Today

- Key per-primitive operations (clipping, culling)
  Various slides credit John Owens, Kurt Akeley, and Pat Hanrahan

- Programmable primitive generation
  - Geometry shader
  - Modern GPU tessellation
Recall: in a modern graphics pipeline, application-specified logic computes vertex positions

vertex positions emitted by vertex processing (or the geometry shader, if enabled) are represented in homogeneous clip-space coordinates.

Vertex is within the view frustum if:

\[-w \leq x \leq w\]
\[-w \leq y \leq w\]
\[-w \leq z \leq w\]

Vertex’s position in euclidian space is \((x/w, y/w, z/w)\)
Per primitive operations

Assemble vertices into primitives
Clip primitive against view frustum
For each resulting primitive
   Divide by w
   Apply viewport transform
Discard back-facing primitives [optional, depends on config]
Assembling vertices into primitives

How to assemble is part of graphics state (specified by draw command)

Notice: independent vertices get grouped into primitives (dependency!)
Clipping

- May generate new vertices/primitives, or eliminate vertices/primitives
- Data-dependent computation
  - variable amount of work per primitive
  - variable control flow per primitive
Why clipping?

- Avoid downstream processing that will not contribute to image (rasterization, fragment processing)

- Establish invariants for emitted primitives
  - Can safely divide by w after clipping
  - Bounds on vertex positions (can now choose precision of subsequent operations accordingly)
Guard-band clipping

- Reduces variance in per-primitive clipping work
- Cost (conservative: primitives no longer guaranteed to be fully on screen)
  - Rasterizer must not generate off-screen fragments
  - Increased precision needed during rasterization
### Back-face culling

- Use sign of triangle area to determine if triangle is facing toward or away from camera
- May discard primitive as a result of this test
  - For closed meshes, eliminates $\sim 1/2$ of triangles
    (these triangles will be occluded anyway)

The triangle area is given by:

$$\text{Triangle area} = \frac{(x_0y_1 - x_1y_0) + (x_1y_2 - x_2y_1) + (x_2y_0 - x_0y_2)}{2}$$
- Divide triangle strips from application into small strips, round robin to geometry engines
- Buffers absorb variance in amount of work per triangle

[Image of SGI Reality Engine 1992 diagram]
Programmable geometry amplification

- Amplification by “geometry shader” or tessellation functionality in a modern pipeline is far greater than that of clipping

- Geometry shader: output up to 1024 floats worth of vertices per input primitive

- Tessellation: thousands of vertices from a base primitive
Thought experiment

Assume maximum amplification factor is large (known statically)

Simple approach 1: make on-chip buffers as big as possible: run fast for low amplification

Simple approach 2: make huge FIFOs (store off-chip in memory)
Modern GPU tessellation [Moreton 01]

- **Motivations:**
  - Reduce CPU-GPU bandwidth
  - Animate/skin course resolution mesh, but render high resolution mesh

- Requires parametric surfaces (must support direct evaluation)

Note: D3D11 Stage Naming (not canonical stage names)
Parametric surface

\[(u,v)\]

\[f(u,v) = \langle x, y, z \rangle\]
Parametric surfaces: common examples

Bicubic patch, 16 control points (quad domain)

PN Triangles, 3 vertices + 3 normals (defines bezier patch on triangular domain)

See “Approximating Catmull-Clark Subdivision Surfaces With Bicubic Patches”, Loop et al. 2008

See “Curves PN Triangles”, Vlachos et al. 2008
Modern GPU tessellation

- **Hull shader**
  - Accepts primitives after traditional vertex processing
  - Computes tessellation factor along each domain edge
  - Computes control points for parametric surface (from primitive vertices)

Note: D3D11 Stage Naming (not canonical stage names)
Hull shader produces edge tessellation rates

Based on estimate of parametric surface position

(Note: rates need not be integral)
Fixed-function tessellation stage

Input: edge tessellation constraints for a patch
Output: (almost) uniform mesh topology meeting constraints
Domain shader stage

Input: control points (from hull shader) and stream of parametric vertex locations \((u,v)\) from tessellator

Output: position of vertex at parametric coordinate: \(f(u,v)\)
Modern GPU tessellation

- **Heterogeneous implementation**
- **Hull shader**
  - Original primitive granularity
  - Data-parallel
  - Large working set (typically a primitive + one-ring)
- **Tessellator**
  - Surface agnostic, fixed-function hardware implementation
  - Irregular control flow
- **Domain shader**
  - Fine-mesh-vertex granularity
  - Data-parallel (preserves shader programming model)
  - Direct evaluation of surface (extra math, but data-parallel)
Challenge: avoid cracks!
Modern GPU tessellation summary

- Heterogeneous, 3-stage implementation
  - Algorithms co-designed with pipeline abstractions and hardware

- Enables adaptive level-of-detail, high-resolution meshes in games

- Challenges
  - Application developer: avoiding cracks (requires consistent edge rate evaluation -- this is tricky in floating point math)
  - GPU implementor: managing large data amplification... while maintaining parallelism, locality, and order