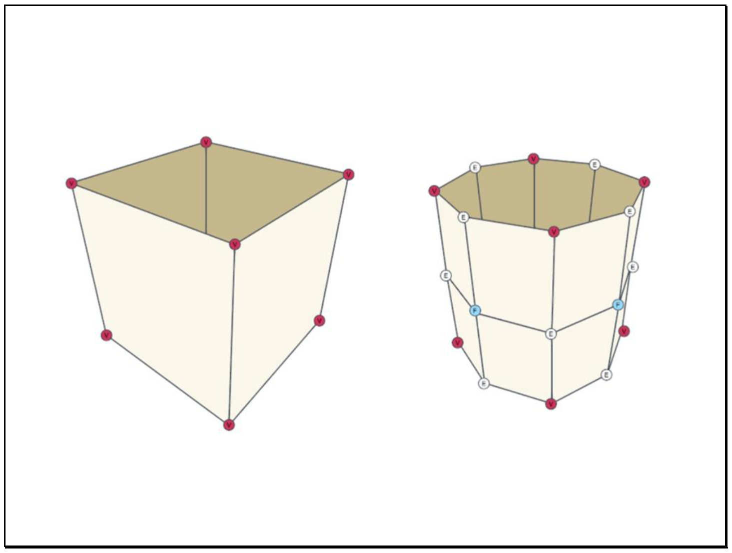
We can now treat the result as a new control mesh, and repeat the subdivision.

Slide 14



As we continue to subdivide, the resultant mesh gets closer to the subdivision surface, which we also call the limit surface.

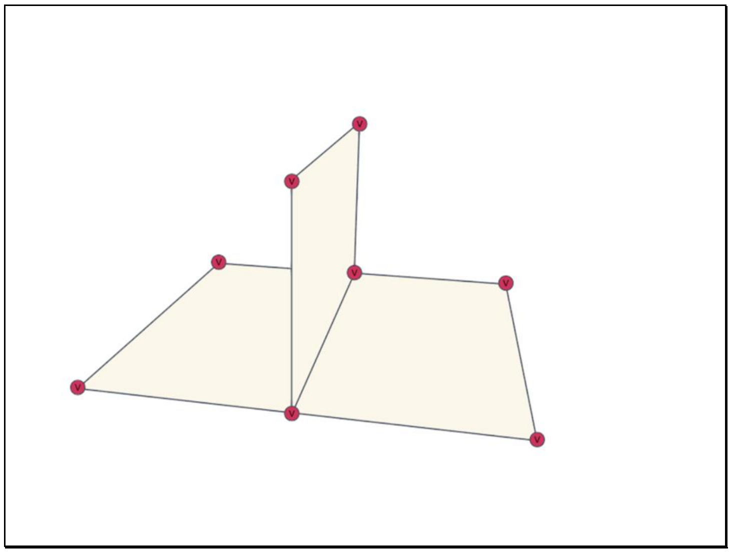
Slide 15



If the control mesh is not closed, rules are slightly different at border edges.

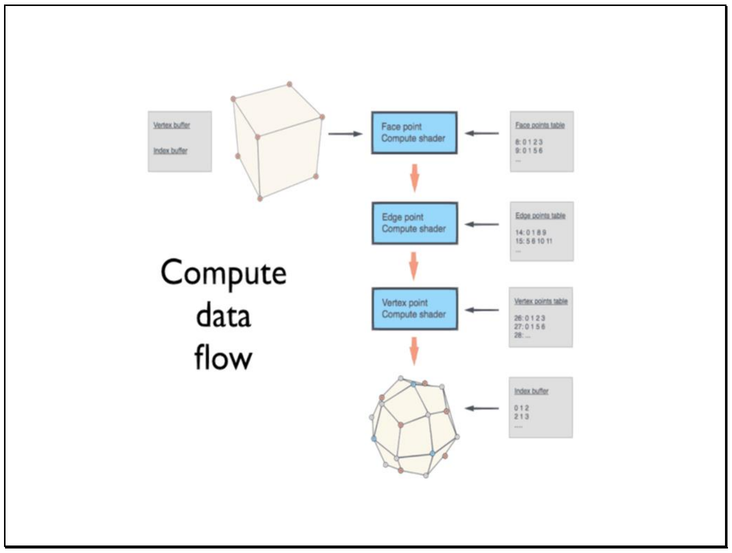
Border edge points are just the midpoint of the edge, and border vertex points are moved only towards their border edge endpoints, not towards inner edge endpoints or face points.

Slide 16



When topological circumstances get weird, like in non-manifold geometry, vertex points become corner vertex points and are not moved at all.

Slide 17



Compute data flow

It's fine to run global subdivision offline, and send the results through a standard model pipeline.

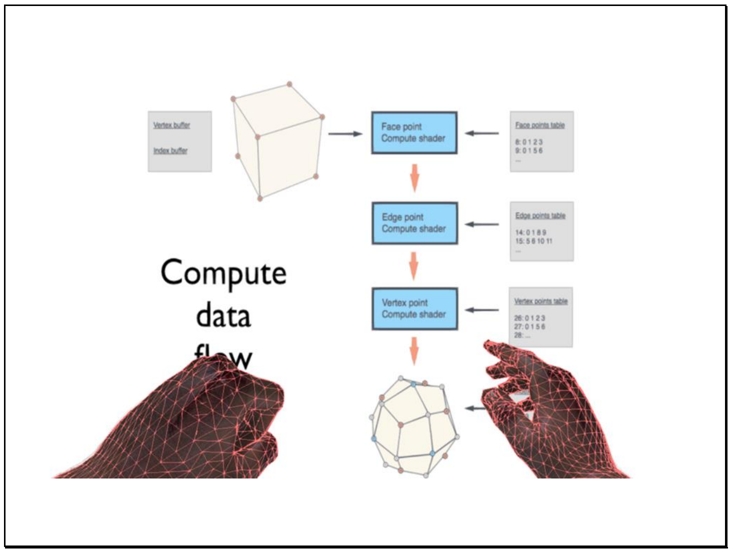
But global subdivision can also be implemented efficiently on modern GPUs in realtime, assuming the topology is fixed.

First, we analyze the control mesh and create a table of input vertex indices and weights, according to the Catmull Clark rules.

By executing a sequence of DirectX11 compute shader passes over the vertex buffer, we calculate the resultant face points, edge points and vertex points.

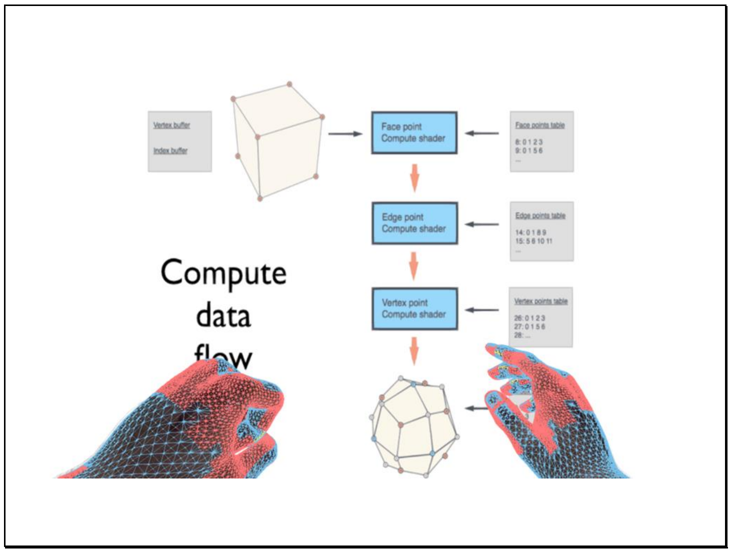
These points are then used to render the model, along with a prebuilt index buffer of triangles.

Slide 18



Why do this at runtime? One reason is that you can skin the control mesh, and then use compute shaders to evaluate the final mesh for rendering.

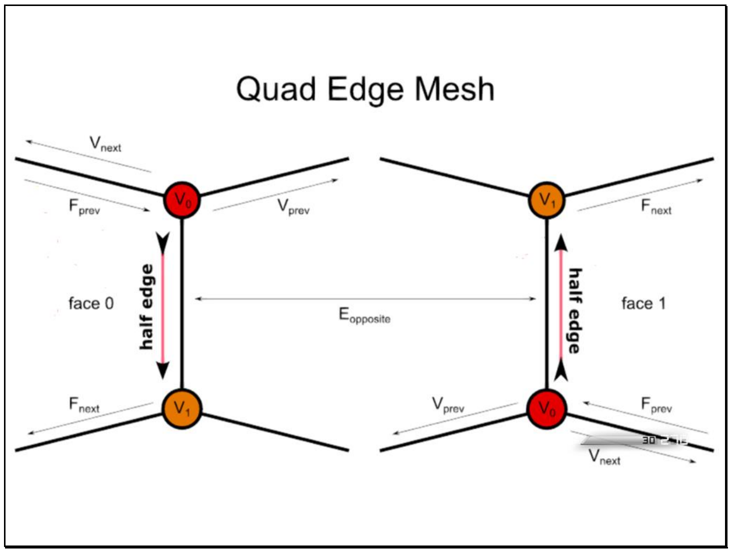
Slide 19



Compute data flow

The control mesh is much cheaper to skin, and this possibly enables more expensive techniques like tension mapping, extra bones and dual quaternion blends.

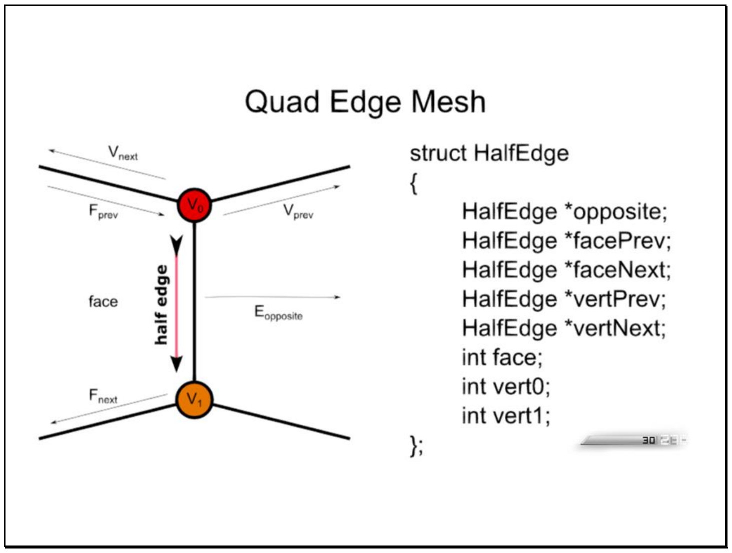
Slide 20



To build the tables, we must first construct a data structure which represents the topology of the mesh. A good one to start with is the quad edge mesh, which uses directed half-edges to represent all elements of the mesh, including faces, vertices and edges. This diagram shows how a half edge is connected to its various neighboring components.

On the left we have a half edge running from v0 to v1. It stores its vertex indices, and the face it's connected to. It also stores connections to other half edges. We have the next half edge on the same face, winding clockwise, and the previous half edge on the same face, winding counterclockise. We also have the next half edge departing from the origin vertex, winding clockwise, and the previous half edge departing from the origin vertex, winding counter clockwise. Finally, we store a reference to the corresponding edge going in the opposite direction, if this is not a border edge.

Slide 21



Quad Edge Mesh

```
struct HalfEdge
{
        HalfEdge *opposite;
        HalfEdge *facePrev;
        HalfEdge *faceNext;
        HalfEdge *vertPrev;
        HalfEdge *vertNext;
        int face;
        int vert0;
        int vert1;
};
```

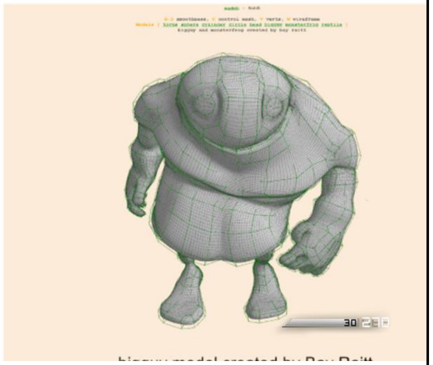
V_{next}
F_{prev}
V_0
V_{prev}
face
half edge
$E_{opposite}$
F_{next}
V_1

This slide backs up the diagram with an example of how the half edge might be defined in C.

Slide 22

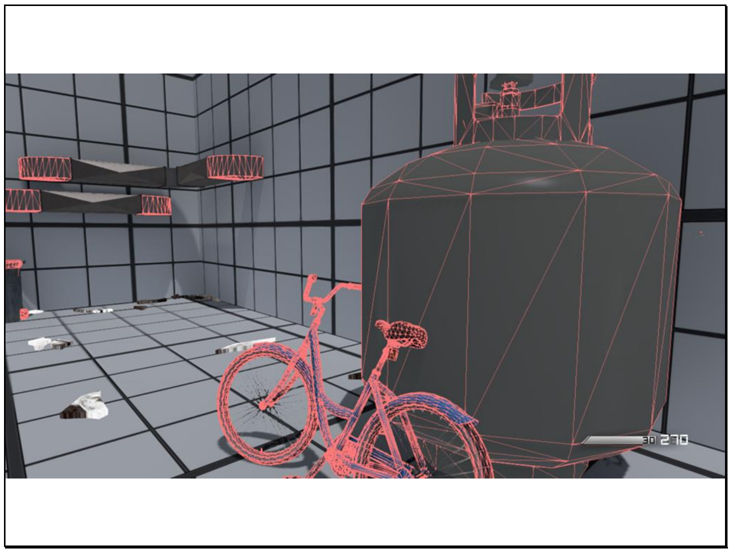
Quad Edge Mesh

Sample implementation
in JavaScript + WebGL
http://wadeb.com/subd



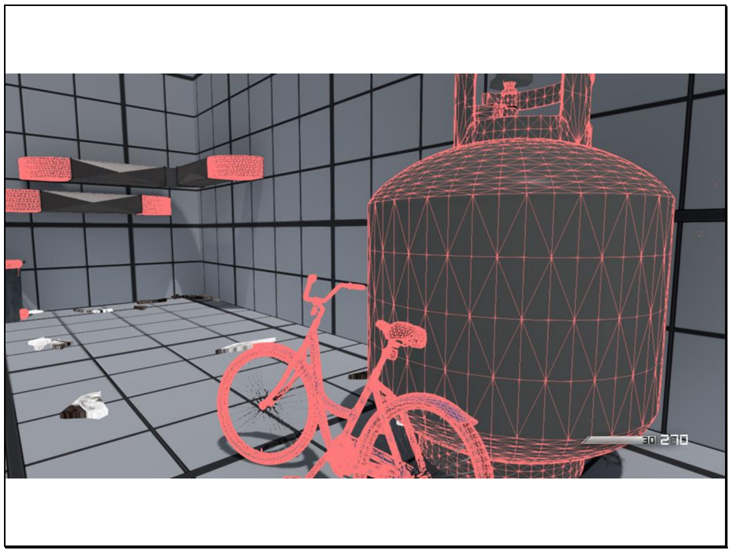
I've created an example implementation of the Quad Edge Mesh and global subdivision in JavaScript / WebGL, which you're welcome to use as a starting point.

Slide 23

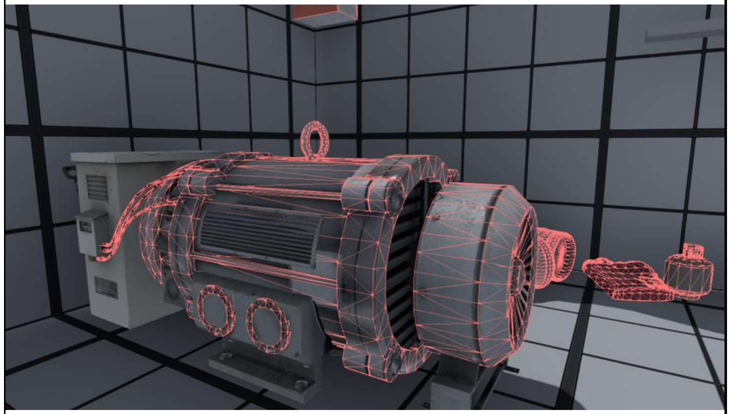


This room contains some examples of game models rendered using compute shader global subdivision.

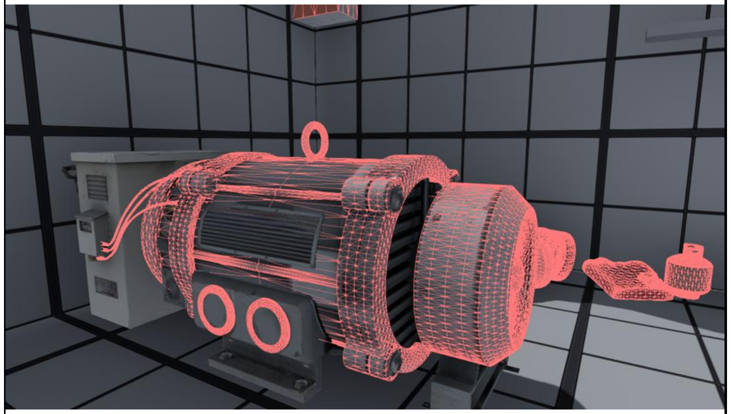
Slide 24



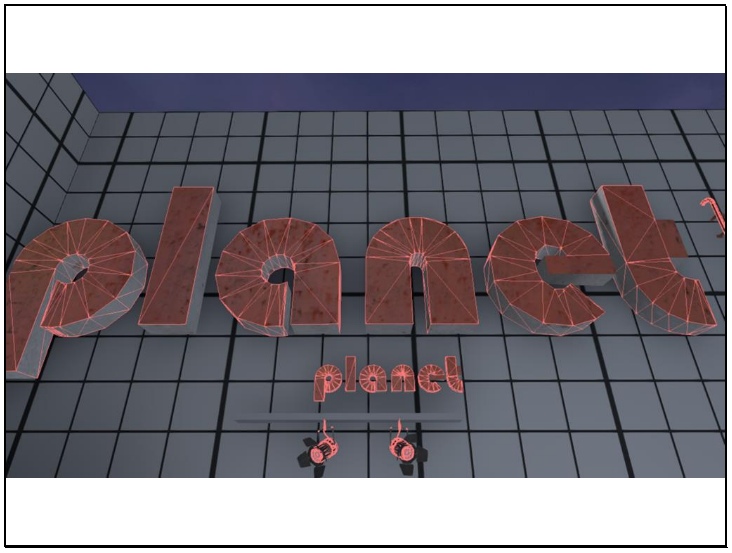
Slide 25



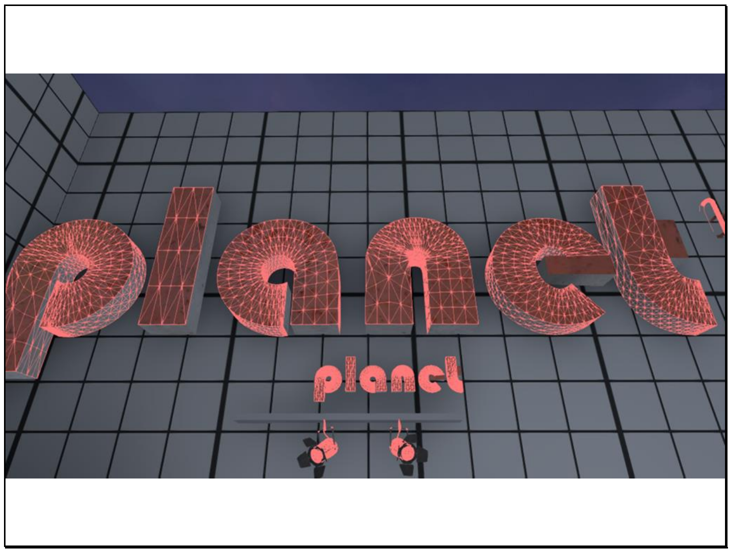
Slide 26



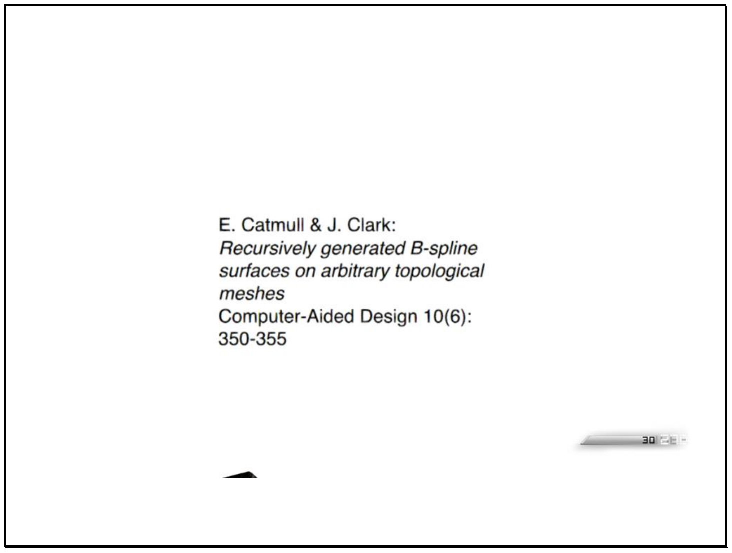
Slide 27



Slide 28



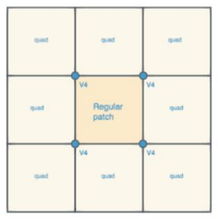
Slide 29


E. Catmull & J. Clark:
Recursively generated B-spline surfaces on arbitrary topological meshes
Computer-Aided Design 10(6): 350-355

This presentation is supposed to be about tessellation though, so let's look at a different way to evaluate SubDs.

When a face in the control mesh is a quad, surrounded by eight other quads in the topology, its limit surface can be evaluated directly as a B-spline surface, without resorting to global subdivision. The faces surrounding a face in the control mesh topology are called its "one ring" faces. To qualify for B-spline evaluation, note that all four vertices of the face must connect to exactly four edges.

Slide 30

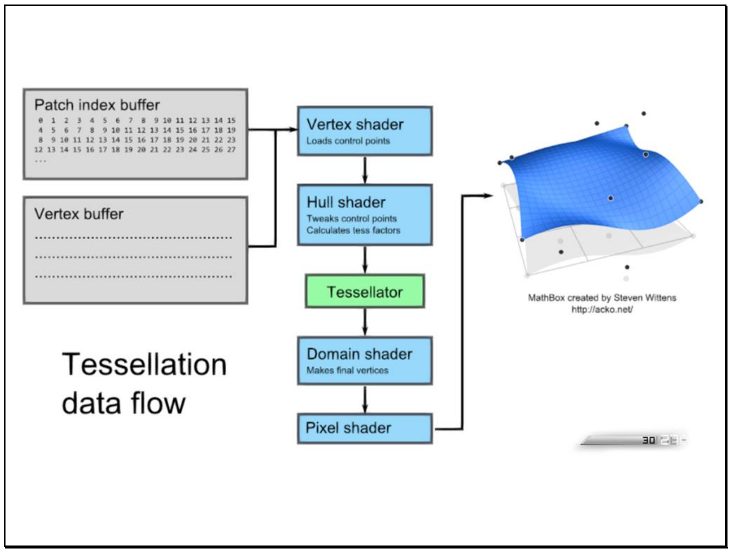


30

When a face in the control mesh is a quad, surrounded by eight other quads in the topology, its limit surface can be evaluated directly as a B-spline surface, without resorting to global subdivision.

The faces surrounding a face in the control mesh topology are called its "one ring" faces. To qualify for B-spline evaluation, note that all four vertices of the face must connect to exactly four edges.

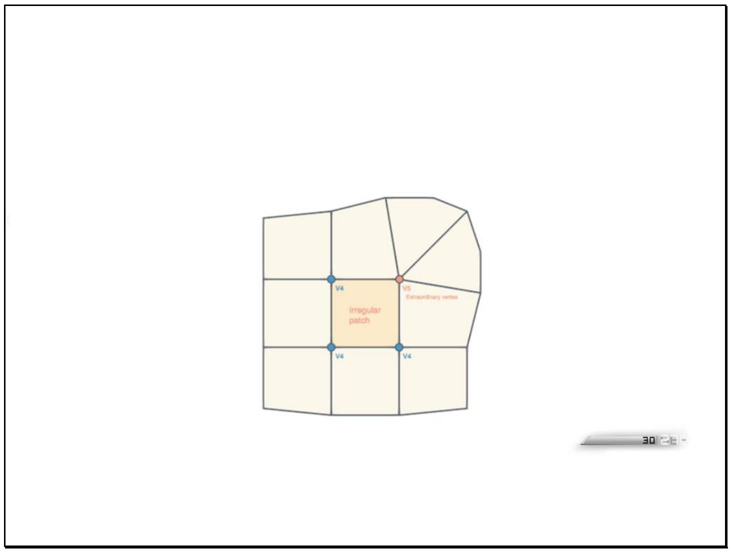
Slide 31



By treating the vertices of the face and its one ring neighbors as a 4x4 grid, we have the coefficients of a bicubic function that evaluates to the limit surface of the face.

These 16 points are stored in an index buffer, the control mesh vertices are stored in a vertex buffer, and together they are passed through the DX11 tessellation pipeline.

Slide 32



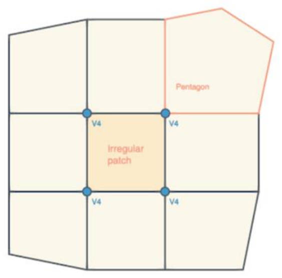
Faces which meet the criteria for B-spline surfaces are called "regular faces" or "regular patches".

Faces which do not meet this criteria are called "irregular faces" or "irregular patches".

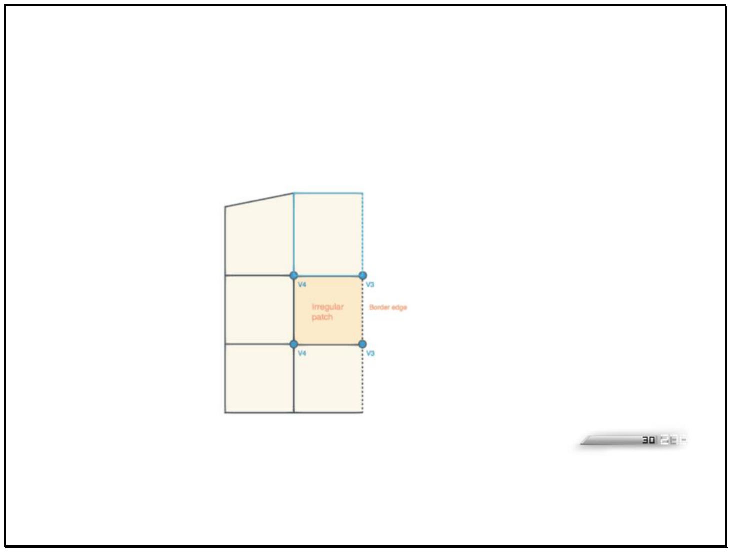
Vertices which are connected to other than four edges are called "extraordinary vertices".

Some examples of faces which do not meet the regular criteria include faces connected to extraordinary vertices, faces whose one ring faces are not quads, and border faces.

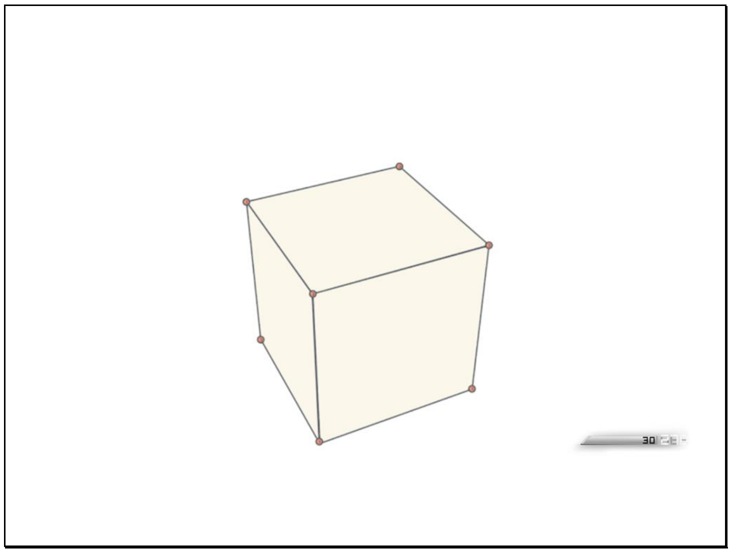
Slide 33



Slide 34

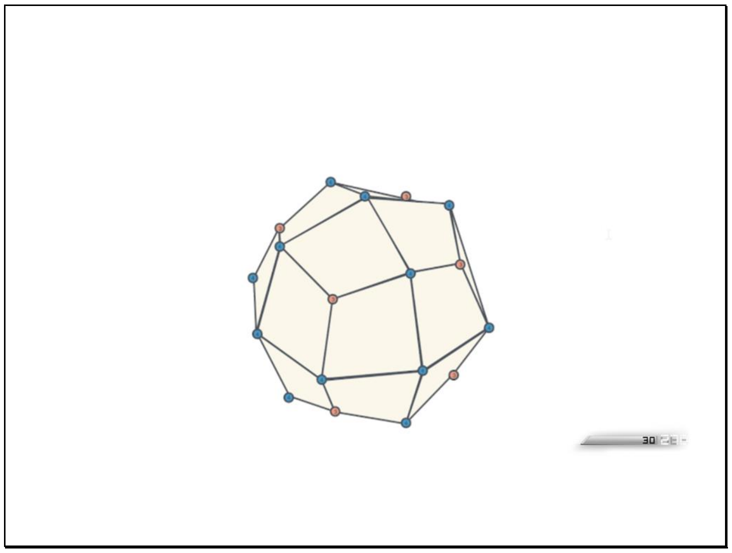


Slide 35



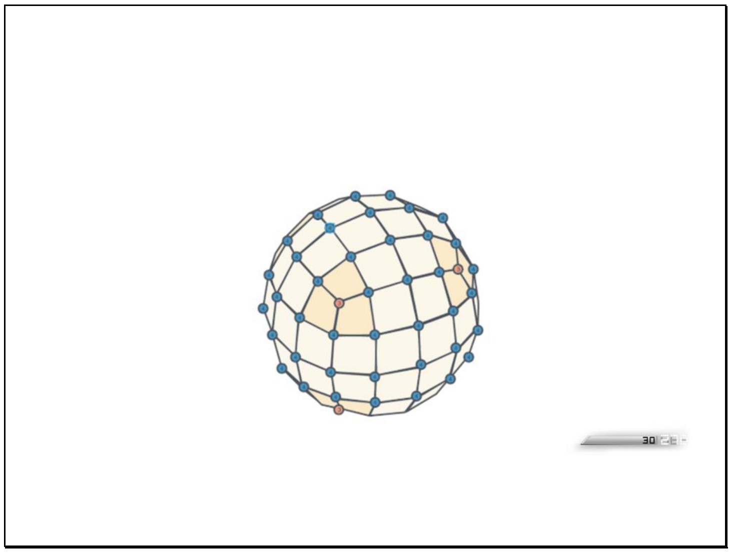
Let's look at how global subdivision affects regular patch extraction.

Slide 36



With each level of global subdivision, the result is a new SubD control mesh, on its way to the limit surface.

Slide 37

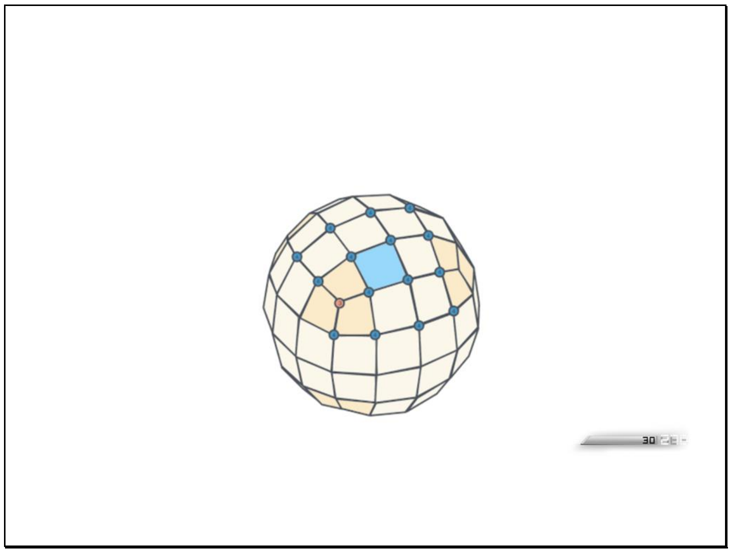


As we subdivide, more faces will meet the criteria and become regular.

The light patches here are regular, while the dark shaded patches are still connected to an extraordinary vertex.

With a cube, for example, the second subdivision introduces regular faces, and you can start to see the isolation of the extraordinary vertices.

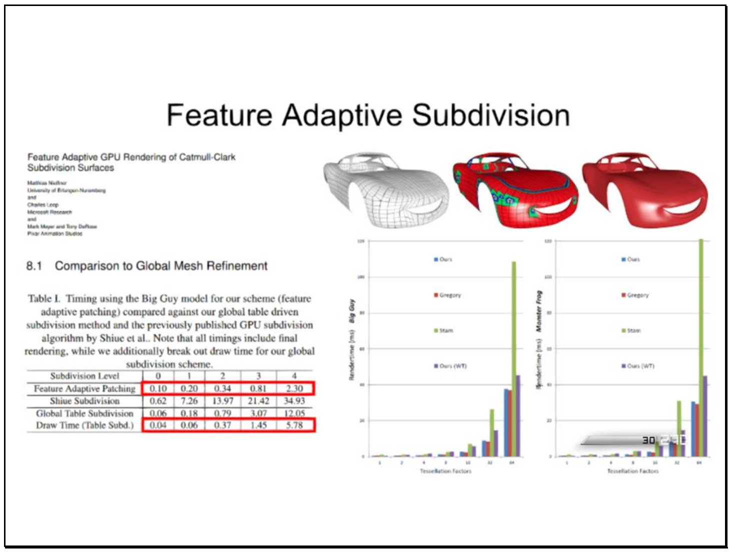
Slide 38



I've highlighted regular one face and its corresponding control points so you can see exactly what runs through the bspline function.

Eventually, only tiny clusters of irregular faces are left around borders and extraordinary vertices. Unfortunately, no amount of global subdivision will eliminate the irregular faces entirely.

Slide 39



Feature Adaptive Subdivision

Feature Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces

Matthias Nießner
University of Erlangen-Nuremberg
and
Charles Loop
Microsoft Research
and
Mark Meyer and Tony DeRose
Pixar Animation Studios

8.1 Comparison to Global Mesh Refinement

Table I. Timing using the Big Guy model for our scheme (feature adaptive patching) compared against our global table driven subdivision method and the previously published GPU subdivision algorithm by Shiue et al.. Note that all timings include final rendering, while we additionally break out draw time for our global subdivision scheme.

Subdivision Level	0	1	2	3	4
Feature Adaptive Patching	0.10	0.20	0.34	0.81	2.30
Shiue Subdivision	0.62	7.26	13.97	21.42	34.93
Global Table Subdivision	0.06	0.18	0.79	3.07	12.05
Draw Time (Table Subd.)	0.04	0.06	0.37	1.45	5.78

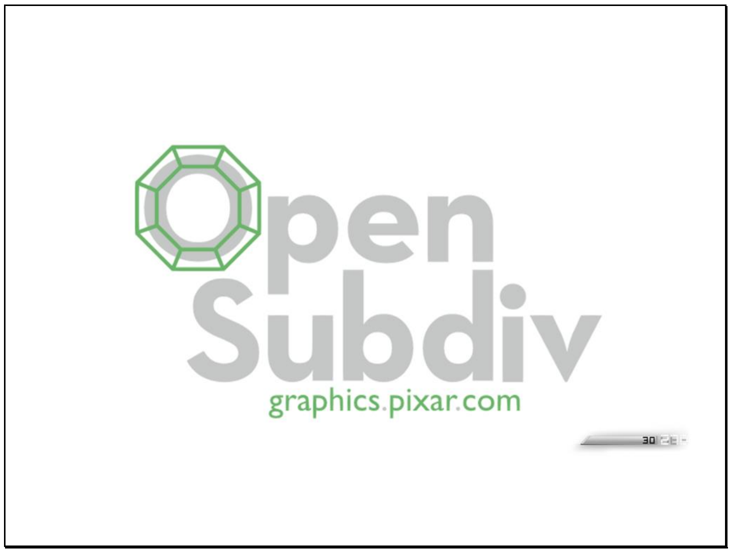
This is where feature adaptive subdivision comes into play.

Invented by Matthias Niessner, Charles Loop, Mark Meyer, and Tony Derose, Feature Adaptive subdivision divides the faces into regular and irregular groups, evaluating regular faces with the tessellation hardware and irregular faces with global subdivision.

At each global subdivision level, all regular faces are extracted, leaving only the irregular ones to subdivide globally.

Neissner's paper shows significant memory and performance benefits for feature adaptive subdivision compared with global subdivision, and as the algorithm is a superset of global subdivision, we had a good fallback option in case tessellation failed to work out. So, feature adaptive subdivision is what we decided to implement.

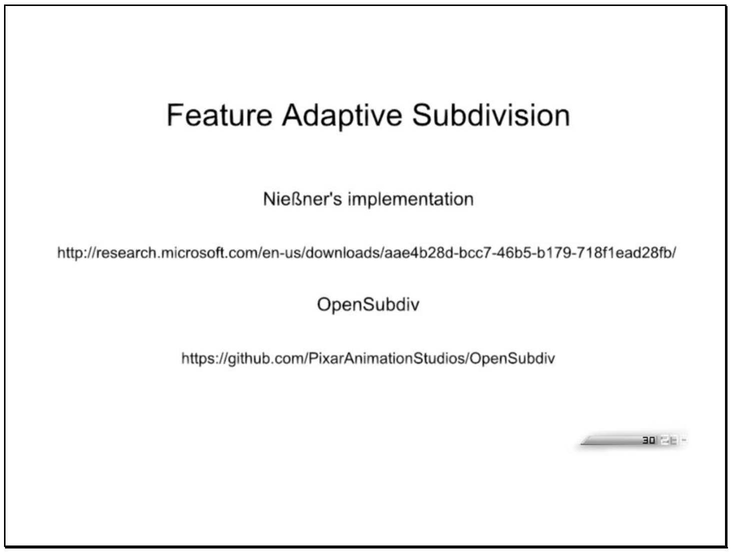
Slide 40



About halfway through our implementation of FAS, Pixar released OpenSubdiv, which is an open source library that implements feature adaptive subdivision, among other algorithms.

This would likely be a good starting point for anyone who is looking to do their own implementation.

Slide 41



Feature Adaptive Subdivision

Nießner's implementation

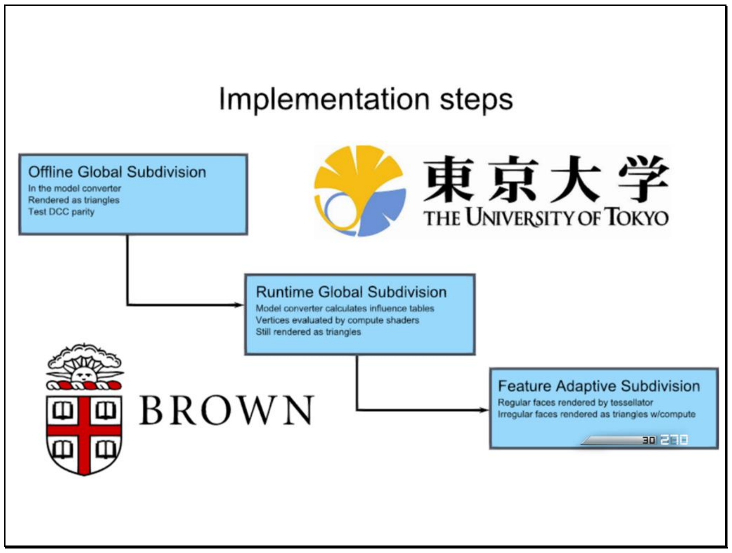
http://research.microsoft.com/en-us/downloads/aae4b28d-bcc7-46b5-b179-718f1ead28fb/

OpenSubdiv

https://github.com/PixarAnimationStudios/OpenSubdiv

Neissner's own source code is available from Microsoft Research, and is also a good starting point.

Slide 42



Implementation steps

Offline Global Subdivision
In the model converter
Rendered as triangles
Test DCC parity

Runtime Global Subdivision
Model converter calculates influence tables
Vertices evaluated by compute shaders
Still rendered as triangles

Feature Adaptive Subdivision
Regular faces rendered by tessellator
Irregular faces rendered as triangles w/compute

We started by implementing global subdivision only in our model pipeline, so the artists were able to try modeling SubDs quickly and give feedback.

The model exporter was upgraded to export N-sided faces, and the model converter ran global subdivision and produced a triangle mesh.

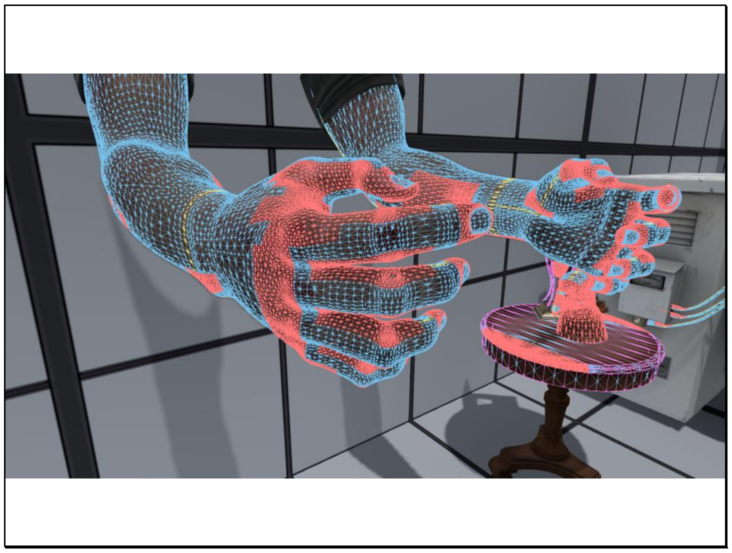
The next stage in our implementation was to move the subdivision out of the model converter and into the runtime, using DirectX 11 compute shaders to perform the global subdivision, driven by tables emitted by the model converter.

The first version of table generation was written by one of our summer interns, Maceij Kot, from the Tokyo Institute of Technoloy.

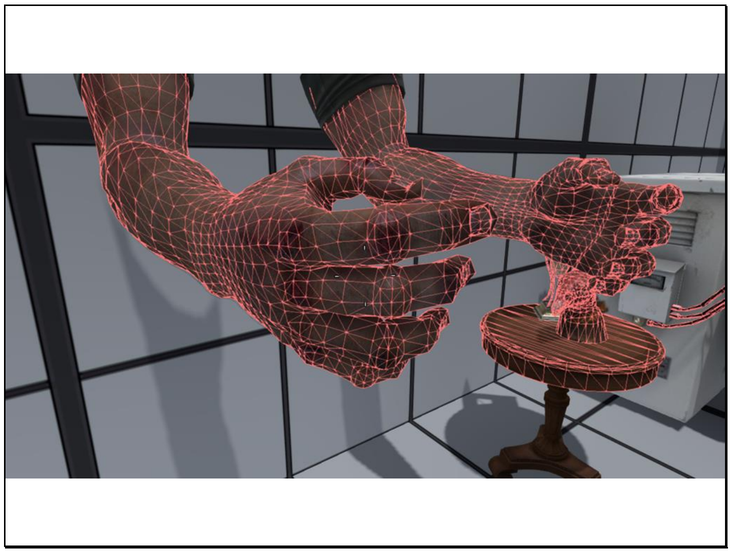
Once runtime global subdivision was working well, we moved on to extracting regular patches from our control mesh topology and using hardware tessellation to render them. The initial tessellation work was done by our other intern, Kefei Lei, from Brown University.

The models in this scene are rendered using feature adaptive subdivision. Blue, yellow and purple wireframe represent regular faces, thus tessellation, while red continues to represent irregular faces, thus global subdivision.

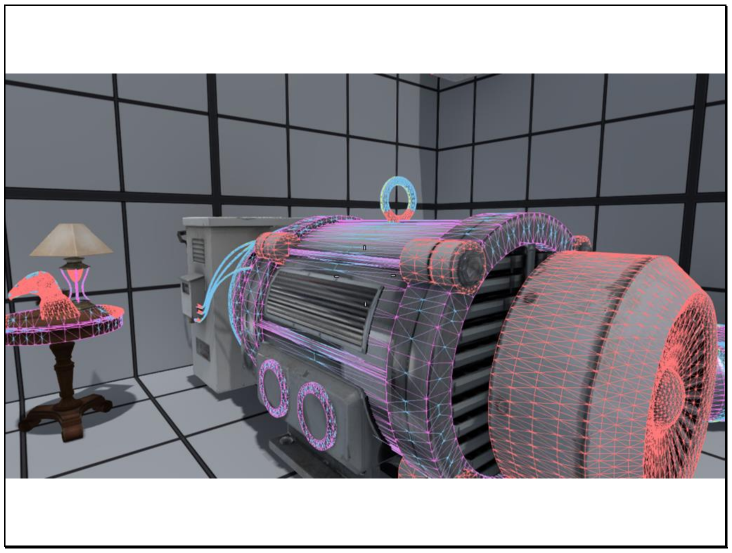
Slide 44



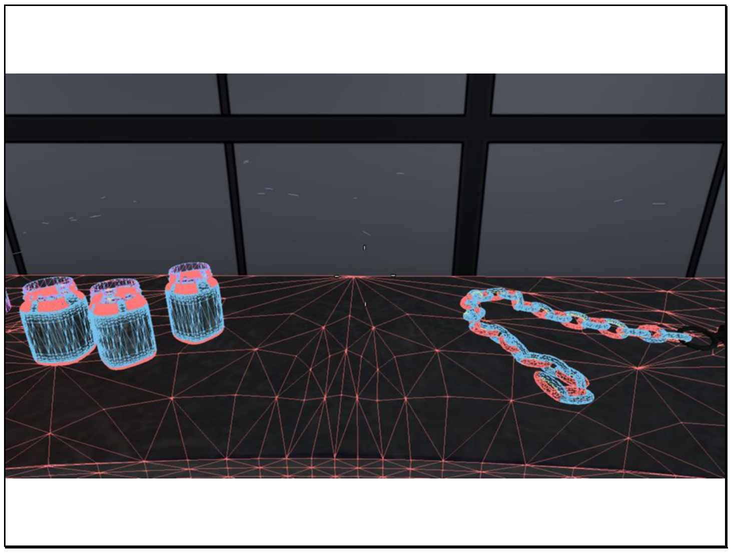
Slide 45



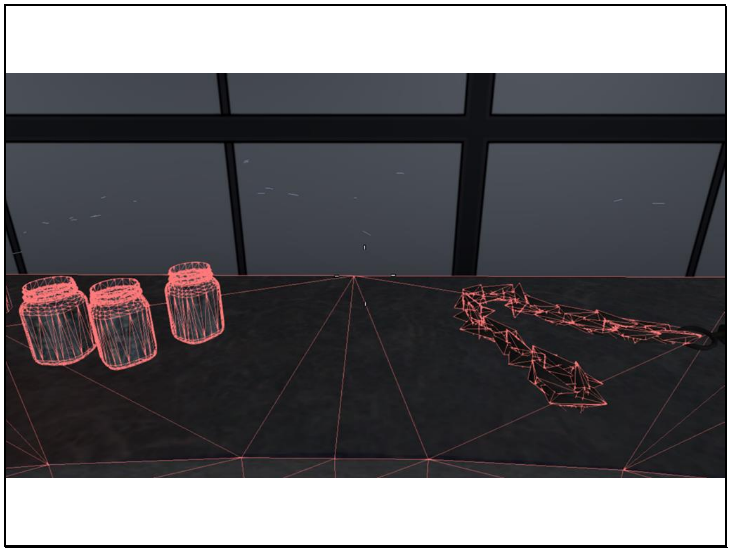
Slide 46



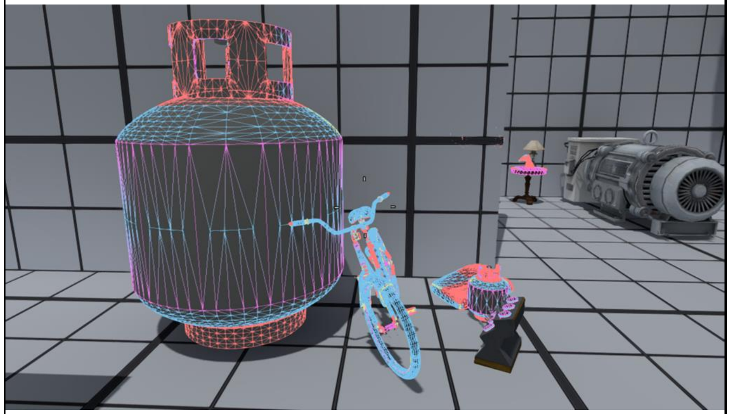
Slide 47



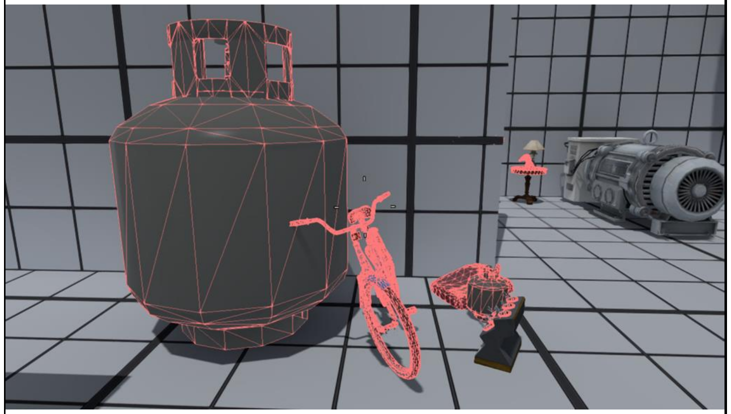
Slide 48



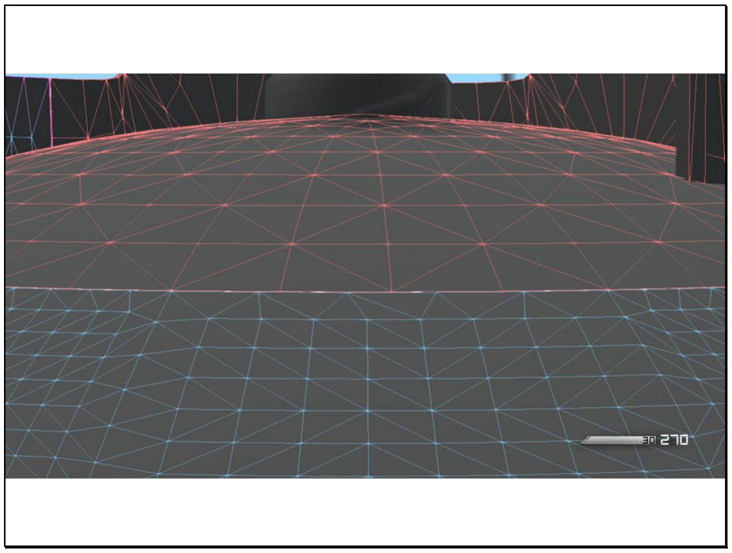
Slide 49



Slide 50



Slide 51

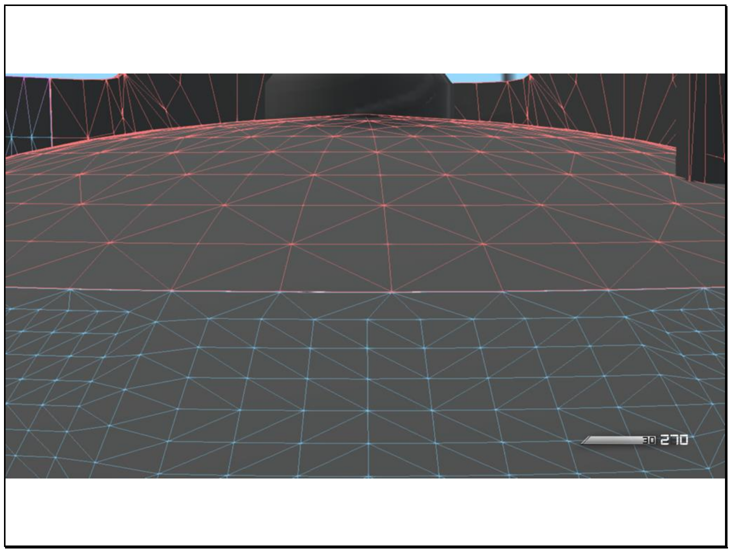


Now we'll look at some of the details. First up is cracks.

The first source of cracks is tessellation factors differing between the regular and irregular faces.

Global subdivision can only divide faces evenly, but the tessellator should divide things as much as it wants to. With a different number of divisions along each side of the edge, cracks appear.

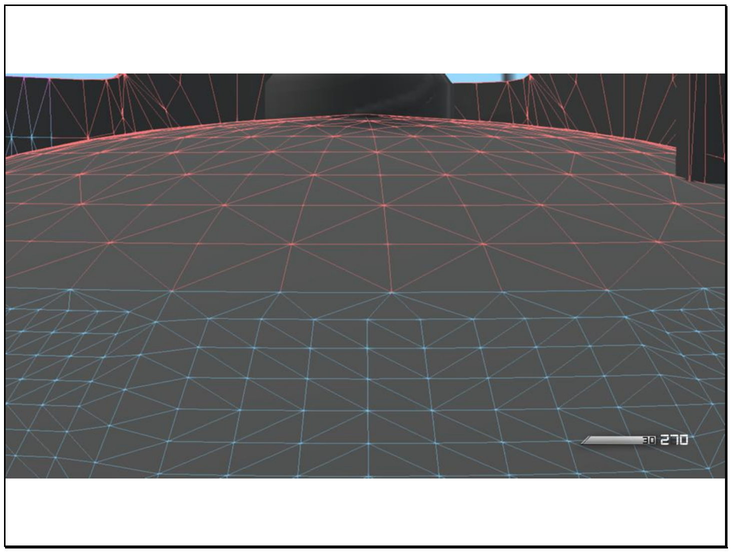
Slide 52



To solve this, we store flags alongside each regular face, with a bit for each edge to indicate whether that edge connects to an irregular face. If the bit is set, we lock the tessellation factor for that edge to the global subdivision level for the model.

This resolves one source of cracks, but as you can see the mesh is not closed.

Slide 53

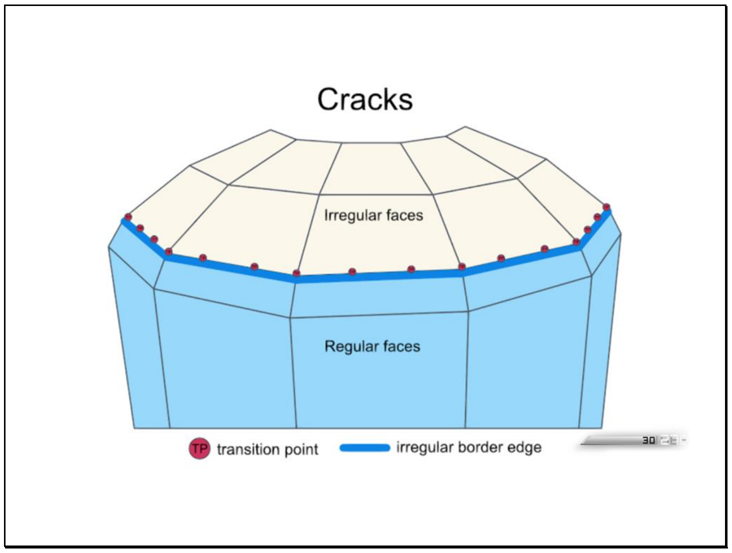


The next source of cracks is the limit surface itself.

Recall from the description of global subdivision, that as you repeat the process, the mesh vertices approach the limit surface. The B-spline function used by the tessellator outputs vertices that are on the limit surface.

To resolve this discrepancy, we identify the global subdivision vertices that are coincident with tessellated vertices. We call these transition points, and run the B-spline evaluation code in the compute shader, using the corresponding regular patch and domain location as inputs. We write the result to the vertex buffer, instead of the global subdivision result.

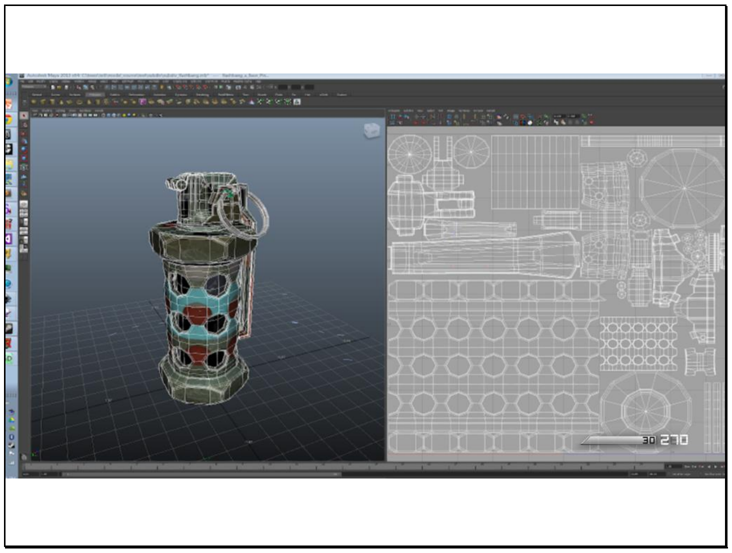
Slide 54



Cracks

This diagram shows the border between regular and irregular faces, and the transition points and edge point flags.

Slide 55



The next detail I want to talk about is texture coordinate smoothing.

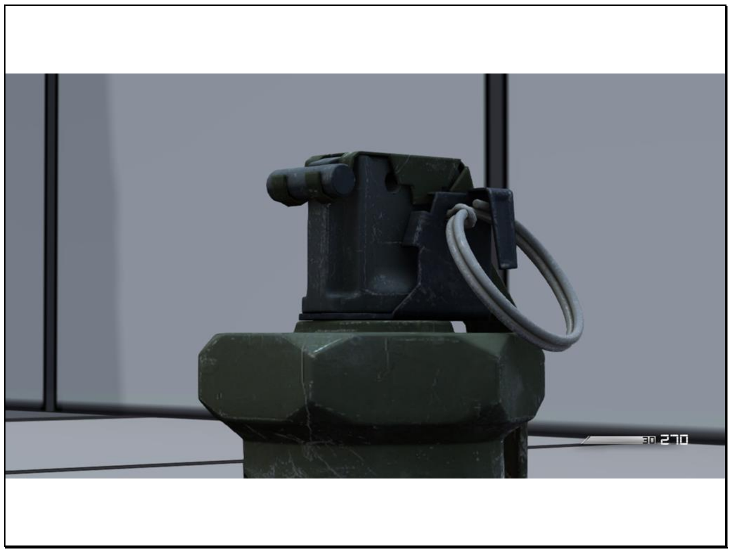
When SubD smooths the surface of the control mesh, it pulls the existing vertices towards their neighbors. Depending on the shape and size of the polygons, this can move vertices quite far. If vertex components like texture coordinates are linearly interpolated, they suffer from bad stretching.

The solution is to apply the SubD algorithm to the texture coordinates also, which, assuming they were mapped relatively uniformly to begin with, applies something like an inverse of the stretching the vertex positions undergo.

The Catmull Clark rules depend on topology, and UV topology can be different from vertex topology. While vertex borders can always be considered UV borders, UV borders often exist where vertices are smooth.

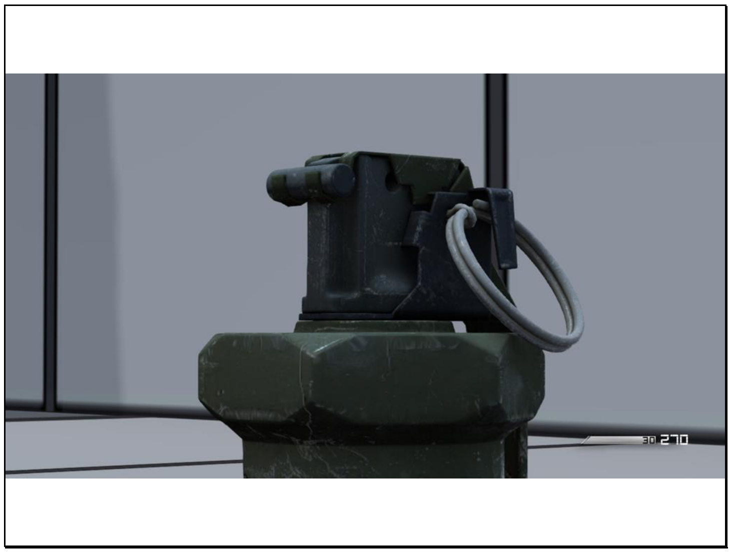
We encode the UV topology as a subset of the vertex topology. At UV borders, vertices are duplicated, as with a triangle mesh. The SubD tables reflect which influences are for vertices only, and which influences are also for UVs.

Slide 56



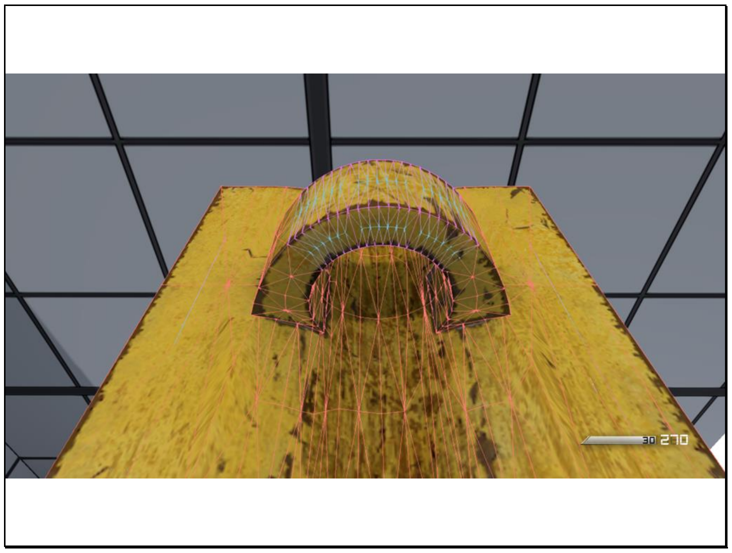
This scene demonstrates the effect of UV smoothing on a model.

This image is rendered with UV smoothing.



This image is rendered without UV smoothing. Observe the stretching near the beveled edges.

Slide 58



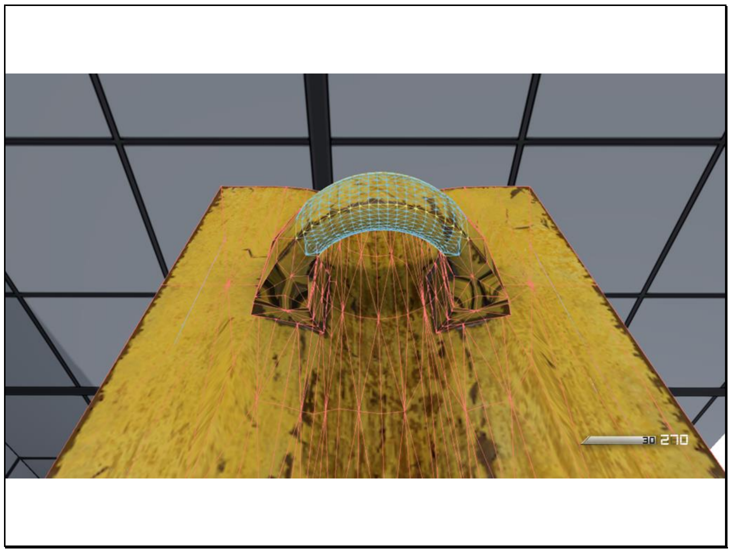
The next detail I'd like to talk about is vertex splitting.

In Maya, artists can arbitrarily split and merge coincident vertices, and this affects the topology that SubD uses to smooth the meshes.

Artists can take advantage of this in SubD models to create sharp creases, by intentionally splitting the vertices along the crease. Similarly, UVs can be split or merged to affect the UV smoothing topology.

The top ring of this pulley was modeled using intentionally split vertices to create a sharp crease. Our artists call this the split edge technique.

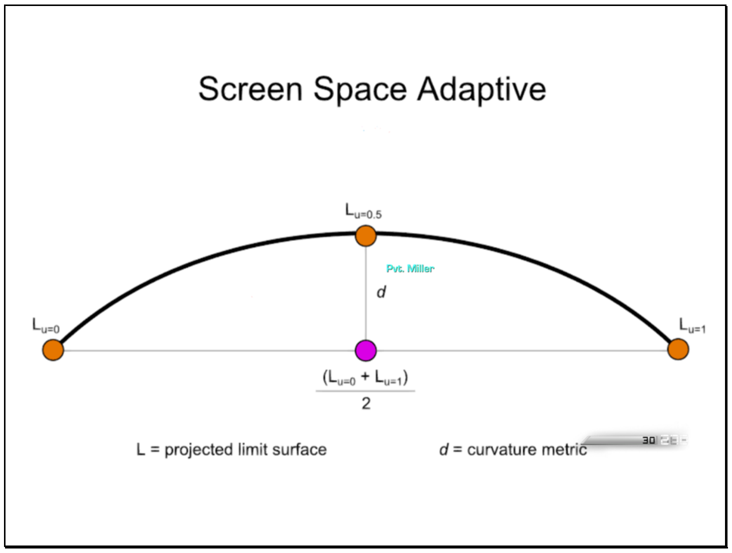
Slide 59



If the vertices were not split, the ring would look like this.

If the game engine is going to match Maya, vertices and UVs cannot be automatically reindexed by the content pipeline until after the topology is analyzed.

Slide 60



Screen Space Adaptive

One of the principal advantages of using DX11 tessellation is the ability to adaptively change tessellation as required.

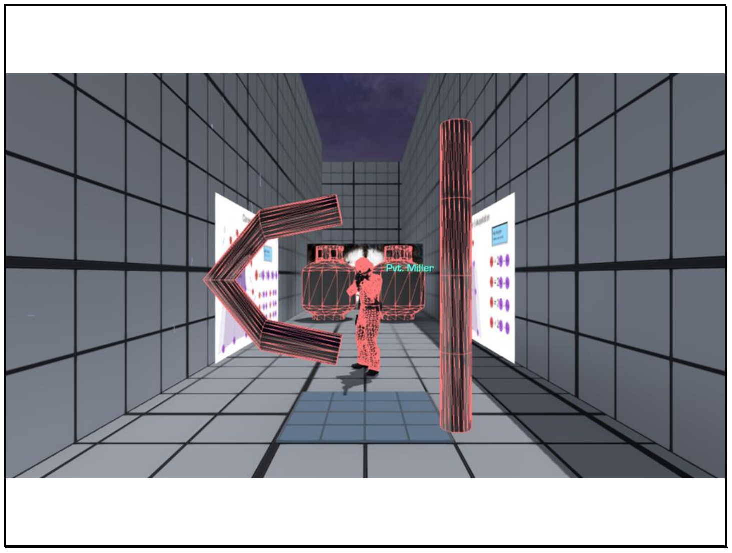
The goal of adaptive tessellation is to emit the right number of vertices for each edge, such that the curvature of the model is well represented.

Our approach is to evaluate the limit surface and project to screen space at discrete points around each patch. These points include each corner, and midpoint of each edge, so 8 points total.

If the edge were not subdivided, the output geometry would be flat and the midpoint would lie on the line between the two limit surface corners.

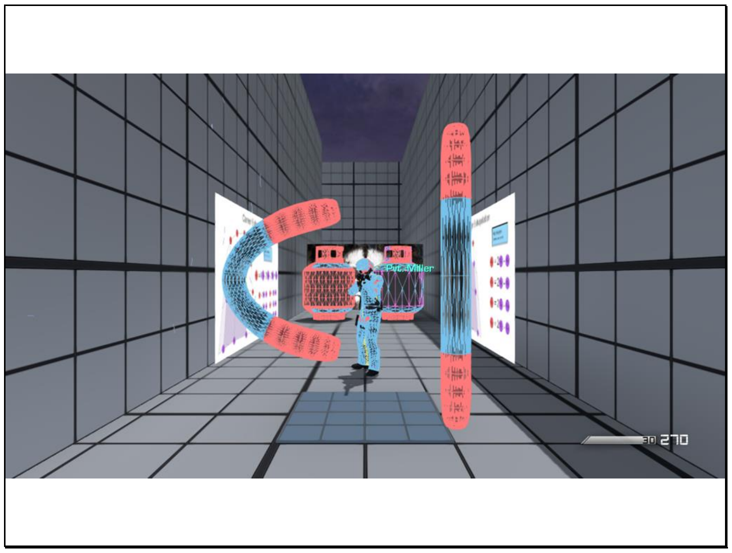
If the edge was subdivided once, the midpoint of the output geometry would lie exactly on the limit surface. We use distance between these two points as our estimate of patch curvature, and derive the tessellation factor from it.

Slide 61



This scene shows two topologically identical pipe meshes. One is straight while the other is bent.

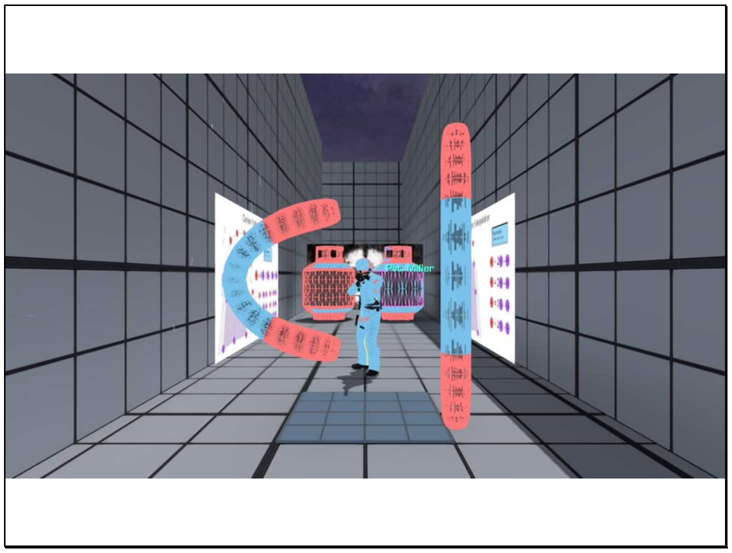
Slide 62



With our adaptive metric, the straight pipe is not subdivided at all along its straight axis.

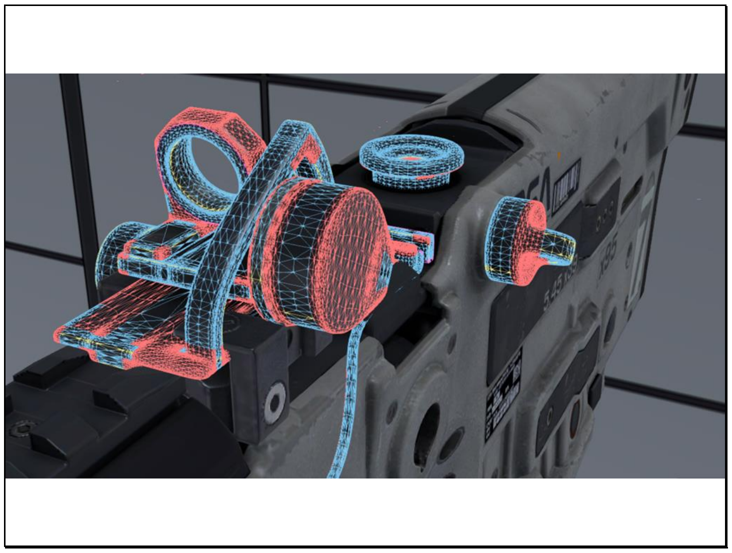
The curved pipe is subdivided an appropriate number of times.

Slide 63



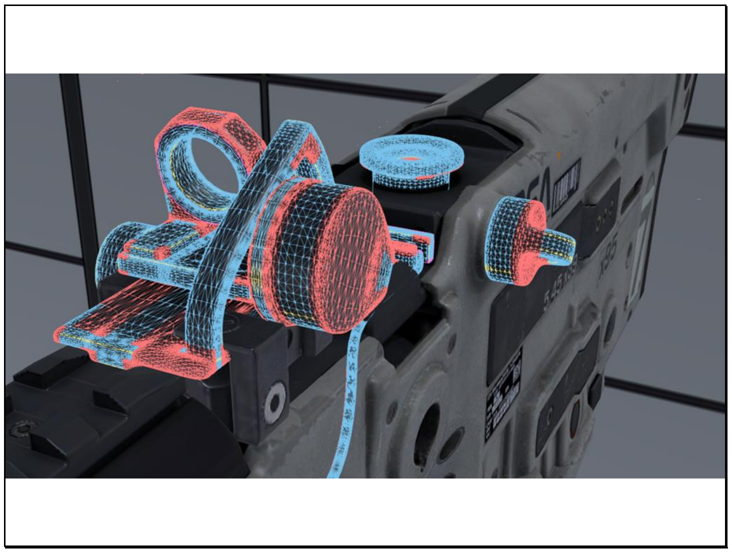
Without our adaptive metric, the straight edge is subdivided too much, and the curved edge is not subdivided enough.

Slide 64



These images show the difference in polygon density with a typical model. Observe the gun sight area where the flat part is not subdivided at all.

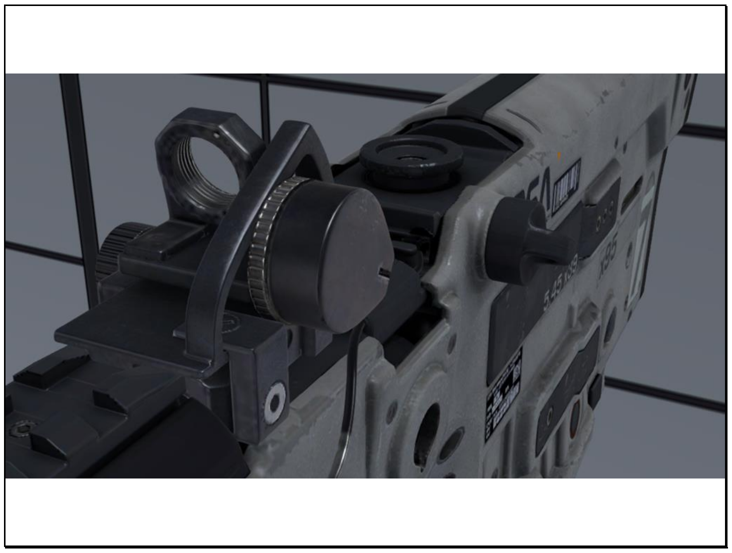
Slide 65



This image shows the same model without adaptive subdivision.

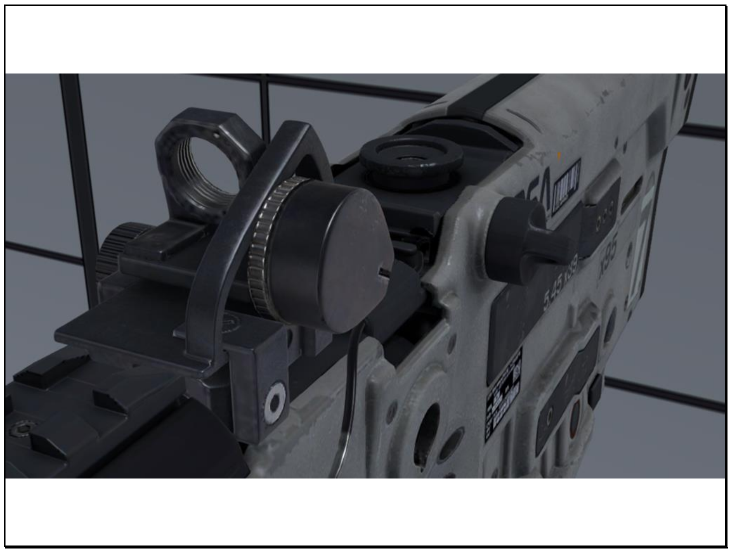
Over-tessellation hurts pixel shader quad efficiency and thus performance.

Slide 66



The next two images show the visual result with and without adaptive subdivision.

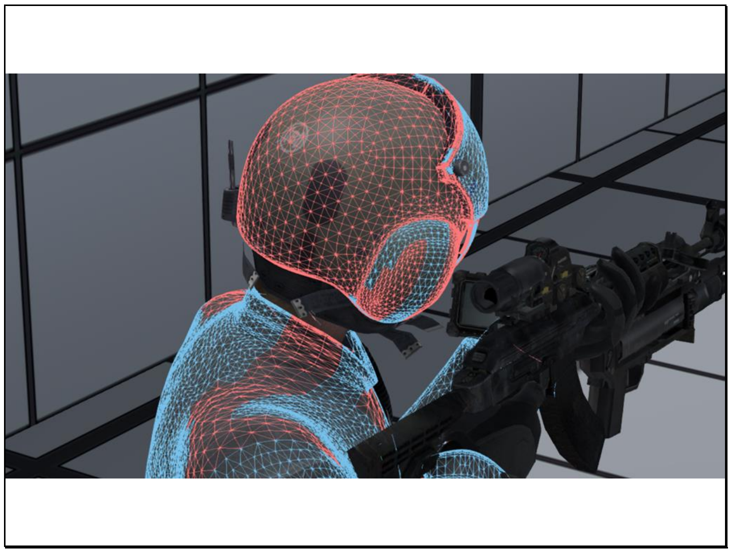
The first image has adaptive subdivision enabled.



The second image has adaptive subdivision disabled.

Our adaptive metric works well enough that we can use integer tessellation factors, which generate fewer triangles than fractional tessellation factors and were a significant performance win.

Slide 68



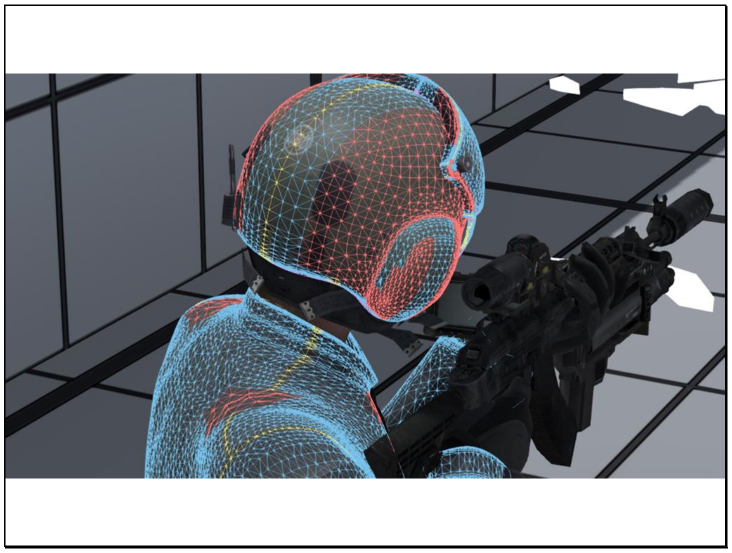
Our tessellated patches are fast and look great close up, but what about the global parts? They use more memory for the tables and are only capable of dividing to power of 2 divisions. In well built geometry much of the model will be regular, but this isn't something artists are used to dealing with.

Some danger areas:

When artists are modeling with split edges, borders will prevent patches from being regular, because border faces do not have eight neighbors.

Intersections between different parts of the model occasionally involve pentagons, or extraordinary vertices.

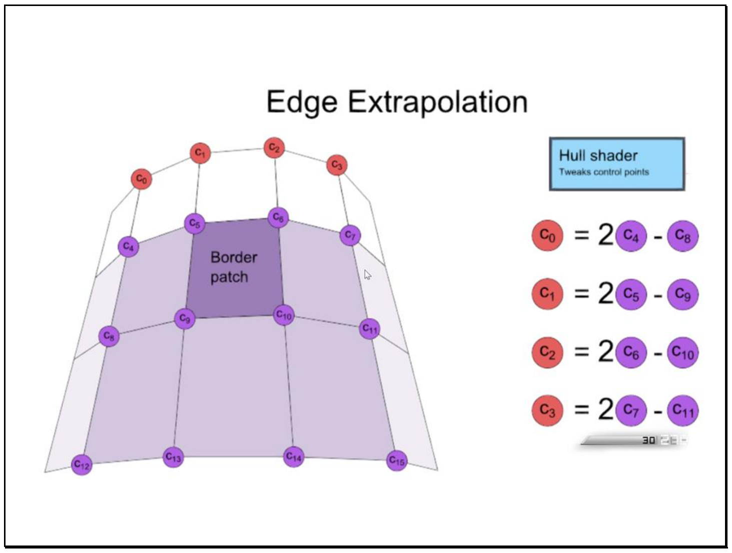
When texture mapping comes into play, UV seams are effectively border edges, and cut swaths of irregularity into the geometry. Observe this behavior on the pilot’s shoulder, where two UV seams meet.



We alleviated some of these issues by elevating certain topological special cases into custom regular faces.

In this image, yellow edges represent UV border edges that remained regular.

Slide 70



Edge Extrapolation

Hull shader
Tweaks control points

$c_0 = 2c_4 - c_8$

$c_1 = 2c_5 - c_9$

$c_2 = 2c_6 - c_{10}$

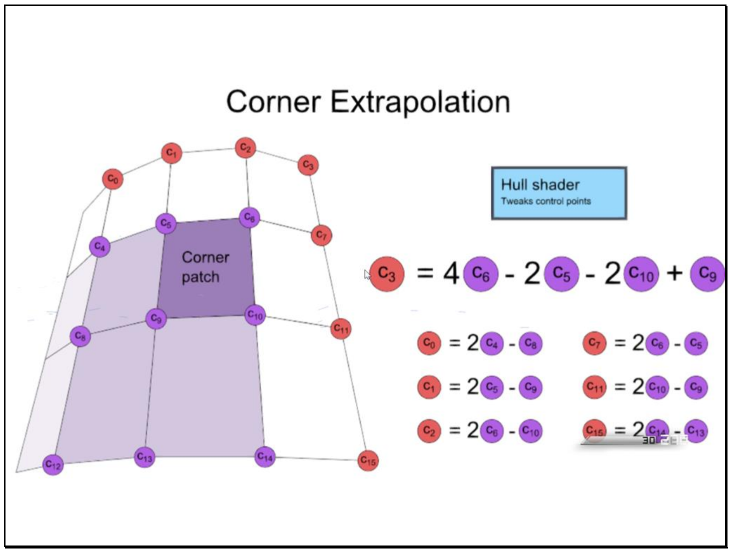
$c_3 = 2c_7 - c_{11}$

Border patch
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15}

To elevate a border edge to a regular patch, we must extrapolate the missing control points. This is a simple linear extrapolation from the parallel rows of control points.

The missing control points are extrapolated in the hull shader stage, based on per-patch flags.

Slide 71



Corner Extrapolation

Hull shader
Tweaks control points

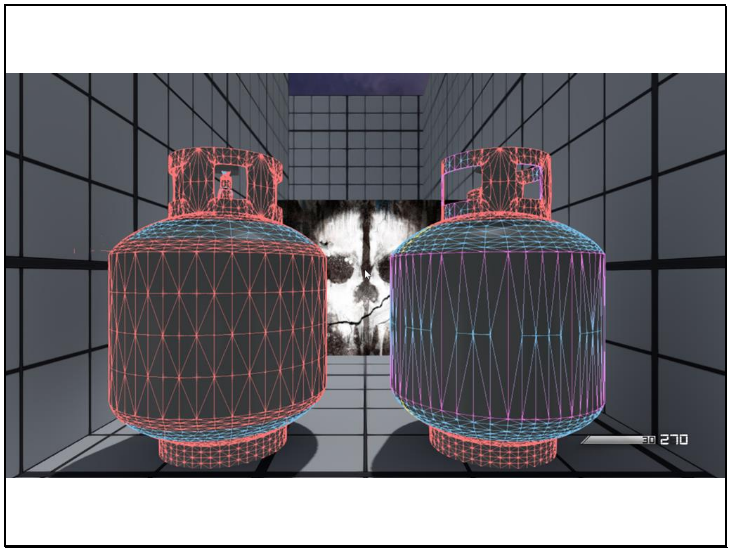
$$c_3 = 4c_6 - 2c_5 - 2c_{10} + c_9$$

$c_0 = 2c_4 - c_8$	$c_7 = 2c_6 - c_5$
$c_1 = 2c_5 - c_9$	$c_{11} = 2c_{10} - c_9$
$c_2 = 2c_6 - c_{10}$	$c_{15} = 2c_{14} - c_{13}$

To elevate a corner to a regular patch, we extrapolate horizontally and vertically, and then do a 2D extrapolation for the corner point.

The resulting border edges lie on the limit surface.

Slide 72

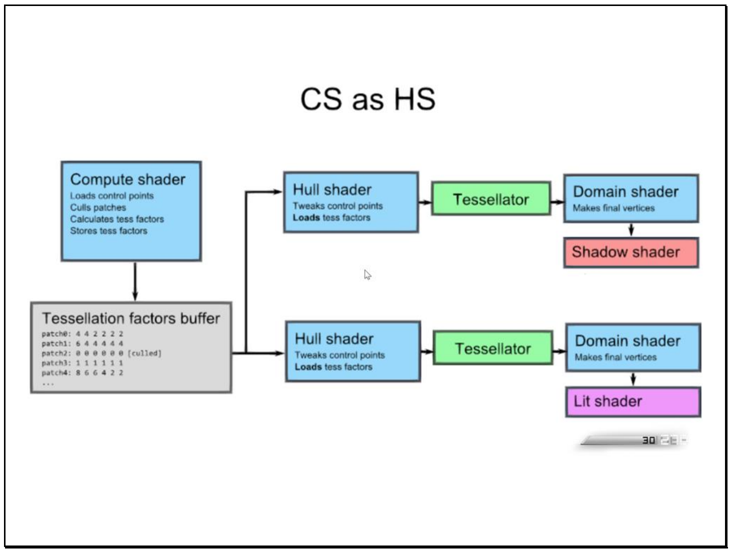


This scene demonstates how patch extension affects prop models.

The prop on the left has almost no regular patches, while the prop on the right is largely regular.

The purple edges on top and bottom of the canister are split edge borders.

Slide 73



CS as HS

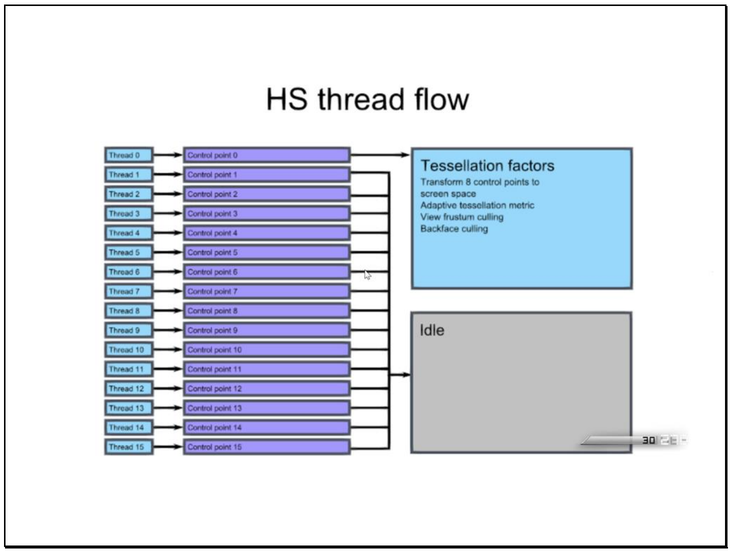
We found that given the limited geometry that passes through it after regular patch extraction, the global compute shader subdivision runs very fast. But the fastest way to do something is to not do it at all? So for rigid models, we store their globally computed irregular faces in a cache. This drops their compute time to 0, for some cost in memory.

The next optimization is moving the calculation of tess factors from the hull shader stage into a compute shader, at Neissner's suggestion.

We allocate a per-frame buffer to store the tess factors, packed down to 32bits per patch. Each model instance is assigned an offset into this buffer. At the beginning of the frame we run the CS for each model, fetching the patches and doing the tess factor evaluation, and then packing & storing the result to the buffer via a UAV. When the hull shader runs, it only has to unpack the tess factors and pass them on to the fixed function hardware.

Why is this faster? For one, the results can be reused for shadows and other passes. We use the same tess factors in the shadow pass as the lit pass to reduce self shadowing artifacts.

Slide 74

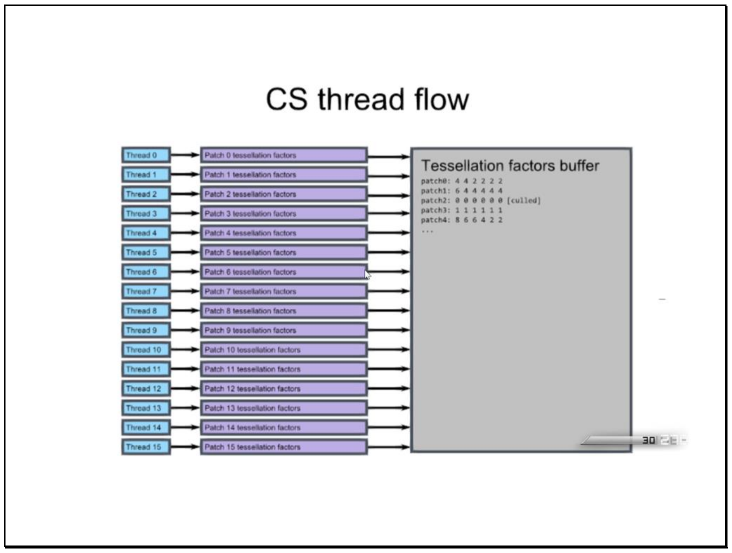


The other reason has to do with how hull shaders run. The DX11 hull shader is divided into two parts- the control point function and the constants function. The control point function runs one thread per cp, optionally altering its value. The constants function runs one thread per patch, after the control point function has finished, and outputs the tessfactors as well as any per-patch constants for the later shader stages.

Behind the scenes, these two shaders are actually compiled into one.The combined shader runs the control point shader as one thread per control point, and then the first thread continues on and runs the constants shader while the other threads go idle.

In our case, the most expensive calculation, evaluating 8 limit surface positions for curvature analysis, runs in the constants shader - on only 1 out of every 16 threads!

Slide 75



CS thread flow

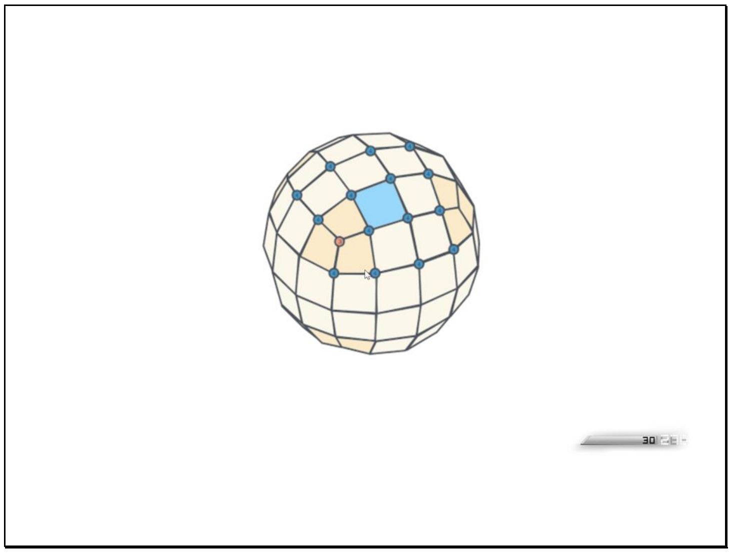
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Patch 0 tessellation factors
Patch 1 tessellation factors
Patch 2 tessellation factors
Patch 3 tessellation factors
Patch 4 tessellation factors
Patch 5 tessellation factors
Patch 6 tessellation factors
Patch 7 tessellation factors
Patch 8 tessellation factors
Patch 9 tessellation factors
Patch 10 tessellation factors
Patch 11 tessellation factors
Patch 12 tessellation factors
Patch 13 tessellation factors
Patch 14 tessellation factors
Patch 15 tessellation factors
Tessellation factors buffer

```
patch0: 4 4 2 2 2 2
patch1: 6 4 4 4 4 4
patch2: 0 0 0 0 0 0 [culled]
patch3: 1 1 1 1 1 1
patch4: 8 6 6 4 2 2
...
```

Moving the expensive calculation into a compute shader means that every thread runs a patch, giving full efficiency. Leaving the constants HS as a simple fetch / store also frees up hardware registers to run the DS and PS, meaning better parallelism for those expensive stages.

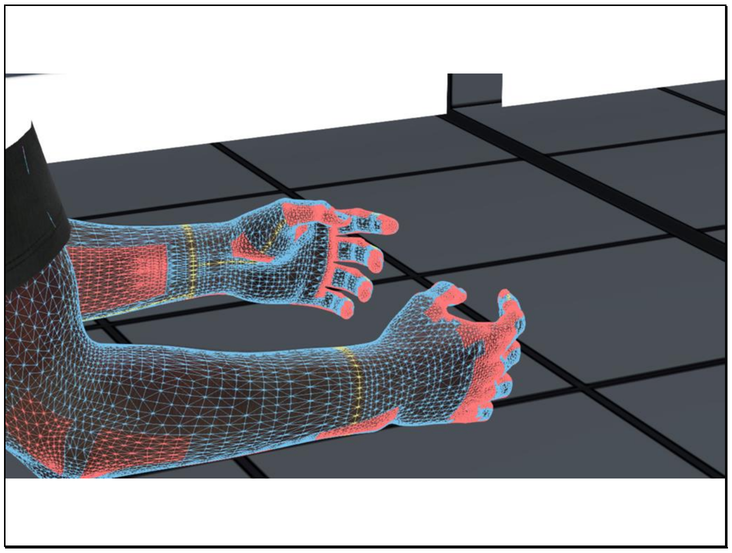
One caveat to this optimization is dispatch serialization. The HS dispatches are tiny, and we emit one per SubD surface. They all write to their own allocated segments of the buffer, but the driver doesn't know that and makes the dispatches run one at a time. This makes our compute code run much slower on PCs than on consoles.

Slide 76

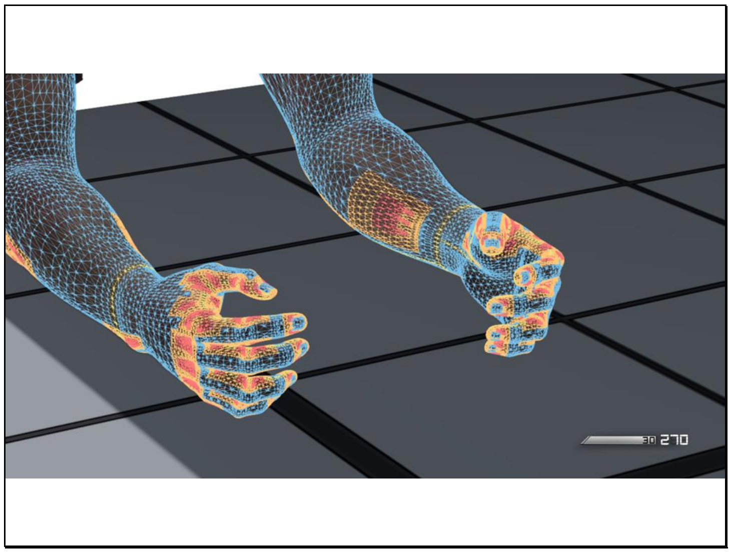


One negative performance issue that we ran into is that under-tessellation will cause performance to drop below that of the standard triangle pipeline. We found that 1x1 and 2x2 tessellation are typically a net perf loss, and 4x4 tessellation or better is where benefits appear.

Slide 77

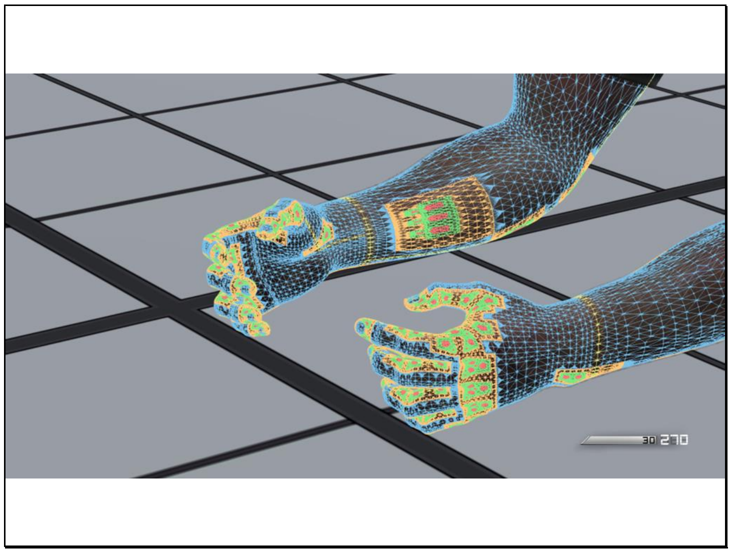


Slide 78



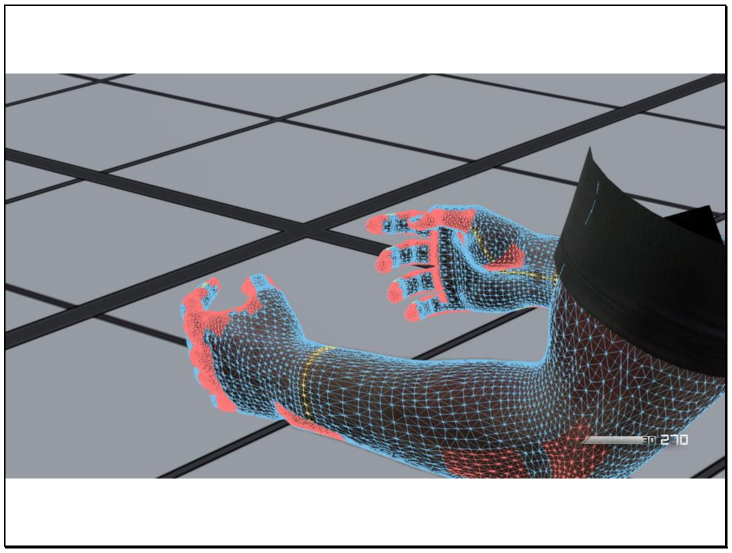
With each step of global subdivision, additional regular patches appear in the topology.

In this image, the orange patches are regular patches extracted after the first round of global subdivision.



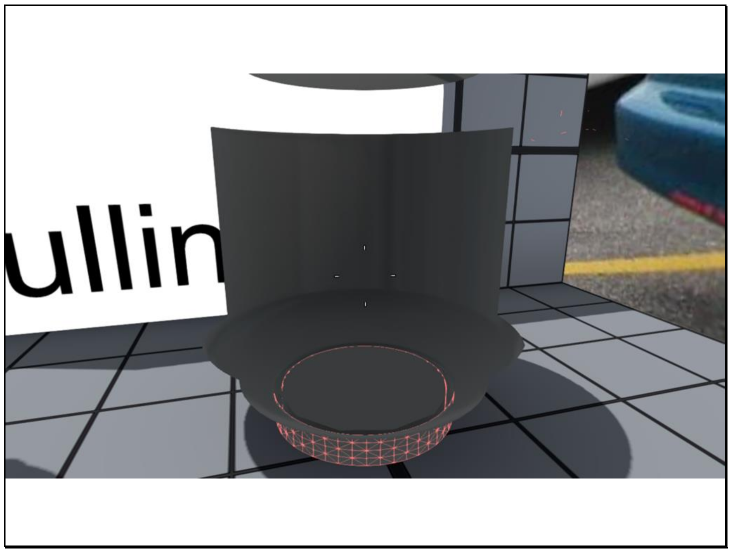
In this image, green patches are regular patches extracted after the second level of global subdivision.

After two steps, only tiny irregular patches are left.



Unfortunately, the under-tessellation of many small patches hurts performance.

For this reason, we chose to stop extracting regular patches after the base control mesh. While additional regular patches do appear after each global subdivision, they tessellate half as much as the prior level, dragging down performance.

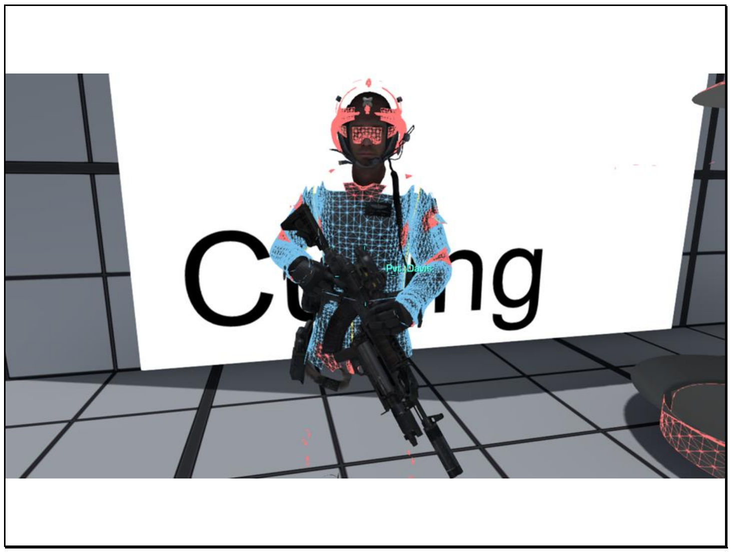


A nice thing about the DX11 tessellation pipeline is that you can set tessfactors to 0 for a patch, to discard entirely.

For rigid models, we precalculate a normal cone for each patch, and use a simple dot product to cull patches at runtime, per another of Neissner's papers. He actually helped us implement this one, along with his graduate student Jens Raab.

The culling happens in the compute shader, ad the result is stored in the dynamic packed tess factors buffer. When the hull shader encounters a patch with the cull bit set, it sets the tessfactors to 0 and returns.

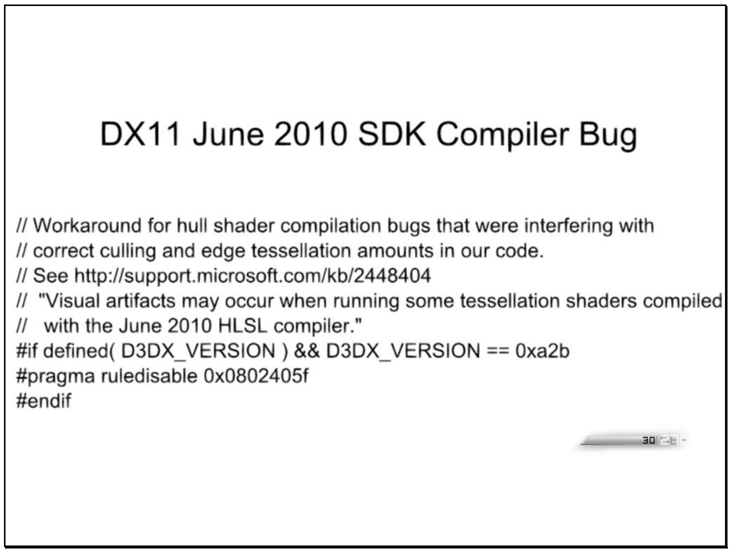
In this image, I’ve modified the hull shader to only display the backfacing patches.



We also do view frustum culling, taking advantage of the fact that the convex hull of the 16 control points of a B-spline patch forms a bounding volume.

In this image, I’ve modified the hull shader to cut clip space down to 1/3rd, showing how patches are rejected at the screen borders.

Slide 83

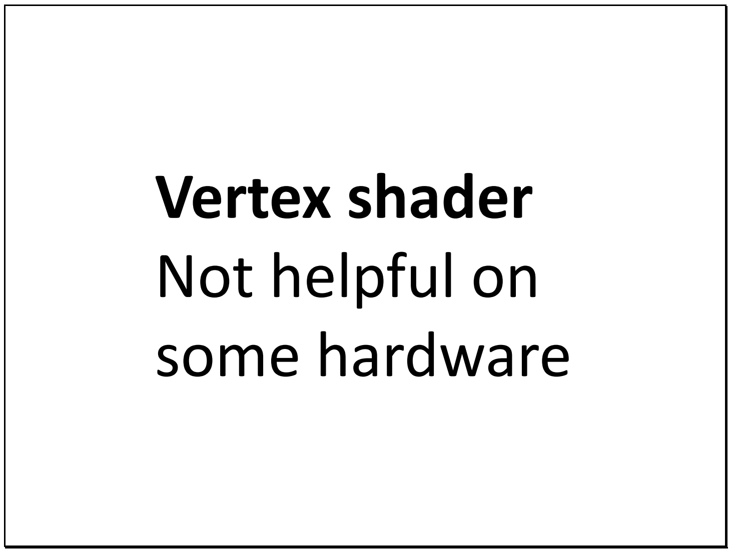


DX11 June 2010 SDK Compiler Bug

```
// Workaround for hull shader compilation bugs that were interfering with
// correct culling and edge tessellation amounts in our code.
// See http://support.microsoft.com/kb/2448404
//  "Visual artifacts may occur when running some tessellation shaders compiled
//   with the June 2010 HLSL compiler."
#if defined( D3DX_VERSION ) && D3DX_VERSION == 0xa2b
#pragma ruledisable 0x0802405f
#endif
```

If you are still on the DirectX 11 June 2010 SDK, there is a bug in the hull shader compiler that will cause your patches to go crazy.

Put in this magic line to fix it, or better yet, upgrade to the Win8 SDK.



Vertex shader

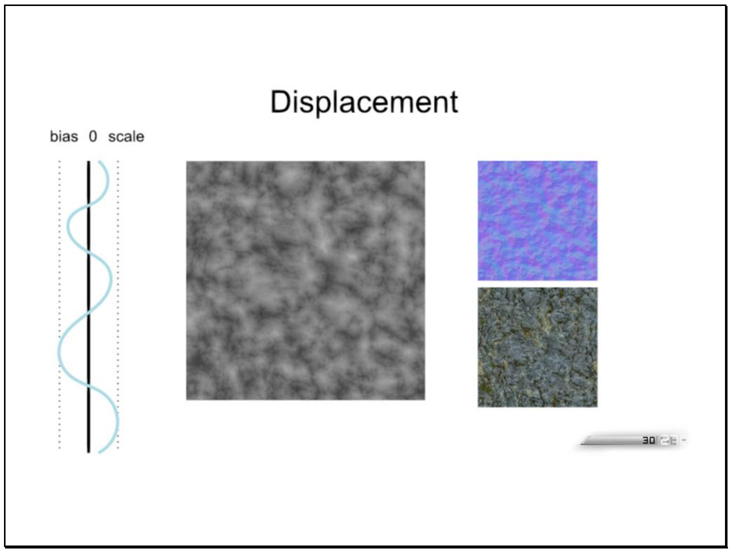
Not helpful on some hardware

Another negative performance issue is that on some hardware, the results of the vertex shader are not cached before being fed into the hull shader.

Our patches have 16 control points, and 12 are typically shared between consecutive patches.

We initially worked to move all redundant calculation into the vertex shader, but found this to be a net performance loss on some hardware.

Slide 85



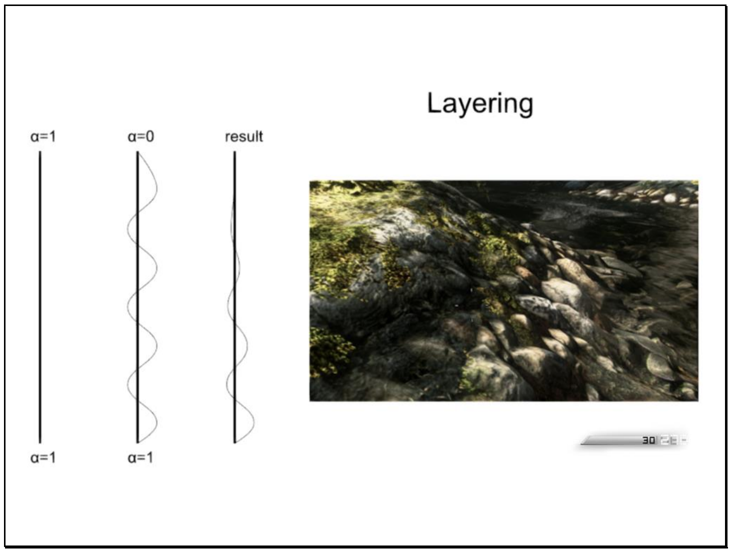
Displacement

The second, short part of the presentation covers our terrain displacement pipeline.

We use tessellated triangles to apply signed displacement maps to our BSP geometry. Artists can specify a positive or negative bias to the displacement map, along with a scale.

Note that the displacement maps only affect geometry- lighting is still derived from the normal map.

Slide 86



Layering

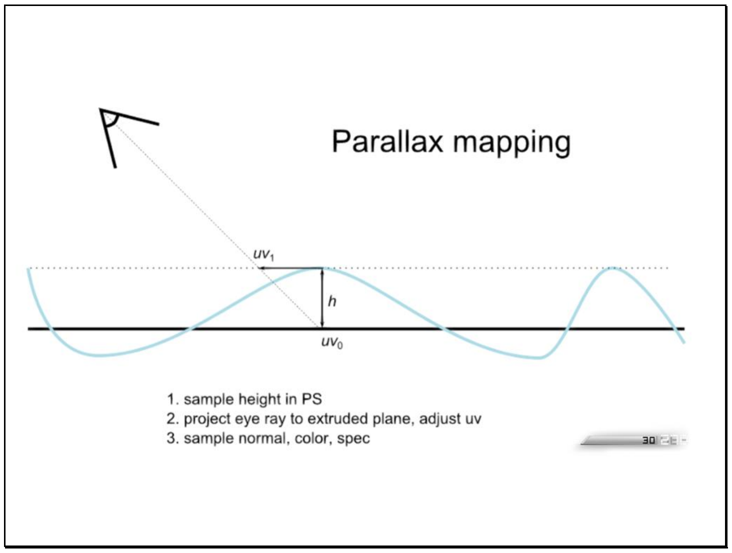
Our displacement system is based on our terrain renderer’s layering system.

Artists can combine multiple material layers together with painted vertex alpha, and the result is synthesized by the map compiler into a single blended material.

We support “add height” and “blend height” displacement blend modes, to allow artists to achieve different effects.

This image shows a displaced stone layer with additive moss displacement, blending into displaced smooth river bottom rocks.

Slide 87



Parallax mapping

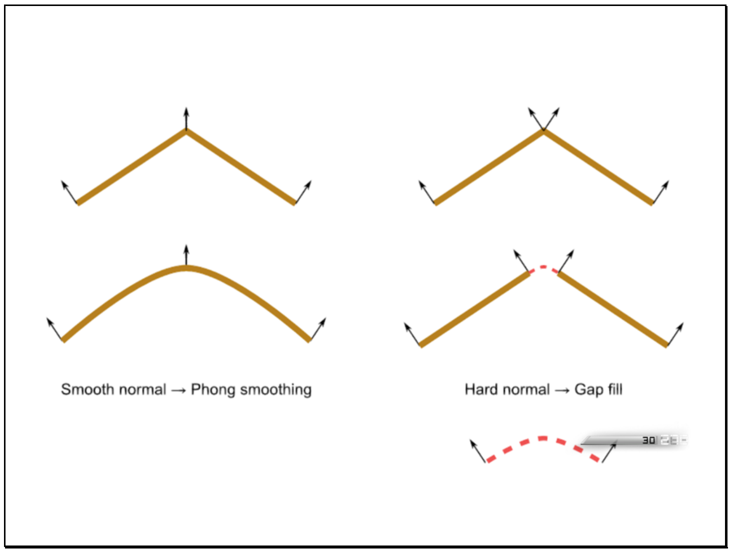
1. sample height in PS
2. project eye ray to extruded plane, adjust uv
3. sample normal, color, spec

Parallax mapping helped to smooth out some of the popping we would see when close to tessellated geometry.

Parallax mapping is very simple and fairly cheap- you sample the displacement map at the pixel, and then use that height to extrude the base geometry to an elevated plane, re-intersect the eye ray with that plane, and sample the color map, normal map, etc. from there.

Using parallax mapping enabled us to use integer tessellation factors in this pipeline as well.

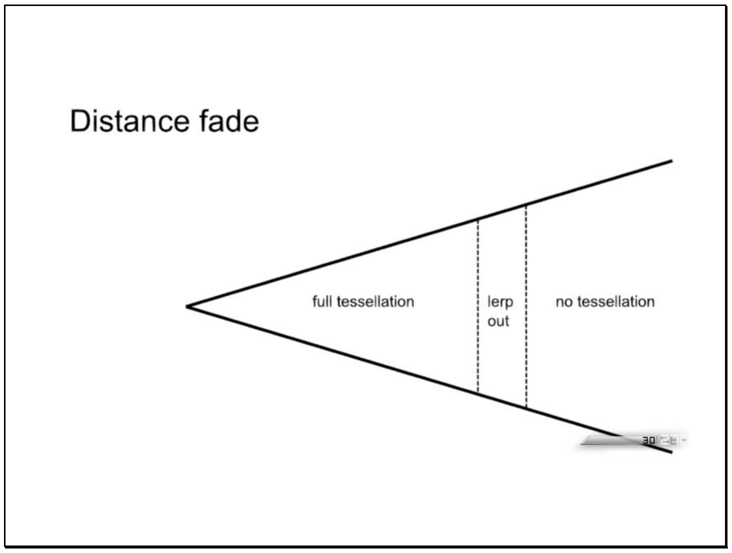
Slide 88



For cracks in dispaced geometry, we adopted two solutions. When edge normals are smooth, implying the artists intended smooth lighting, we use phong smoothing to smooth the geometry as well.

When normals are disjoint, we displace the surfaces along their normals, and the map compiler inserts edge geometry to fill in the cracks. The edge geometry inherits one normal from one face, and its other normal from the other face, resulting in a smooth interpolation. When the faces have different materials, the gap geometry is a blend between them.

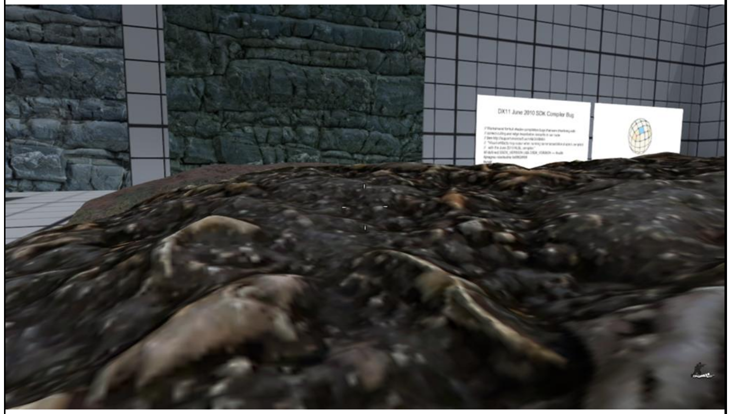
Slide 89



Distance fade

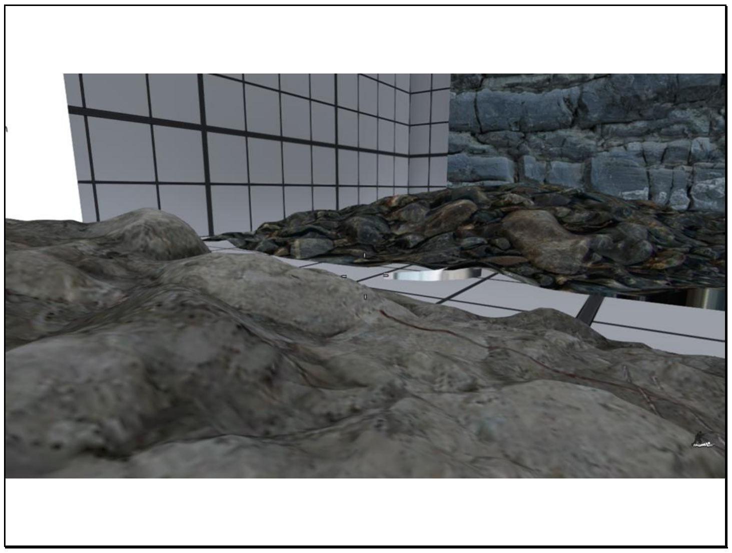
Finally, for performance on consoles and lower spec PCs, we disable tessellation in the distance. You get full tessellation up to a fixed range, then tessfactors lerp down to 1, after which we render with standard vertex and pixel shaders.

Slide 90



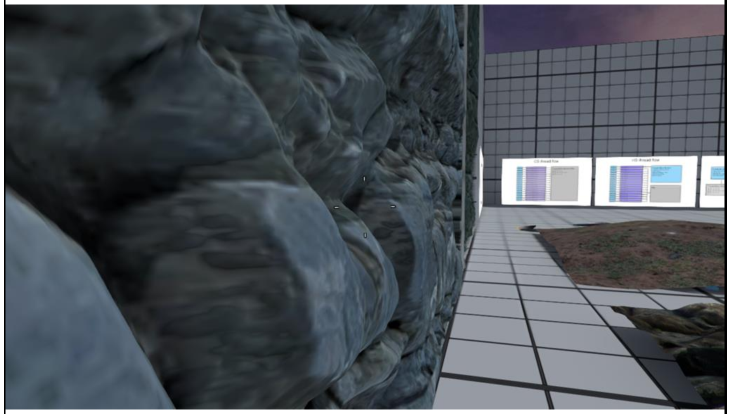
Here are some images of example materials rendered using our displacement pipeline.

Slide 91



Slide 92

Slide 93



Slide 94