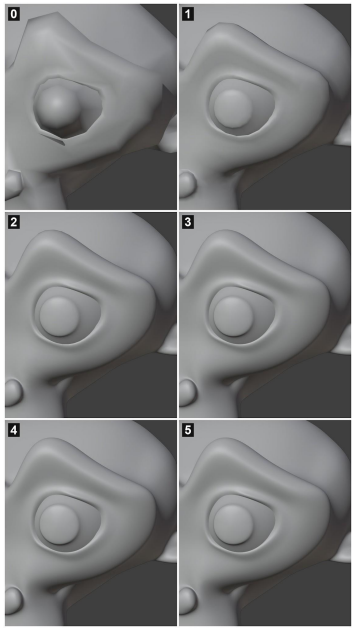
The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

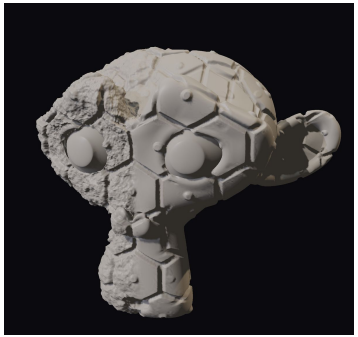
Adaptive Tessellation and Subdivision

John Hable

Subdivision Surfaces



- Catmull-Clark Subdivision [Catmull & Clark]
- Geris's Game - 1997
- From low poly mesh to high poly smooth surface
  - Displacement
- Clean LOD transition



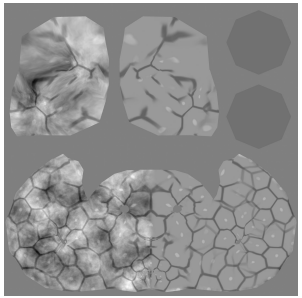
Triangle Meshes

- High-res triangle meshes
- Topology: 12 bytes per triangle
  - 3 indices (4 bytes each)
- Vertices: ~64 bytes per Vertex
  - Position (3 floats, 12 bytes)
  - Normal (3 floats, 12 bytes)
  - Tangent/Flip (4 floats, 16 bytes)
  - UV (2 floats, 8 bytes)
  - Color (4 floats, 16 bytes)
- Raw Data: ~44 bytes per triangle
  - 12 bytes for connectivity
  - 32 bytes for vertices (1 vert = 2 tris)

Compression

- Virtualized geometry (Nanite): [Karis et al. 2021]
  - 5.6 bytes per Nanite triangle on disk
- DGF: Block-based Geometry Supercompression [Barczak 2026]
  - Built for hardware ray tracing
  - 2.3 bytes per triangle
    - Extra cost for extra attributes
- Draco (GLTF) - [Draco] [Rossignac]
  - Quality depends heavily on settings.
- Arbitrary Baseline: 5 bytes/tri

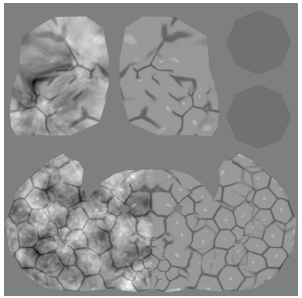
Subdivision Displacement



- Subdivision surface + offset displacement
  - Infinite smoothness is "free" - in terms of disk space
    - Memory and performance is not!
  - 8-bit displacement is often enough for real surfaces
- Suzanne (Blender Monkey)
  - Modeled as multiresolution displacement

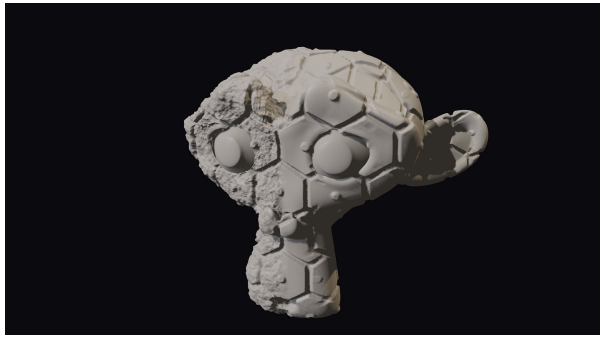
Fixed disk cost

Storage	bits / triangle	× vs baseline
Triangle baseline (5 B/tri, hypothetical)	40	1×
8-bit displacement (1 B/texel)	6.2	6.5×
Monkey height JPEG (2K, lossless, 1.19 MB)	1.74	23×



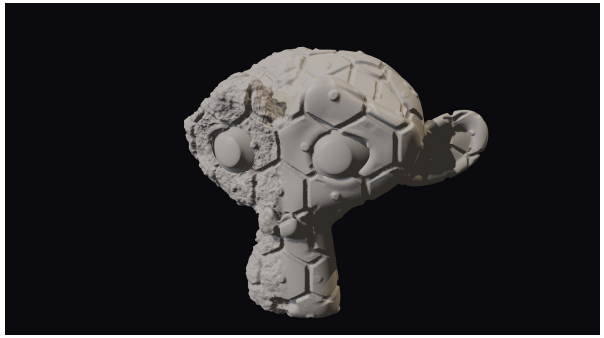
- Subdivision data is a fixed cost
- Baseline Estimate: Store triangles at 5 byte/tri.
  - Actual number will be higher or lower depending on technique
- Monkey (Blender's Suzanne)
  - 2K height map, lossless JPEG
    - 64.9% UV coverage
  - 1.19 MB for ~5.45M triangles (1 texel = 2 triangles)

Caveats



- Loss when using displacement.
  - Resampling
  - 8-bit quantization
- Shader Techniques
  - Standard Tiling, Triplanar mapping, Hex tiling, Detail Textures
  - Layering / Blending
  - Fully procedural shaders
- Limitation: No Topology Changes
  - Overhangs, Folds

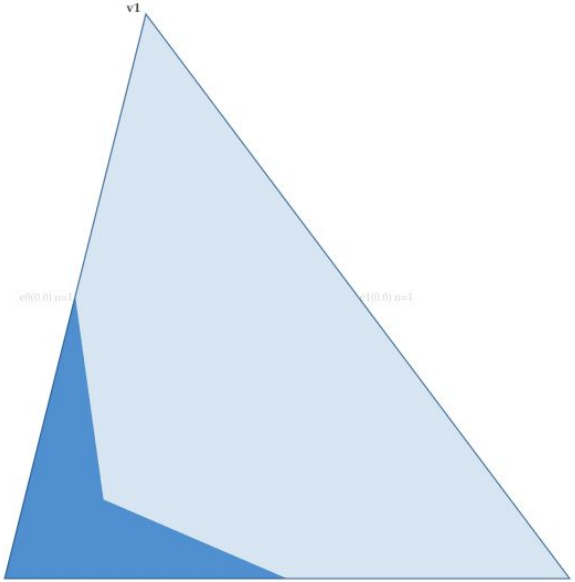
Agenda



1. Tessellating a triangle
    - Including a deep dive into DX11 hardware tessellation
    - Then clamped parallelogram tessellation
2. Tessellating a Mesh
    - Adaptively
    - With Compute
3. Subdividing a Surface
    - Also with compute

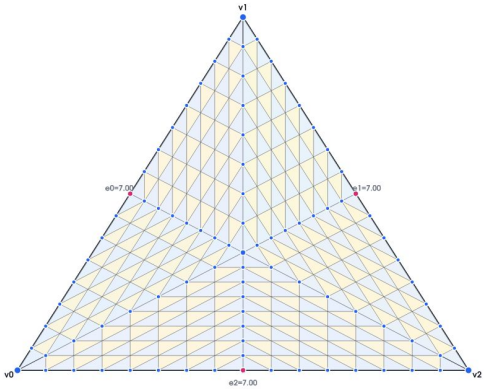
Full explanations, animated diagrams, and MIT licensed WebGPU examples. (filmicworlds.com)

Part 1: The Triangle

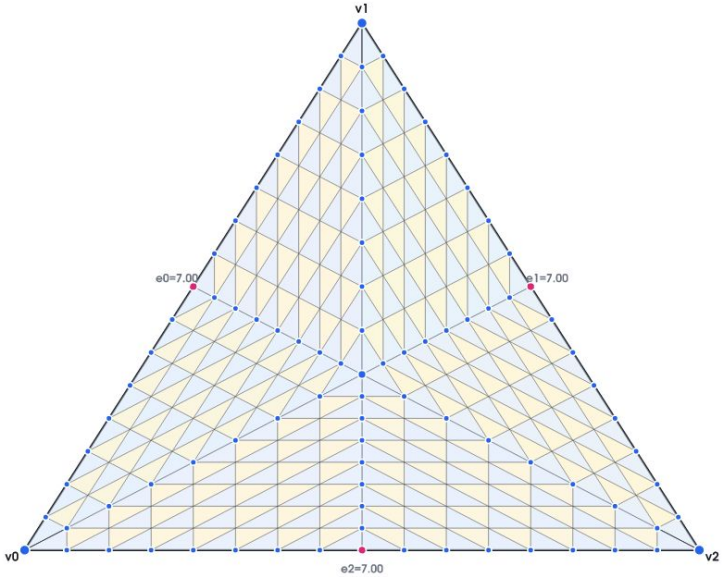


20 years of hardware tessellation

- DX11-style tessellation traces to Xbox 360 (2005) [Microsoft]
- Rooted in Pixar's Reyes: dice primitives into micro-tris [Cook et al. 1987]
- Games shipped with Tessellation. (Gears of War 2, Halo: Reach).
- Activision's Call of Duty: Ghosts used Tessellation and Catmull-Clark Subdivision [Brainerd 2014]



Key tessellation requirements

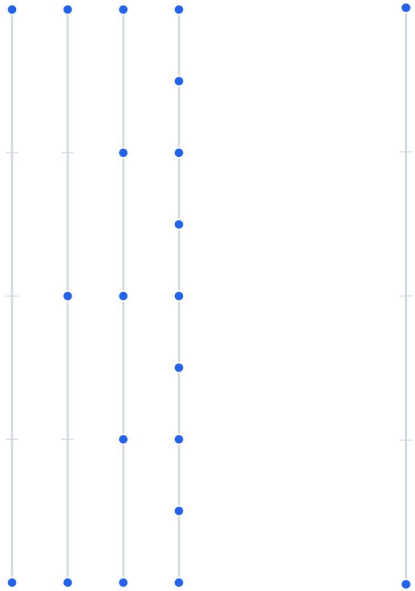


- Fractional tessellation factors: No popping!
- All three edges independent of each other
- Several differences with DX11 tessellation:
  - DX11 tessellation is symmetric, but we are going to break this constraint.
  - DX11 has an interior tess factor, but our interior pattern is implicitly defined by the edge tess factors.

Tessellating a line

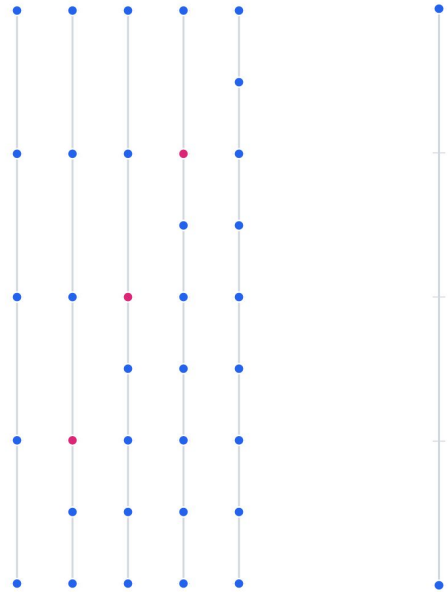
- Before we can tessellate a triangle, we need to understand how to tessellate a line.
- Each edge has a float tess factor (DX11: [1,64])
- No popping!
- This is what the final pattern will be. So let's derive it.

Tessellating a line: Powers of Two



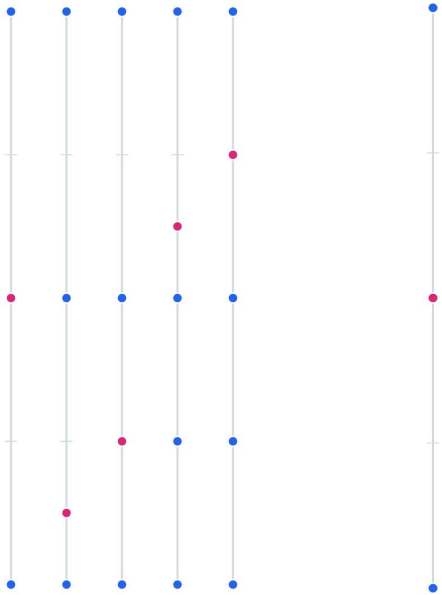
- What is the simplest solution?
- Just subdivide at each power of two.
  - 1 -> 2 -> 4 -> 8 -> 16 -> 32 -> 64
- Results in a very strong pop.
  - But how can we make it transition between levels?

Tessellating a line: One at a time



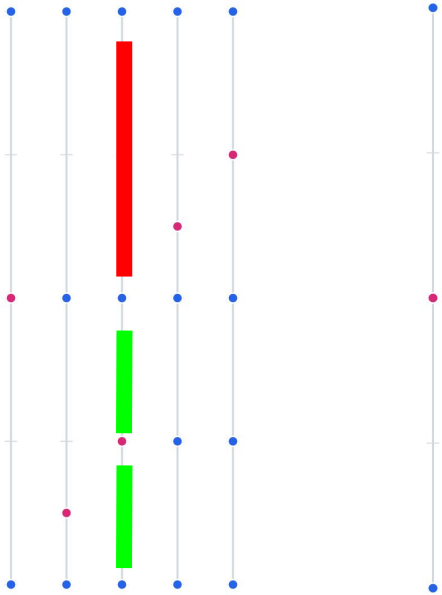
- Add one new point at a time.
- When going from 4 -> 8
  - Add points 5, 6, 7, 8 one at a time.
- Exact same pattern as before

Tessellating a line: Distribution



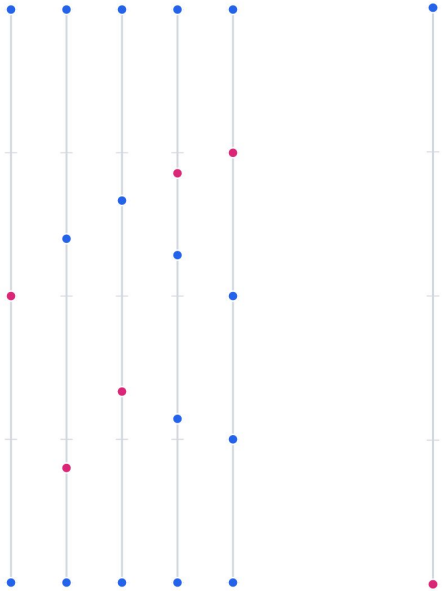
- When adding a point, have it "slide" in from the previous point
  - Added point is in red
- All other points stay locked
  - Only the pivot moves
- Resolves to the same position as before
- Results in lopsided distribution.

Tessellating a line: Distribution



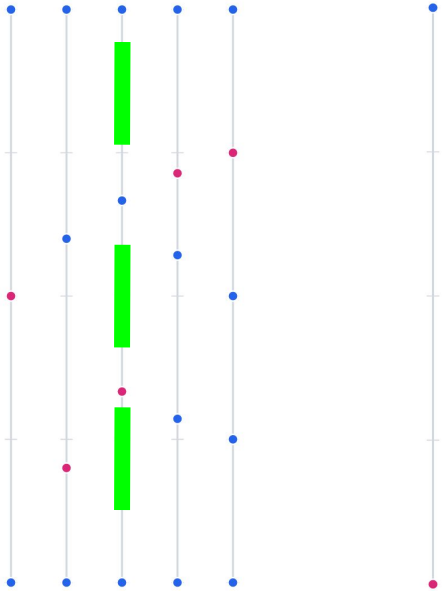
- When adding a point, have it "slide" in from the previous point
  - Added point is in red
- All other points stay locked
  - Only the pivot moves
- Resolves to the same position as before
- Results in lopsided distribution.
- Tess factor of 3
  - Two small green segment.
  - One big red segment.

Tessellating a line: One at a time



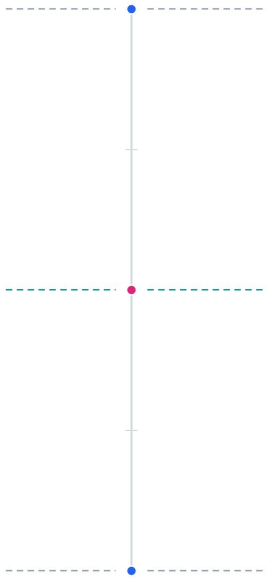
- Adjust the distribution based on new points.
- Example: Tess factor 10.3
  - 11 segments.
  - 10 segments at 1.0, sliding segment at 0.3
- Lands at exact same position at powers of two
- Keeps a more even distribution

Tessellating a line: One at a time



- Adjust the distribution based on new points.
- Example: Tess factor 10.3
  - 11 segments.
  - 10 segments at 1.0, sliding segment at 0.3
- Lands at exact same position at powers of two
- Keeps a more even distribution

Tessellating a line: One at a time

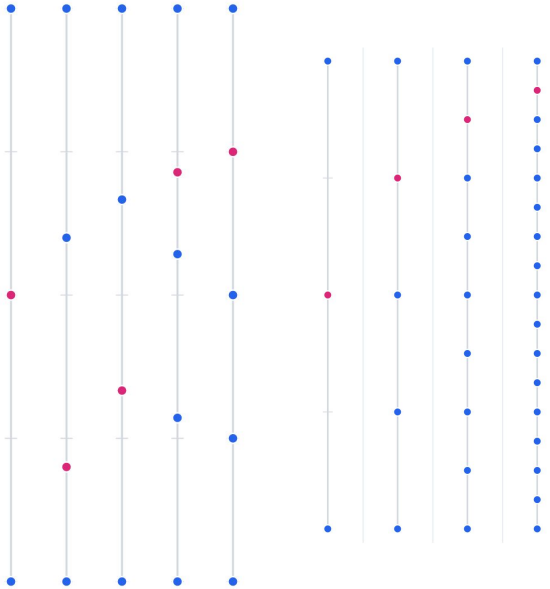


- Take the previous pattern and mirror it.
- Minimum tess factor is 2.
  - Midpoint always enabled
- This is DX11, Fractional-Even Tessellation

DX11 Line Segments

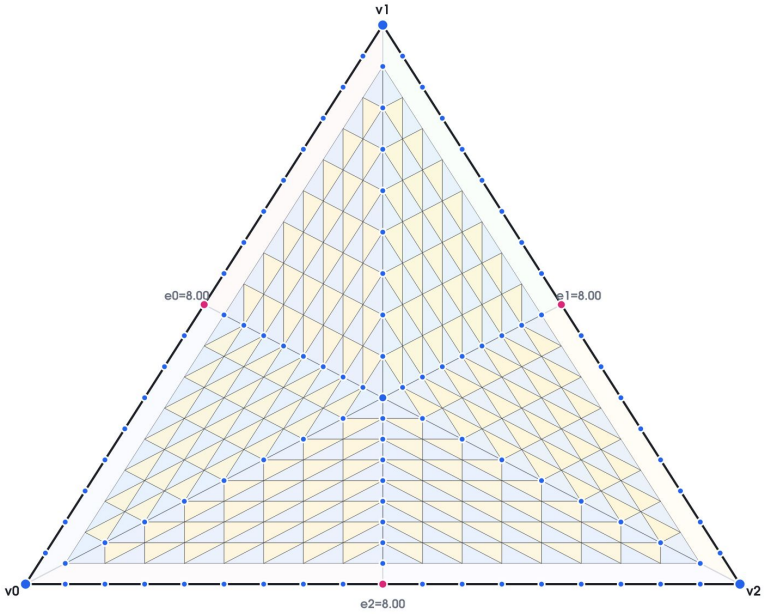
- The distribution is very even
  - ..but at the cost of very low stability
- The simpler algorithm has a lopsided distribution.
  - But the points are stable.
  - Less aliasing with displacement

Why DX11 does not converge



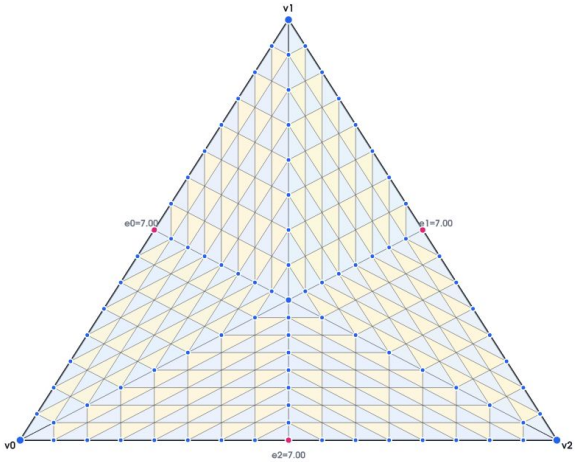
- As each point is added (in red)
  - Points above are pushed UP
  - Points below are pushed DOWN
- During each octave, the middle point: 0.5 -> 2/3 -> 0.5.
- The figure on the left is the first octave.
- The animated mesh shows the 4 octaves.
  - 2->4, 4->8, 8->16, 16->32.

DX11 Triangles Tessellation



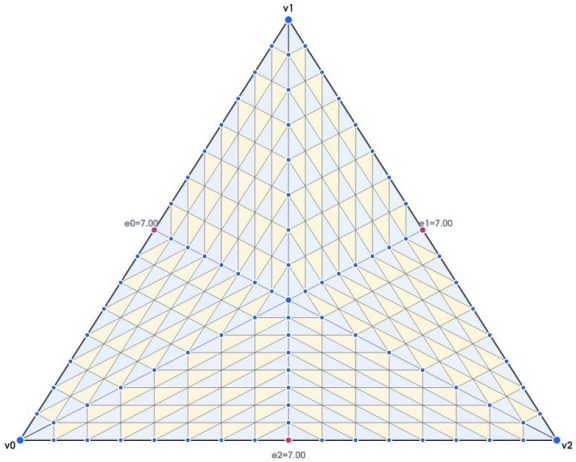
- How does DX11 Tessellation actually create a triangle?
- Split the triangle into 6 regions, triforce-tessellation
- [Microsoft], [Giesen 2011]
  - Discard the outer strip so each edge can differ

Each edge moves on its own



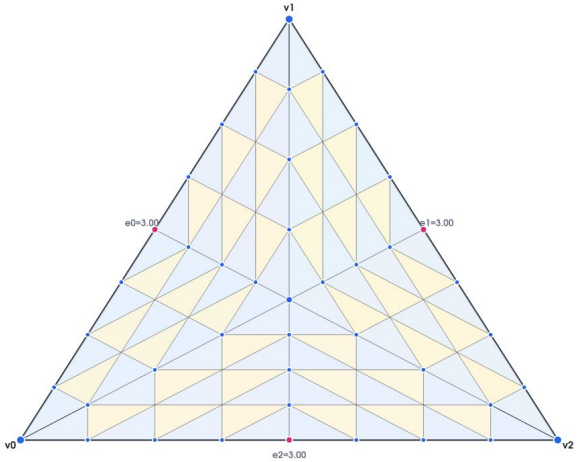
- Edge tess factors are separate from internal tess factors
- Drop one edge at a time to 0, then bring it back
- The gap-matching strip absorbs the difference; the interior never re-flows

DX11 interior: the internal tess factor



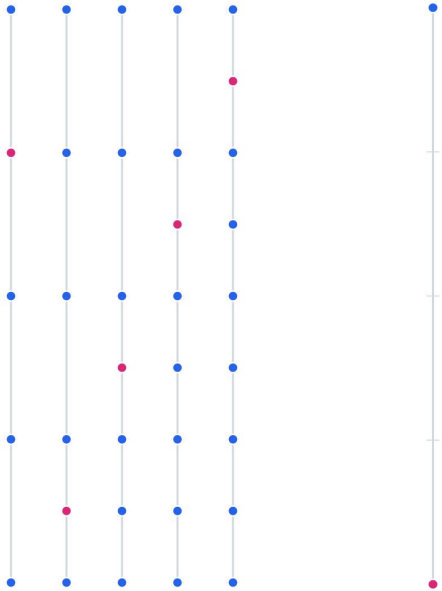
- The internal factor drives the inner tessellation density
- Edges fixed here; only the interior grows

DX11 Tessellation



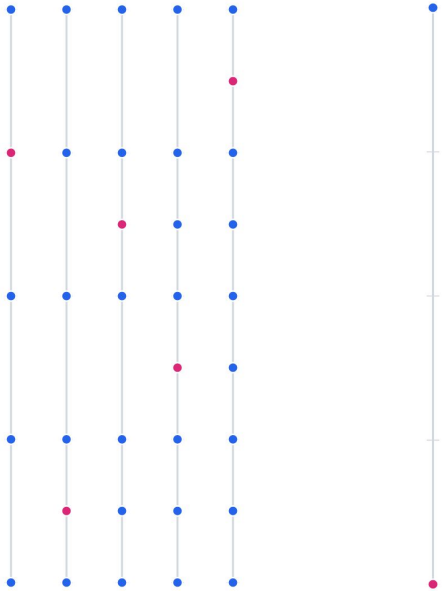
- And that is fractional even.
- Each edge can carry a different factor
- Interior can change as well.
- Ideas for improvement?

Fixing the line first



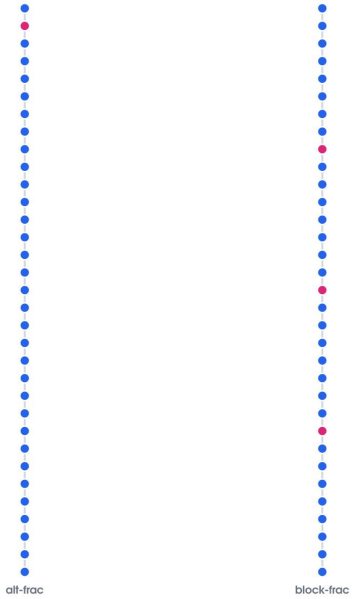
- Revert to stable "add one at a time" spacing
- Improve stability.
- …which means regressing the distribution.

Fixing the line first



- Insert odd points before even for better balance
  - Same segments, but less lopsided
- Note the 3rd column

Fixing the line first

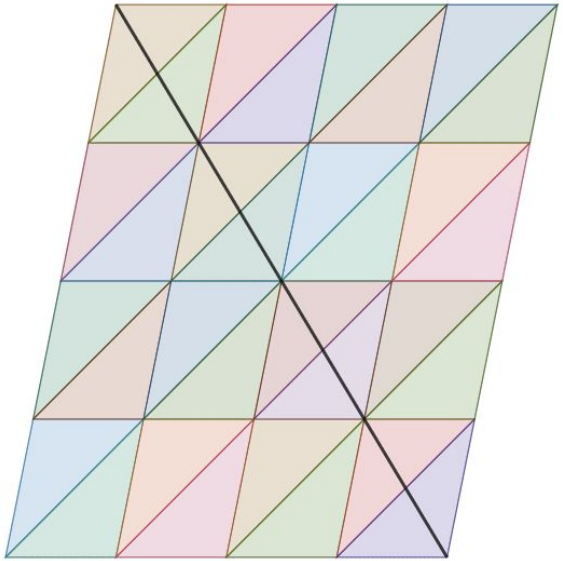


- Previous algorithm (left)
  - Note how red points seem like they pop in.
  - Technically it’s continuous
- Instead of quickly sliding in one point at a time, slowly slide in multiple points
  - Note the points in red moving together (right).

Fixing the line first

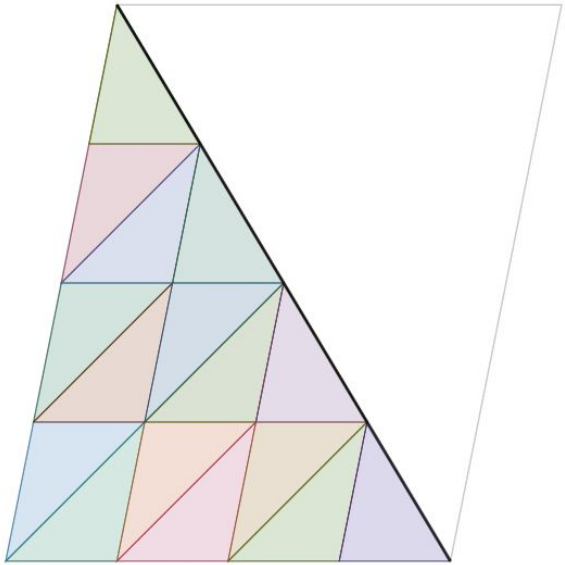
- Clean 1->2 transition: slide in from both ends
- DX11 fraction even has a range of [2,64]
  - Technically, you can pass in 1, but it gets clamped to 2.
- Instead, allow a transition from 1->2

The clamped parallelogram



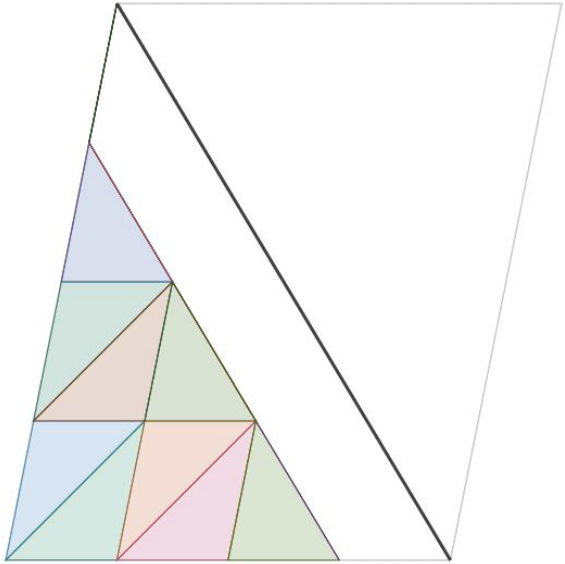
- Start with a parallelogram from two edge factors
- Use the 1D point algorithm
  - In both U and V
- Then create a parallelogram.
- Pros: Straight!
- Cons: Not a triangle.

The clamped parallelogram



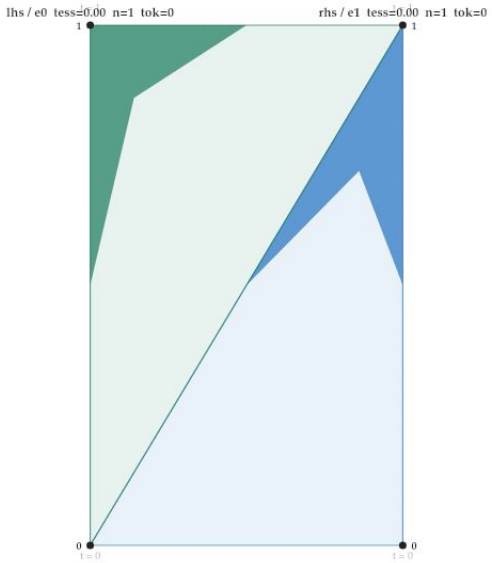
- Start with a parallelogram from two edge factors
- Clamp it into a triangle: u = min(u, 1 − v)
- Add a gap term for differing L/R edges
- Creates degenerate points
  - Solvable problem
- Problem: Left and right forced into same tess factors.

The clamped parallelogram



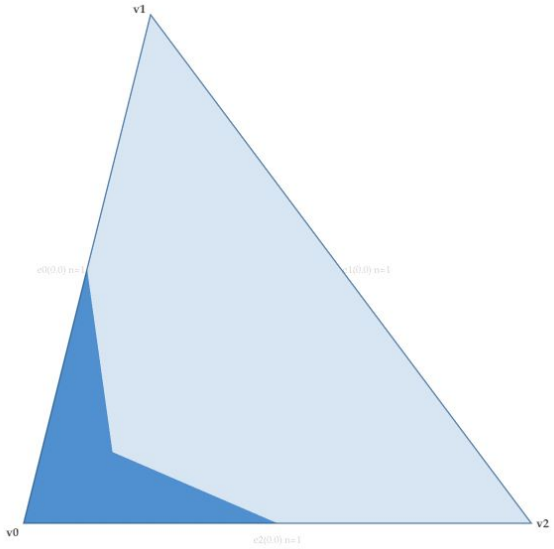
- Mind the gap.
  - Both the horizontal and vertical 1d points have a gap from the last point.
  - Use the max of both as the gap
- Clamp it into a triangle:
  - u = min(u, (1 - gap) − v)
- This constraint leaves an open area.

Matching the gap



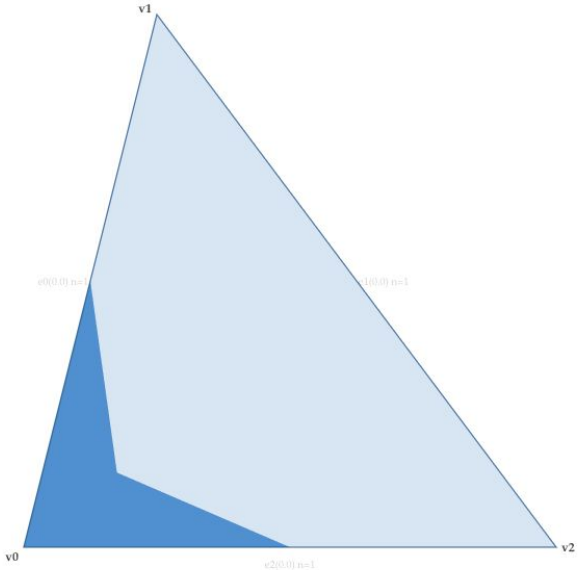
- Matching algorithm to fill the gap.
- Built offline as a table with a fast lookup.
- Algorithm:
  - Round up to the nearest power of two
  - Then match across
  - Details are a bit finicky. See sample code for details.

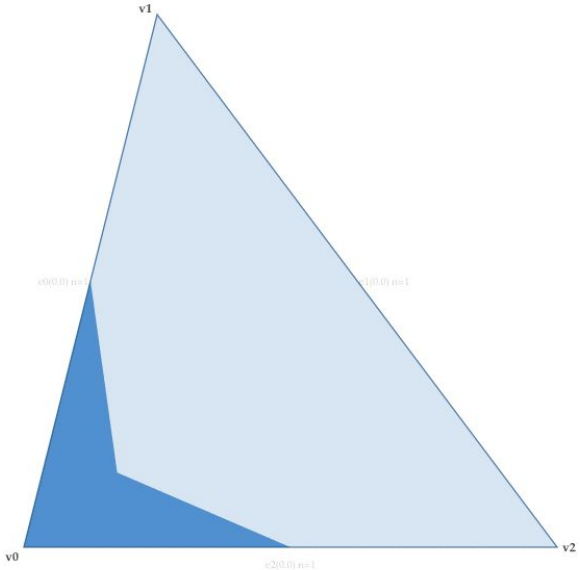
The full tess pattern



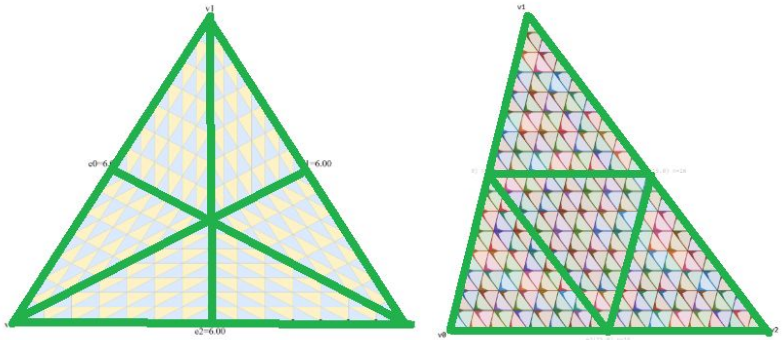
- For low tess patterns, we need an extra point in the middle.
- The clamped parallelogram only kicks in when the tess factor of the left edge is at least 2.
  - It doesn’t work when the points on the left edge are sliding from 1->2
- Otherwise, a center points is added.

Full pattern



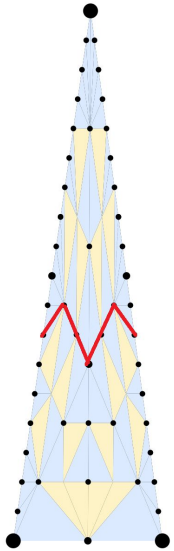


DX11 vs. clamped: inefficient patterns



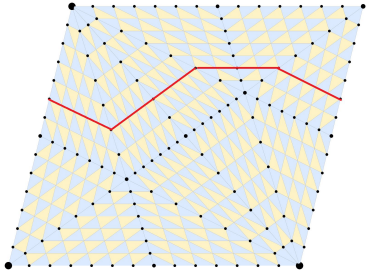
- DX11: 6 regular regions - more triangles than needed
  - Clamped Parallelogram is 4
  - 1/3 less triangles at equal power of 2 tessellation.
  - In this case, Interior and Exterior tess factors are 16.

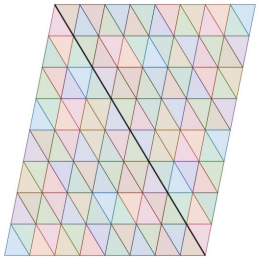
DX11 vs. clamped: varied factors



- Thin triangle
- DX11: single internal tess factor
  - Impossible to have more internal points vertically than horizontally
  - Either rows are under-tessellated or columns are over-tessellated
- Clamped Parallelogram: Gracefully handle long, thin triangles.

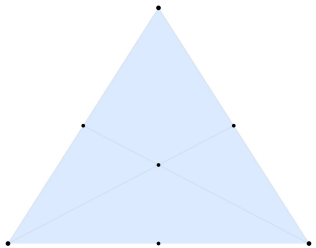
DX11 vs. clamped: bending & edge loops

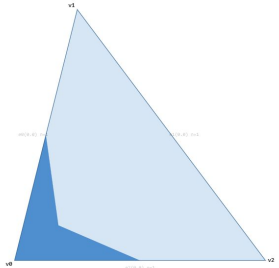




- Deformation wants straight edge loops - no sharp bends
- DX11: no clean loops (quad primitive - mixed tri/quad
- Clamped: regular quads, loops that degrade gracefully
  - This is why we explicitly reject rotational symmetry

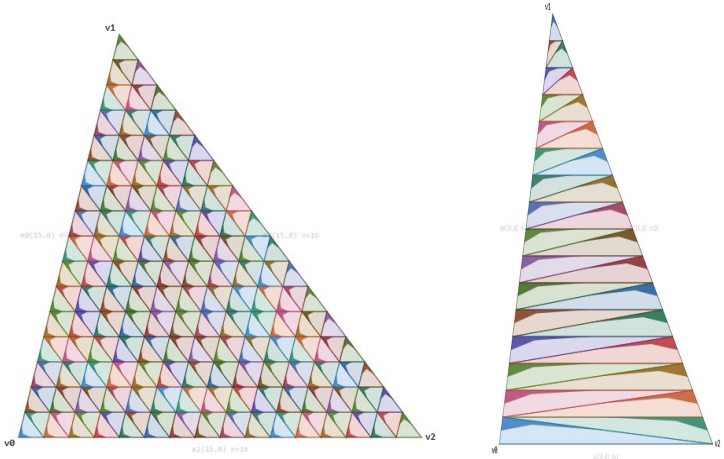
DX11 vs. clamped: minimum triangles





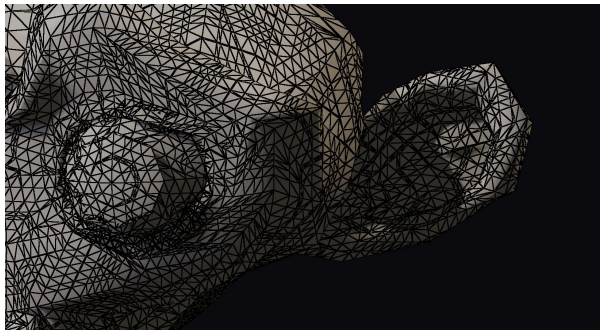
- DX11 fractional even: minimum 6 triangles.
  - Simply “turning it on” causes a hefty increase in triangles.
  - Fractional odd can go from a single triangle.
    - At the expense of even more instability.
- Clamped Parallelogram
  - Transitions all the way from a single triangle

Implementation Details



- Store everything as tables.
  - 25 unique 1D lines
  - Each line has a start and end t for each position
- Clamped Parallelogram Tables
  - Based on left edge and bottom edge
    - 25*25 tables.
- Gap Table
  - Based on left edge and right edge
    - 25*25 tables
- Total tables size: 1.6MB

Part 2: The Mesh

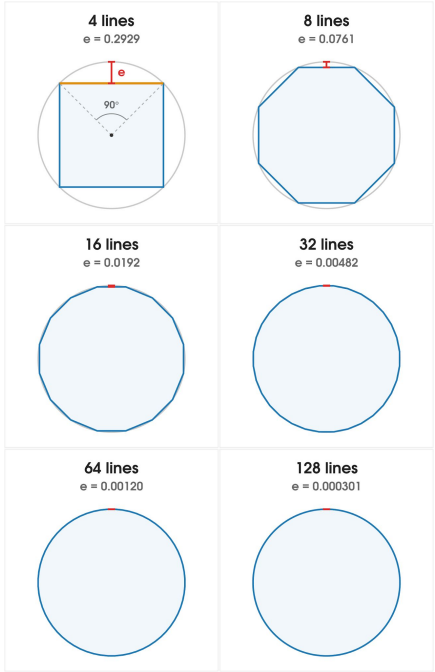


- Full Compute Adaptive Tessellation
  - Choose tess factors for edges.
  - Tessellate into large, flat buffers.
- A reference baseline:
  - Vanilla WebGPU, no extensions
- Algorithm options?

Recent Compute Tessellation Algorithms

- Nanite Tessellation [Karis 2026]
  - Integer tess factors, nested splits
  - Pattern pops when tess factors change
    - Not visible with sub-pixel triangle error, similar to Reyes (Reyes)
- Concurrent Binary Trees / Longest-edge bisection [Dupuy 2020], [Deliot et al. 2021]
  - Recursive cascading splits
  - Dependencies between edges
- Goal: Fractional tessellation
  - Independent edges
  - No pops

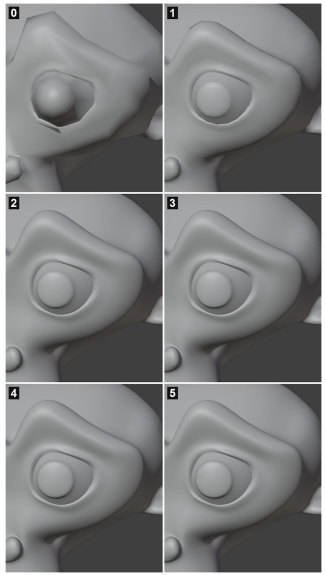
The 2D Rule: 1/8th



- In 2D, each level gives 2× more line segments for a 1/4 reduction in error
- 2× the cost for 1/4 the error means each level is 1/8th as efficient as the last
- Huge early gains, rapidly diminishing returns

Level	Lines	Chord angle θ	Error e = 1 − cos(θ/2)	Reduction vs prev
0	4	90°	0.2929	—
1	8	45°	0.0761	3.85×
2	16	22.5°	0.0192	3.96×
3	32	11.25°	0.00482	3.99×
4	64	5.625°	0.00120	4.00×
5	128	2.8125°	0.000301	4.00×

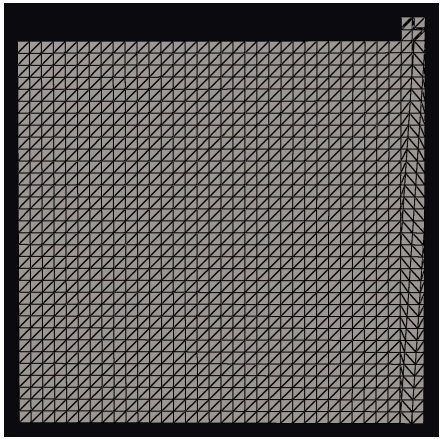
The 3D Rule: 1/16th

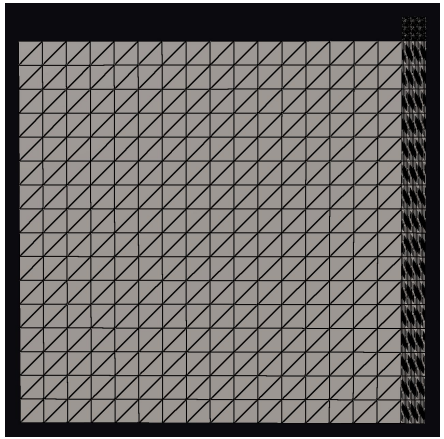


Level	Triangles
0	968
1	3,872
2	15,488
3	61,952
4	247,808
5	991,232
6	3,964,928
7	15,859,712

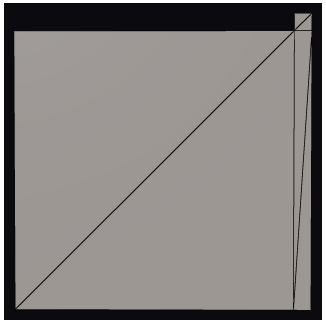
- Same error reduction per level (1/4)
- Each level is 4× the triangles
  - As opposed to 2x for 2D.
- 1/4 the error for 4× the cost for each level is 1/16th as efficient as the last
- Gains diminish even faster than 2x.

Real meshes have unevenly sized polys

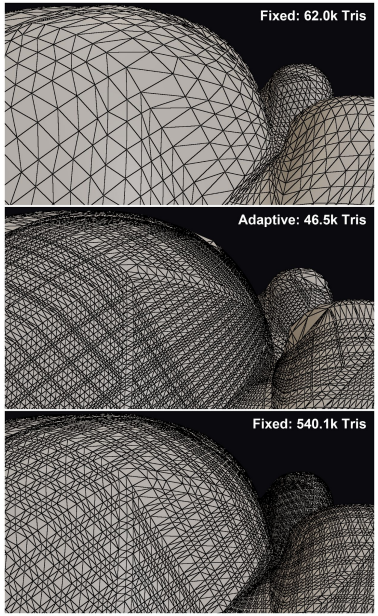




- Simple case, with three initial quads.
- Fixed tessellation:
  - Over-tessellate the thin and/or small quads
  - Under-tessellate the large quads.
- Left: Adaptive tessellation
- Right: Fixed tessellation

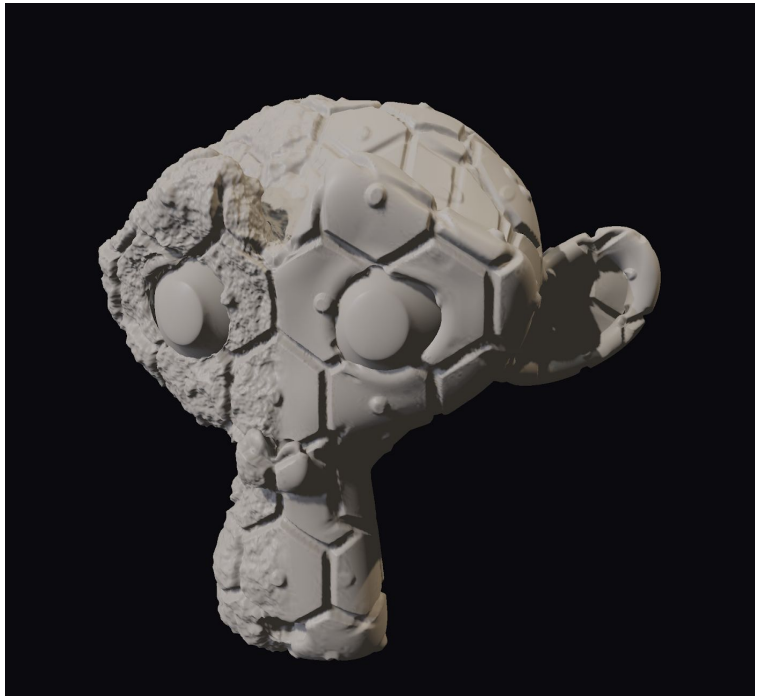


The payoff: adaptive vs fixed



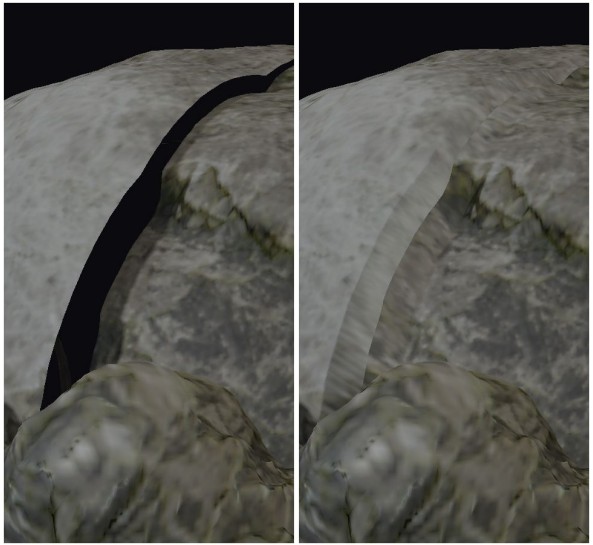
- Top image is fixed, at 62.6k triangles
- The middle image is adaptive. It provides a vastly finer tessellation despite fewer triangles (46.5k)
  - More aggressive tessellation of larger triangles.
  - Less aggressive tessellation of smaller and offscreen triangles.
- To achieve similar tessellation size, we need about 560.1k tris

Big buffers



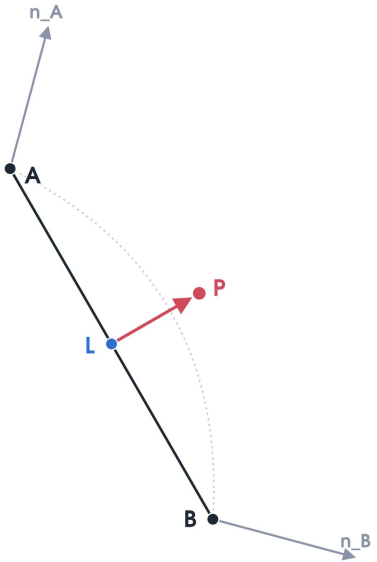
- Store buffers for 16M tris and 16M verts
  - Indices (3 u32 per tri): 192 MB
  - Positions (3 f32 per vert): 192 MB
  - Normals (3 f32 per vert): 192 MB
  - Barycentrics (1 u32 per vert, packed 2x16): 64 MB
  - Src Tri Index (1 1u32 pververt)
- ~704 MB at the 16M-vert / 16M-tri cap
  - Lots of room for improvement (mesh shaders!)
- Interpolate attributes in VS

Welding



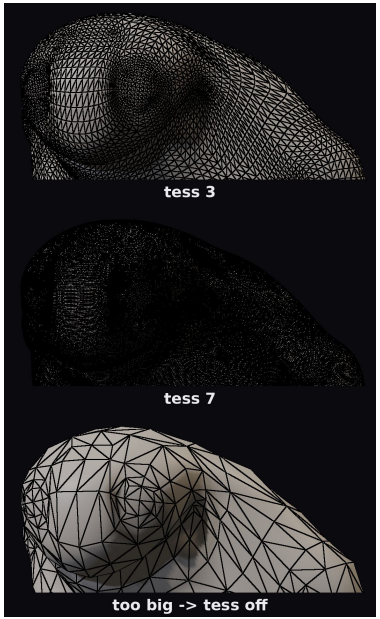
- UV seams
- Texture partial derivatives
- Roundoff error
- Material seams
  - Brick wall to a concrete wall
- Weld positions in a post-pass
  - Separate pass for corners vs edges
- Fundamental limitation with hardware tessellation.

Choosing the tess factors



- Inspirations: [Brainerd 2014], [Karis 2026]
- [Brainerd 2014] uses the distance from the linearly evaluated point to the midpoint.
- Instead, use the Phong approximation with the limit position and normal from both vertices.
  - Error is the offset from the Phong midpoint to the linear midpoint
- Using arc angle instead of screen projection

Too much data?

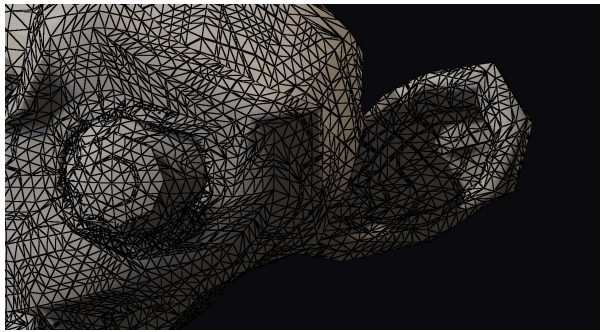


- What if we have too much data?
  - And it would overflow the buffer?
- Goal: Not crash
  - Good enough
- Tess factors first
  - Drop tess factors if we know we would go over
- Significant room for improvement
  - Half or quarter size

Multiple Render Passes

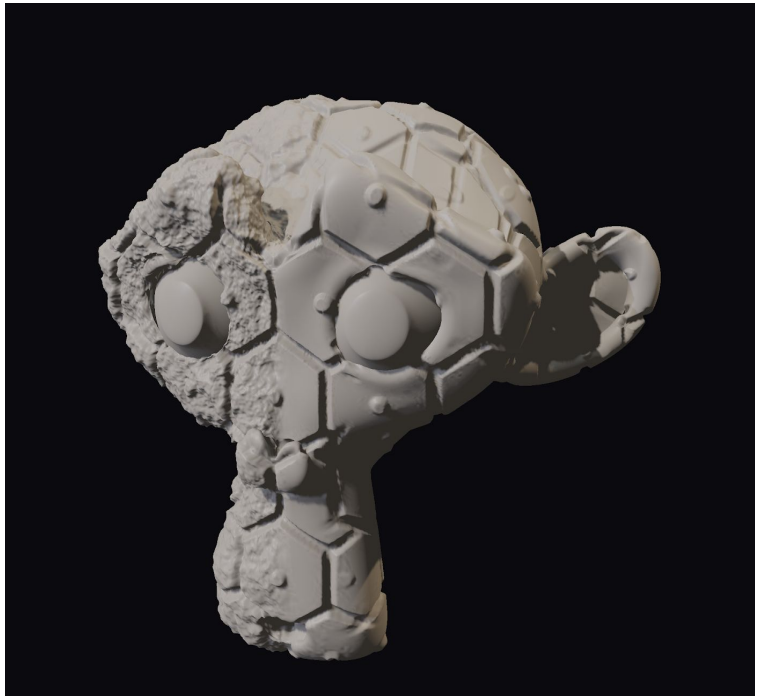
- DX11 tessellation both produces and consumes in the same pass.
  - Elegant: constant memory regardless of tessellation size.
  - Multiple Passes: Full calculation every pass.
- Most engines require multiple passes per frame:
  - Pre-Pass, GBuffer, Motion Vectors, Shadow Depth
- Store the tessellated verts once, weld
  - Reuse for each pass

Implementation: Big Tables



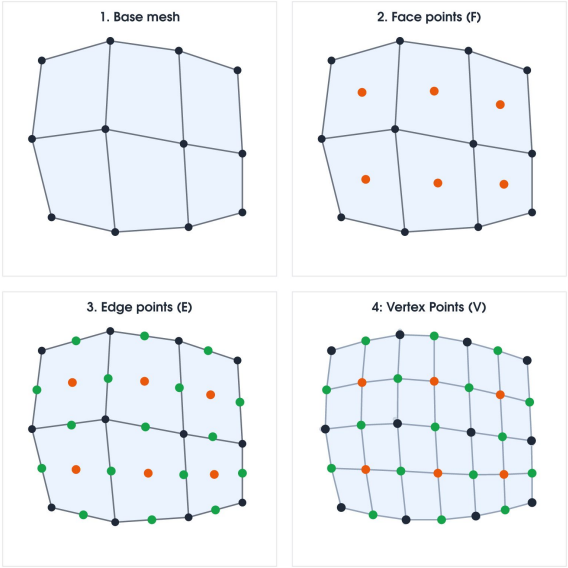
- For each combination of clamped parallelogram (25x25)
  - Precalculate a full table.
  - Size depends on max tess facto
    - 64: 2.5MB
    - 128: 8.6MB
    - 256: 30.2MB
- Also implemented with analytic, but table was faster
  - TODO: precalculate meshlets

Part 3: Catmull-Clark



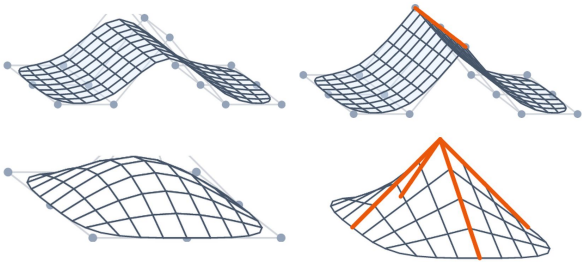
- Film standard since Geri's Game (1997)
- Artists author low-poly, subdivide at render
- Hierarchical displacement
  - Delta vs. the subdivision surface
  - ZBrush/Mudbox/Blender

Catmull-Clark



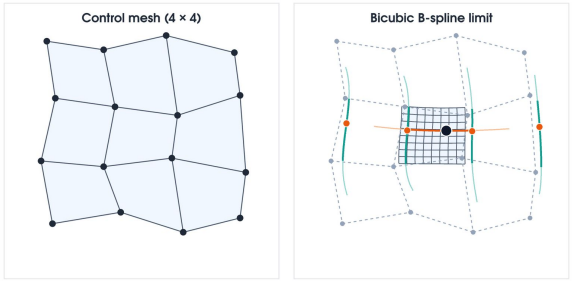
- For each level:
  - Calculate Face Points
  - Calculate Edge Points
  - Calculate Vertex points
- It's stable and well-behaved
- It's linear
  - Each subdivided point is a linear combination of points within the 2 poly ring.
  - I.e. current poly and one ring around it.

Semisharp creases



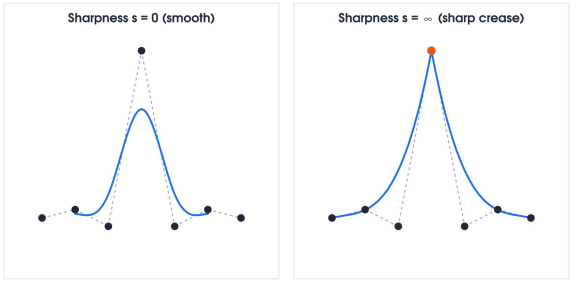
- Vanilla Catmull-Clark struggles with sharp bevels
- Semisharp weights: sharp rule for N levels, then blend to smooth
- Widely adopted - [DeRose et al. 1998]

Regular quads = bicubic B-splines



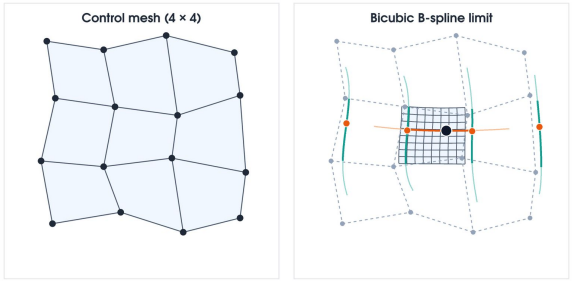
- A regular 4×4 control mesh
  - Evaluate the surface analytically at any (u,v)
  - Both position and normal
- The 2D bicubic basis is separable
  - The 1D cubic basis applied twice
- Evaluate the 4 columns in v, then a single curve in u through the results
- No recursion needed!
- Only works at regular polys

Semisharp creases stay analytic



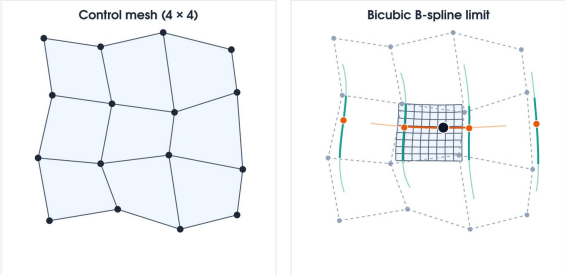
- Take that same 1D cubic basis
  - A crease just changes the basis [Nießner et al. 2012b]
- One 4×4 matrix on the control points, derived from semisharp weight
- Multiple sharp edges, or a sharp corner?
  - Treat that neighbourhood as irregular

Creases live on the x = 0 line



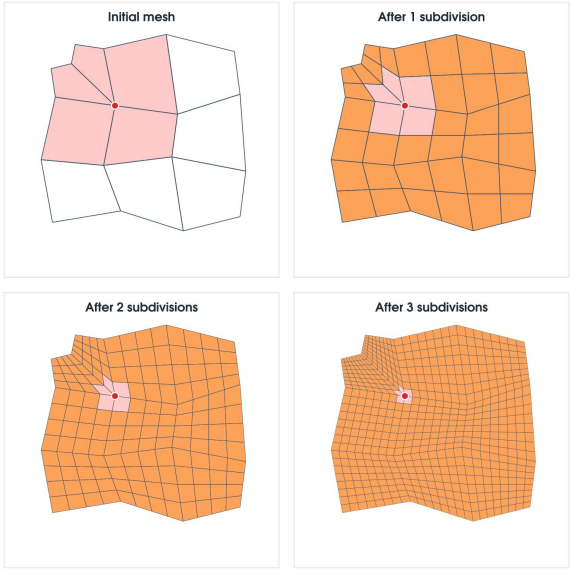
- Same separable evaluation: 4 splines in v, then a single spline in u
- The x=0 line is the one place we allow a semisharp crease
- A crease there is just a change of basis on the 1D cubic [Nießner et al. 2012b]
- Everywhere else stays a plain bicubic

Regular with 1 optional semisharp crease

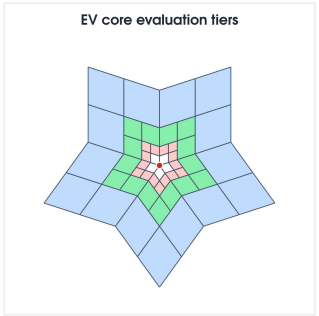


- This is the primary building block of this algorithm.
- Problem:
  - Decompose the full subdivision surfaces into regular grids (with up to 1 semisharp crease)
  - Special handling for rare cases where we are off a regular grid.

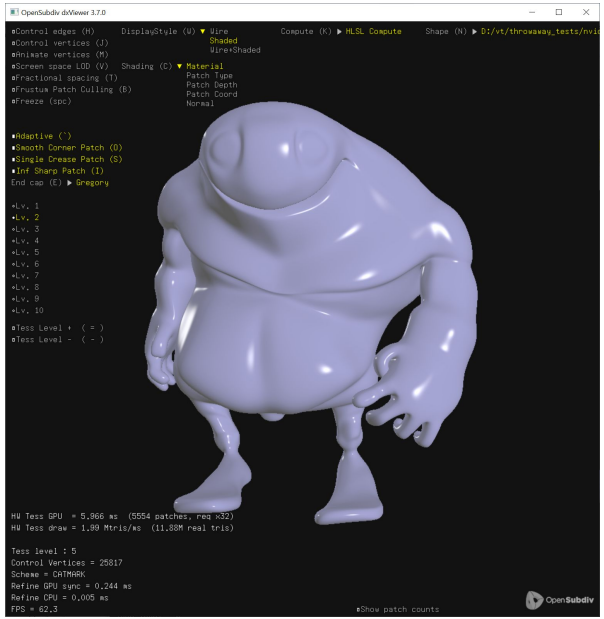
Feature-adaptive subdivision



- Only quads touching an irregular vertex stay irregular - [Nießner et al. 2012a]
  - 2012 was a pretty good year for Catmull-Clark subdivision
- New quads are regular by construction
- Irregular count grows linearly per level (much better than 4× expansion)

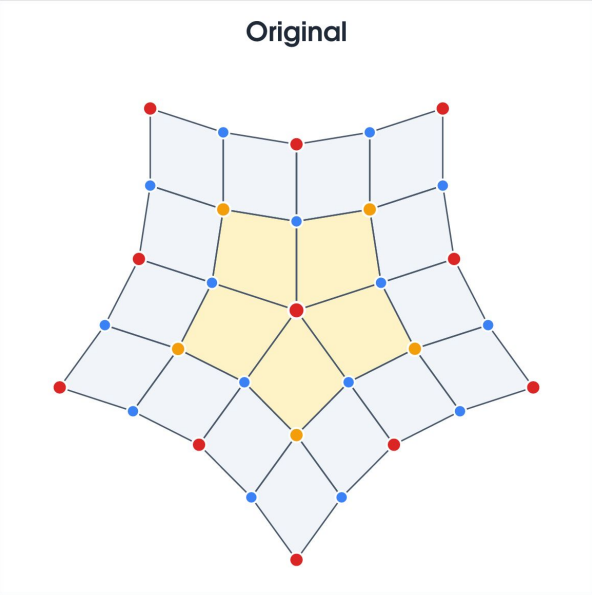


Feature-adaptive subdivision with DX11 Tessellation



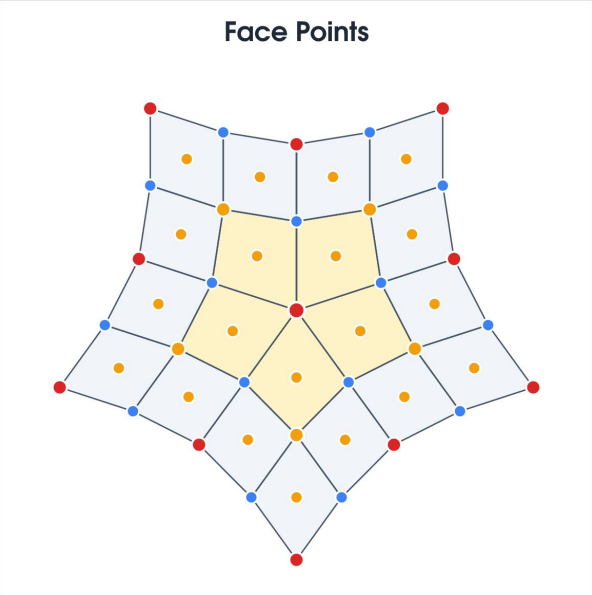
- Store the patches as a quadtree [Brainerd et al. 2016]
  - Deformation via stencil tables [OpenSubdiv]
- Decompose problem into two steps:
  - 1. Tessellate (DX11 Hardware)
  - 2. Evaluate tessellated points (Domain Shader)
- Use compute to generate patch data after deformation.
- Evaluate position/normal in the Domain Shader
- Large tables

Catmull-Clark: the original cage



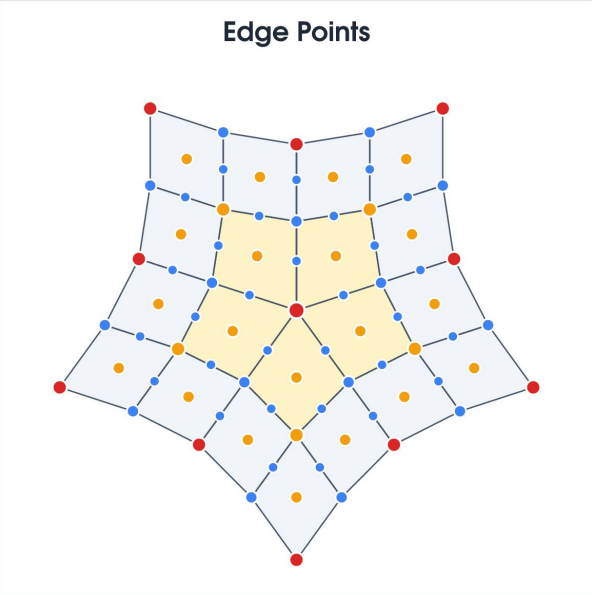
- Start with the control cage
  - An irregular vertex with five quads around it
- Everything that follows is one subdivision step, in four stages

Catmull-Clark: face points



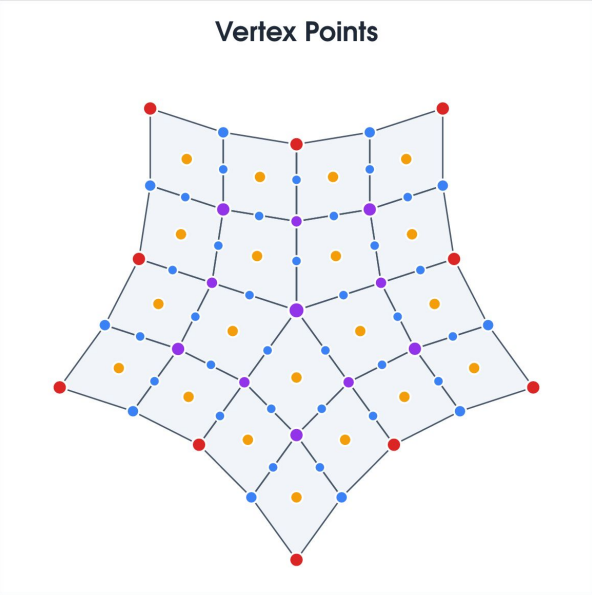
- One new face point per face
  - The average of its corners

Catmull-Clark: edge points



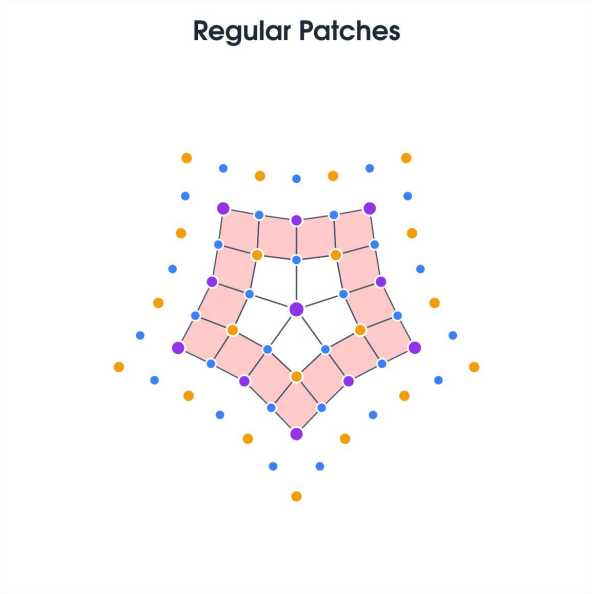
- One new edge point per edge
  - Averaging the edge with its two new face points

Catmull-Clark: vertex points



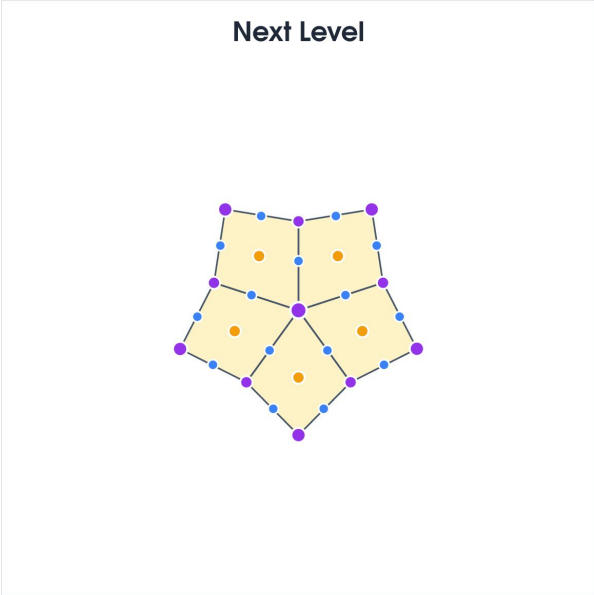
- Finally, move the original vertices
  - Using the surrounding face + edge points
- Self-contained recursive formula

Recursive Regular Patches



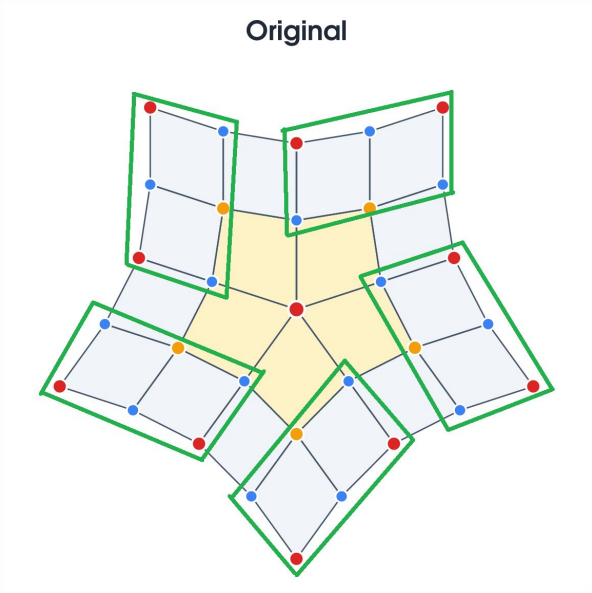
- Each original poly is split into 4
  - 3 polys are now regular
  - 1 poly is still irregular

...and the inner core recurses



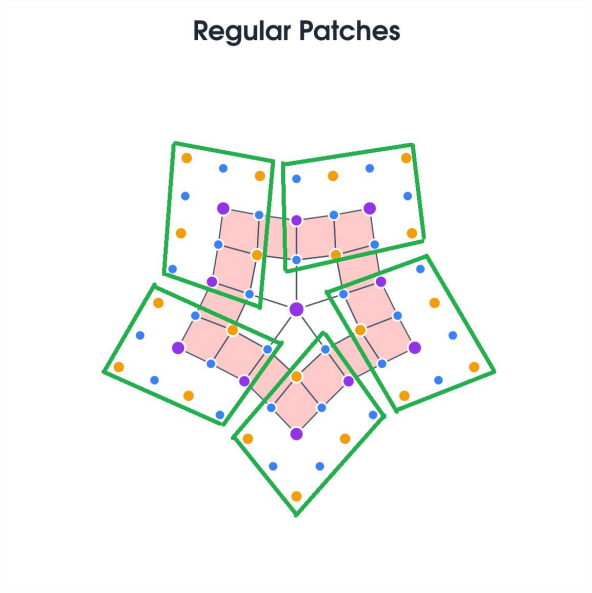
- The inner points are the exact same topology as the initial topology.
- The irregular part shrinks each level
  - Irregular exponentially smaller
  - Continue until irregular region is "small enough"
  - Use linear interpolation in that region
- Self-contained
  - Does not require any other vertices

Our idea: recompute patches in compute shader



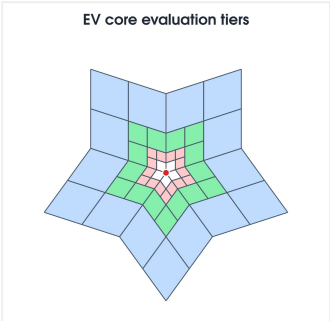
- One thread per poly.
  - 5 polys -> 5 threads
  - 6 initial points per thread
- Group multiple irregular verts in the same group (up to 64 thread workgroup)
- Ping pong back and forth in LDS

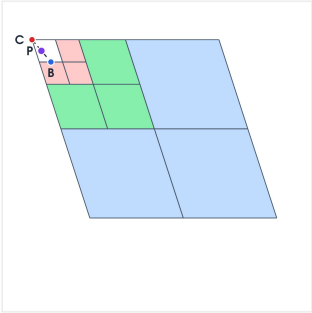
Recompute patches in compute shader



- For each subdivision level
  - Calculate face points. GroupSync.
  - Calculate edge points. GroupSync.
  - Calculate vertex points. GroupSync.
  - Calculate center point. GroupSync.
  - Write Point Data.
- Each iteration:
  - Starts with 6 points per thread
  - Writes 12 points per thread

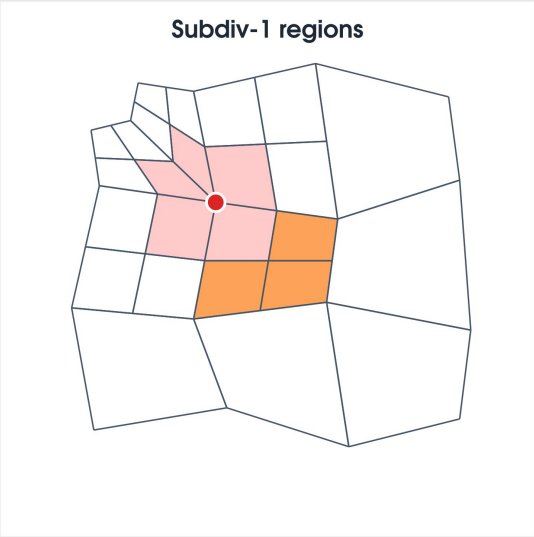
The white patch and limit points





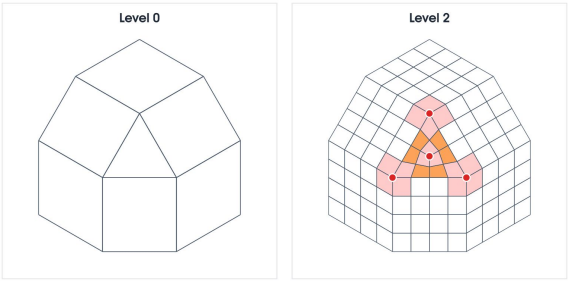
- The irregular point CS generates a set of "slices"
- To evaluate a final point:
  - Determine regular grid and evaluate
- If we are in the small irregular region:
  - Evaluate nearest point on regular patch and interpolate.
- Only the original line can be semisharp
  - Rotate

Irregular Points



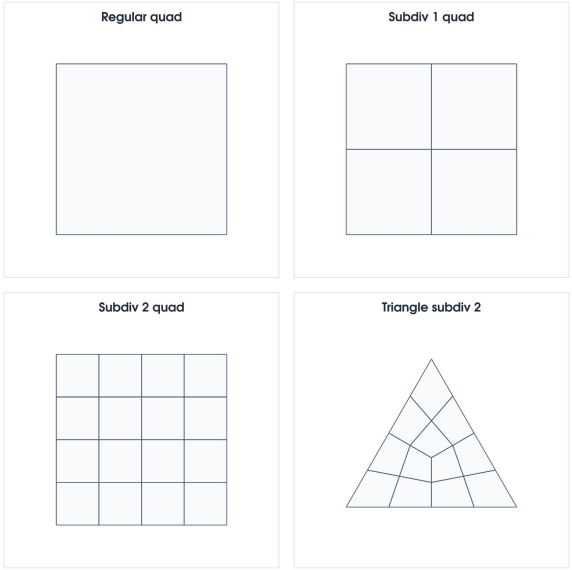
- For any quads touching an irregular point: subdivide once
- Results in an irregular fan, with regular polys elsewhere.

Triangles



- Triangles and all neighbors are subdivided twice
  - The middle point of the triangle is irregular.
  - The triangle corner points may or may not be irregular, but need to reference the twice-subdivided adjacent polys

Regions and primitives (no quadtree)



- Regular Quads: 1 region
- Quads with at least one irregular point (but no triangles): Subdiv 1 (4 regions)
- Quads touching at least one triangle: Subdiv 2 (16 regions)
- Triangles: Always subdivide topology twice (12 regions)

Evaluating a vertex - CS

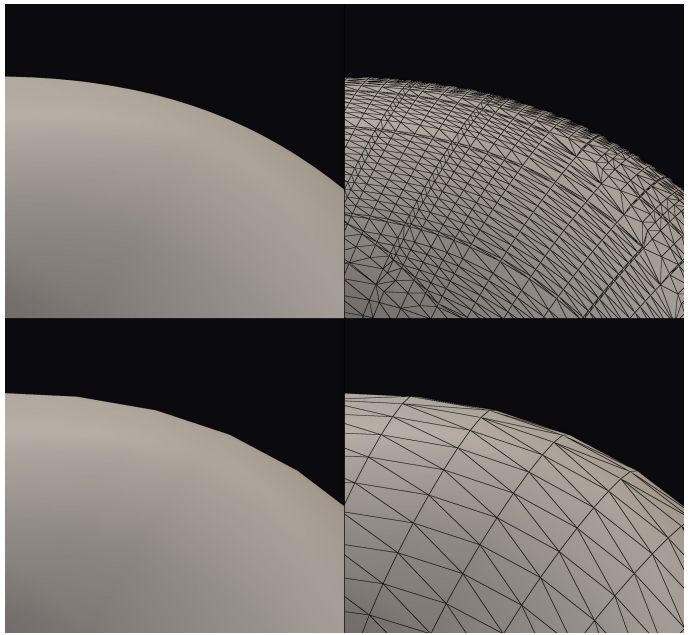
- Input: Triangle ID and barycentric
- Output: Limit position and normal
- Triangles remap into sub-quads first, so everything downstream is a quad
- Regular region: 16 control points, bicubic; normal is analytic
- Irregular region:
  - Find the ring for this point around the extraordinary vertex
  - Fetch 16 points for that ring, apply the crease, evaluate
  - Blend to the center limit if in the unknown region
- Blend against a flat baseline by tess factor: low tess stays flat, so nothing pops
- Displace in the same shader. Write to position and normal.

Results: adaptive vs fixed

- Monkey: 507 source verts / 968 source tris
- Solid shading is visually indistinguishable with fewer triangles.
- Fixed Tris: 61,952
- Adaptive Tris: 30,526

Full head	Fixed	Adaptive Curvature	vs Fixed
Output verts	43,560	24,622	57%
Output tris	61,952	30,526	49%

Results: adaptive vs fixed



- Push the camera in: most of the mesh is now off-screen or behind the camera
- Adaptive pulls density off what you can't see and spends it on the silhouette
- More silhouette density than fixed
  - Using less than half the triangles
- Artists good at authoring for typical size on screen
  - But there are always exceptions

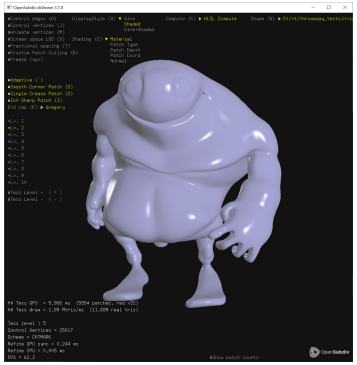
Silhouette zoom	Fixed	Adaptive Curvature	vs Fixed
Output verts	43,560	19,818	46%
Output tris	61,952	29,918	48%

Performance

- Compute cost only (RTX 3070)
- Subdivision: builds the control points.
- Tessellation: builds topology, vertices (position, normal, barycentric) and welds.
- imrod is 5x the subdiv cost due to reliance on triangles.

Mesh	Output tris	Subdivision	Tessellation	Sub + Tess
suzanne	15.9M	0.037 ms	3.368 ms	3.405 ms
bigguy	11.9M	0.025 ms	2.171 ms	2.196 ms
monsterfrog	10.6M	0.028 ms	2.014 ms	2.042 ms
imrod	14.2M	0.139 ms	3.243 ms	3.382 ms

Comparison: OpenSubdiv (hardware tessellation)



Stage (RTX 3070)	Big Guy	Monster Frog
Triangles	11.9 M	10.6 M
Adaptive Compute — subdiv + tess (once)	2.16 ms	2.03 ms
Adaptive Compute — lit render	2.16 ms	1.90 ms
Adaptive Compute — shadow pass (avg ×4)	1.36 ms	1.26 ms
OpenSubdiv — per pass (points + render)	5.97 ms	5.40 ms

- Same models, same GPU (RTX 3070)
- OpenSubdiv:
  - Short compute pass to evaluate control points.
  - Fused DX11 Tessellation pass which evaluates and renders the mesh. [OpenSubdiv]
  - Multiple passes (GBuffer, shadow, etc.) would each require a full Catmull-Clark evaluation.
- Adaptive Compute:
  - Builds the geometry once (subdiv+tess)
  - Each pass rasterizes the stored buffer, interpolating in VS

Cost per pass — ms per 1M triangles

- Adaptive Compute tessellates once
  - OpenSubdiv tessellates every pass
- Single Lit Pass (per 1M tris)
  - 0.36ms vs 0.50ms
- Lit and 4 Shadows: (per 1M tris)
  - 0.81ms vs 2.51ms (~3x)

Per 1M tris	Adaptive · Big Guy	Adaptive · Frog	OpenSubdiv · Big Guy	OpenSubdiv · Frog
Fixed compute (once / frame)	0.18	0.19	0	0
First (lit) render	0.18	0.18	0.50	0.51
Each additional shadow pass	0.11	0.12	0.50 *	0.51 *
1 lit	0.36	0.37	0.50	0.51
1 lit + 4 shadow	0.82	0.85	2.51	2.57

Compute cost across multiple passes

- Timing on a RTX 3070
- Building the indices/positions/normals/barycentrics
  - Not including render time.
- Calculating control points is light: 0.03–0.14 ms.
  - Can perform once if mesh isn't deforming
- imrod's is higher because it has more triangles, which require more control points.

Mesh	Output tris	Subdivision	Tessellation	Sub + Tess
suzanne	15.9M	0.037 ms	3.368 ms	3.405 ms
bigguy	11.9M	0.025 ms	2.171 ms	2.196 ms
monsterfrog	10.6M	0.028 ms	2.014 ms	2.042 ms
imrod	14.2M	0.139 ms	3.243 ms	3.382 ms

Comparison: Edge-Friend

- Uniform Catmull-Clark evaluator. [Kuth et al. 2023]
- Topology is implicit.
- Uses mesh and amplification shaders.
- Normals are computed in mesh shaders.
- Used an implementation by Wei Liao
  - github.com/Flurry-L/Edge-Friend-Loader
- Measured the raw position calculation time but not rendering time (not implemented).

Comparison: DV21

- The halfedge refinement rule, also uniform (not adaptive) [Dupuy & Vanhoey 2021]
- Also noteworthy is [Dupuy & Deliot 2022], which performs Catmull-Clark using DV21.
  - It applies a custom topology to dynamically tessellate the DV21 output.
  - Since it requires DV21 as a pre-pass, the DV21 numbers suffice as a lower-bound on the cost.

Comparison: Uniform

Mesh	Edge-Friend	This work (adaptive)	DV21
Suzanne	6.69M	4.67M	2.22M
Bigguy	8.03M	5.42M	2.21M
Monsterfrog	6.87M	5.19M	2.24M
Imrod	7.05M	4.20M	2.20M
average	~7.15M	~4.87M	~2.22M

- Units: Millions of output triangles per millisecond.
- Edge-Friend and DV21 are uniform Catmull-Clark, position only
- Ours is performing the full adaptive pipeline
  - Calculating positions, normals, and generating topology
  - Locking the tess-factors to a fixed value
- Edge-Friend: Faster than ours by 1.32x - 1.68x.
- DV21: Slower than ours by 1.91x - 2.45x.
- Also tested NVIDIA RTX Mega Geometry SDK.
  - Not enough RAM (10GB minimum)

Try it!

- Live Subdiv WebGPU demo:
  - https://filmicworlds.com/blog/adaptive-catmull-clark-subdivision-with-compute-tessellation/
- Full explanation of tessellation pattern:
  - https://filmicworlds.com/blog/compute-tessellation-with-clamped-parallelograms/
- All code is MIT licensed
- Thanks: Natalya Tatarchuk and Gregory Mitrano

Next Steps

- Visibility Buffer
- WebGPU limitations
  - Each indirect pass requires a compute validation pass with a barrier.
  - If you render 10 indirect compute passes, then you get 10 extra validation passes.
    - Can we get a hint to batch WebGPU indirect dispatch validation?
  - Very strict execution model.
    - Batching point with the on the same bicubic b-spline patch
  - No shader intrinsics: Self-imposed limitation
- Mesh formats
  - Usd, FBX are battle tested. Obj has extensions.
  - GLTF has no support for either quads or semisharp creases

References

- [Catmull & Clark] Catmull & Clark, “Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes.” Computer-Aided Design, 1978.
- [Cook et al. 1987] Cook, Carpenter & Catmull. "The Reyes Image Rendering Architecture." SIGGRAPH 1987.
- [DeRose et al. 1998] DeRose, Kass & Truong. "Subdivision Surfaces in Character Animation." SIGGRAPH 1998.
- [Rossignac 1999] Rossignac. "Edgebreaker: Connectivity Compression for Triangle Meshes." IEEE TVCG 5(1).
- [Boubekeur & Alexa 2008] Boubekeur & Alexa. "Phong Tessellation." SIGGRAPH Asia 2008.
- [Giesen 2011] Giesen. "A Trip Through the Graphics Pipeline 2011, Part 12."
- [Nießner et al. 2012a] Nießner, Loop, Meyer & DeRose. "Feature-Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces." ACM TOG 31(1).
- [Nießner et al. 2012b] Nießner, Loop & Greiner. "Efficient Evaluation of Semi-Smooth Creases in Catmull-Clark Subdivision Surfaces." Eurographics 2012.
- [Brainerd 2014] Brainerd. "Tessellation in Call of Duty: Ghosts." SIGGRAPH 2014, Advances in Real-Time Rendering in Games - advances.realtimerendering.com/s2014
- [Brainerd et al. 2016] Brainerd, Foley, Kraemer, Moreton & Nießner. "Efficient GPU Rendering of Subdivision Surfaces using Adaptive Quadtrees." ACM TOG 35(4).
- [Dupuy 2020] Dupuy. "Concurrent Binary Trees (with application to longest edge bisection)."
- [Deliot et al. 2021] Deliot, Dupuy, Rijnen & Yao. "Experimenting with Concurrent Binary Trees for Large-Scale Terrain Rendering." SIGGRAPH 2021, Advances in Real-Time Rendering in Games.

References (cont.)

- [Dupuy & Vanhoey 2021] Dupuy & Vanhoey. "A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision." HPG 2021. ("DV21")
- [Karis et al. 2021] Karis, Stubbe & Wihlidal. "A Deep Dive into Nanite Virtualized Geometry." SIGGRAPH 2021, Advances in Real-Time Rendering in Games - advances.realtimerendering.com/s2021
- [Dupuy & Deliot 2022] Dupuy & Deliot. "Bisection Based Triangulation of Catmull-Clark Subdivision." SIGGRAPH 2022 Courses, Advances in Real-Time Rendering in Games - advances.realtimerendering.com/s2022
- [Kuth et al. 2023] Kuth, Oberberger, Chajdas, Meyer. "Edge-Friend: Fast and Deterministic Catmull–Clark Subdivision Surfaces." CGF 42(8).
- [Karis 2026] Karis. "How to Tessellate." (Nanite tessellation)
- [Barczak 2026] Barczak. "Introducing AMD DGF SuperCompression." AMD GPUOpen.
- [OpenSubdiv] Pixar. OpenSubdiv. opensubdiv.com
- [Draco] Google. Draco 3D Data Compression. github.com/google/draco
- [Microsoft] "Tessellation Stages." Direct3D 11 documentation.
- [Cantlay] Cantlay. "DirectX 11 Terrain Tessellation." NVIDIA.
- [Reed] Reed. "Tessellation Modes Quick Reference."