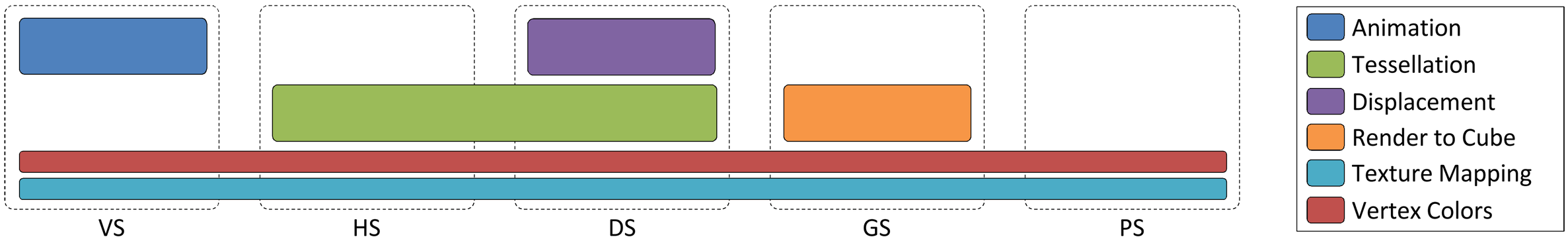
Lecture 7:

Geometry

EEC 277, Graphics Architecture
John Owens, UC Davis, Winter 2017

Tue Guest Lecture: Tim Foley, NVIDIA Research



Project Proposals

- Preproposal Tue 14 Feb, proposal Tue 21 Feb—important!
- Single largest cause of problems in this class
- What I want to see:
- Understanding of big-picture problem
  - With references as appropriate [I can help here]
- What you propose to do to help solve it
  - Small & complete beats big and vague
  - Convince me that your proposal is important
- How you will do that
  - Procedure
  - Schedule with milestones
  - Convince me that your proposal will work

Recent (graphics-related) class projects

- Reyes Rendering on a Graphics Processor
- Ray casting parallelization with CUDA
- Real-time Ray Tracing using Graphical Processing Unit
- Case study on smartphone GPUs and how different implementations affect the performance of a 3D graphics application
- GPU-accelerated mesh deformation
- Texturing in OpenGL versus Reyes Architectures
- Niko: another abstraction of software programmable pipeline

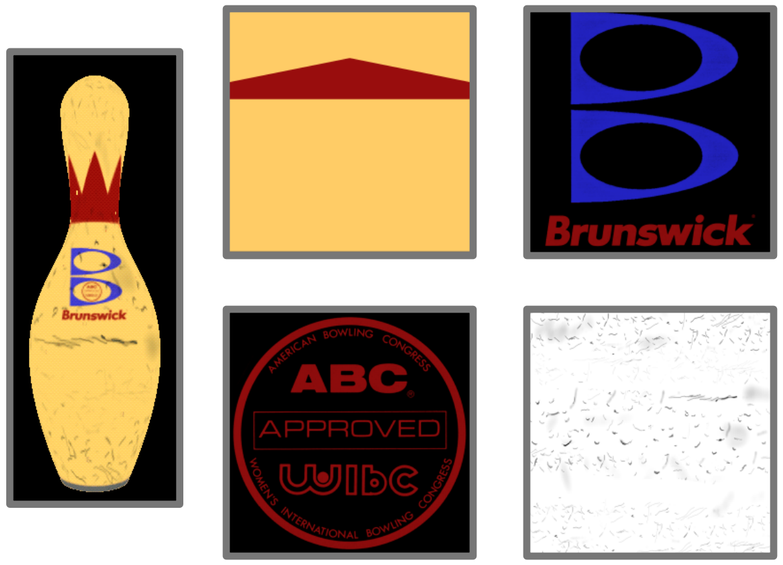
- Suggestion (ambitious): Build OpenGL in CUDA/OpenCL

Trends / Responses

- More transistors / higher clock speeds
  - Faster processing
  - More features
    - New graphics features
    - Features to aid optimization of computation
    - Features easing burden on programmer
    - Higher quality
    - Programmability
- Ratio between off-chip communication and computation changing
- On-chip communication getting more costly

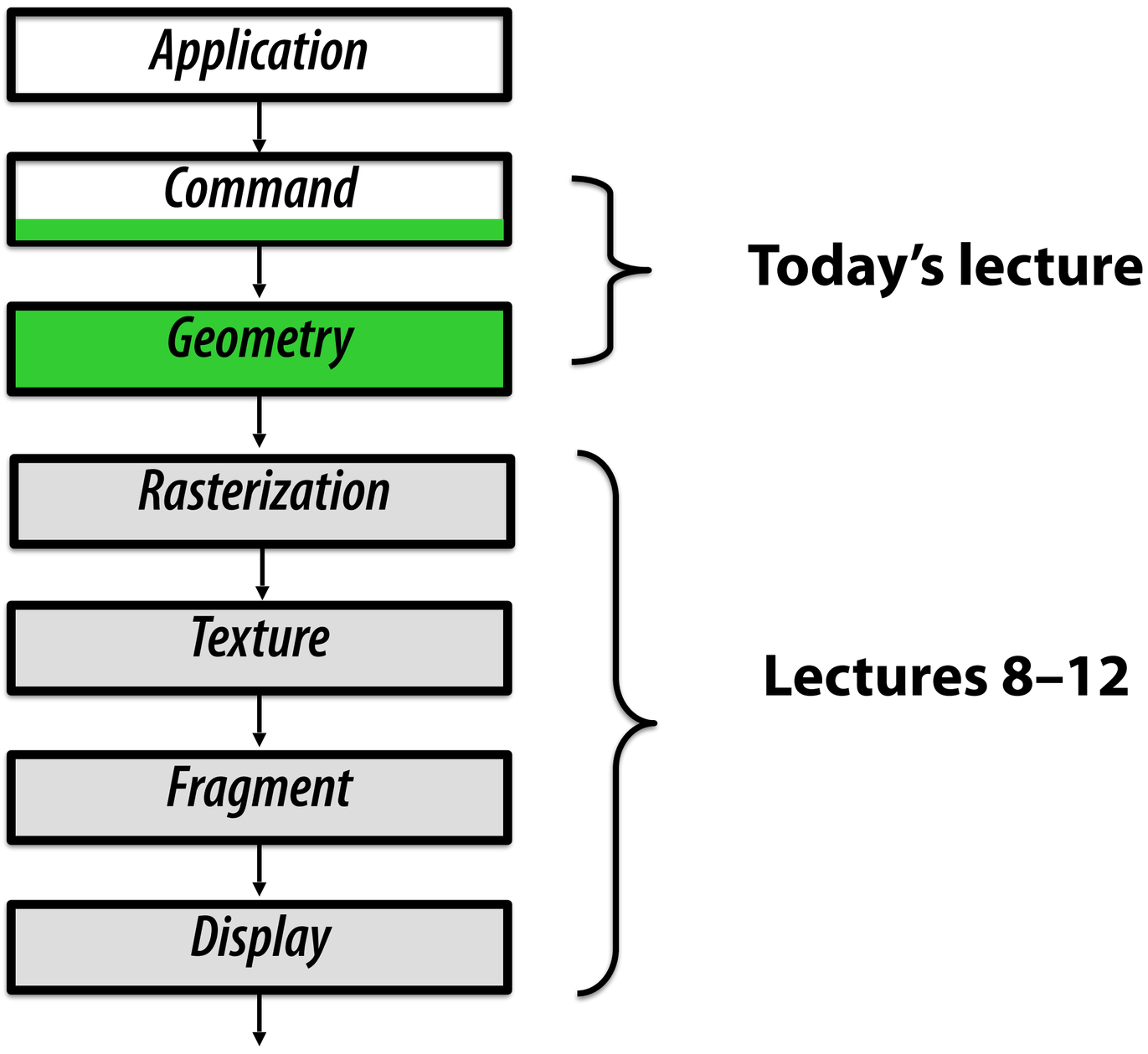
Trends / Responses

- More transistors / higher clock speeds
- Ratio between off-chip communication and computation changing
  - Reduce bandwidth from off-chip
    - Vertex (input) bandwidth
    - Texture bandwidth
    - Bandwidth to framebuffer
  - Compute rather than communicate
  - Patterson: caching, replication, prediction
- On-chip communication getting more costly



Trends / Responses

- More transistors / higher clock speeds
- Ratio between off-chip communication and computation changing
- On-chip communication getting more costly
  - Reduce global on-chip communication
  - Replicate computation rather than communicate
  - Power considerations becoming increasingly important


Application
Command
Geometry
Rasterization
Texture
Fragment
Display
Today's lecture
Lectures 8–12

Trends / Responses

- More transistors / higher clock speeds
  - Faster processing
  - More features
    - Graphics features
    - Features easing burden on programmer
    - Higher quality
- Ratio between off-chip communication and computation changing
  - Reduce bandwidth from off-chip
    - Vertex (input) bandwidth
    - Texture bandwidth
    - Bandwidth to framebuffer
  - Compute rather than load
- On-chip communication getting more costly
  - Reduce global on-chip communication
  - Replicate computation rather than communicate

Theme of graphics API implementation

- Smart techniques to reduce compute
- Perhaps less relevant today because compute is so cheap
  - You will frequently hear me say "Historically, this operation was expensive, so we try to minimize the number of times we do it".
- (Also smart techniques to reduce bandwidth)

Homogeneous Coordinates

- Points and directions can be represented in homogeneous coordinates
- Think of homogeneous *w* as scaling term
- Why?
  - Allows more general xforms: 4-vector homogeneous transforms are much more general than 3-vector
  - Lazy evaluation: don't do divide until we need it

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \Leftrightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

About Matrices

- 4x4 matrix, 4-component coordinate
- Single-precision floating point is typical
- Matrices can be composed
  - By the application
  - By the implementation
- Affine (3x3)—preserves parallelism of lines
  - Translate, rotate, orthographic
- Perspective projection—not affine

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{42} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Homogeneous Transforms

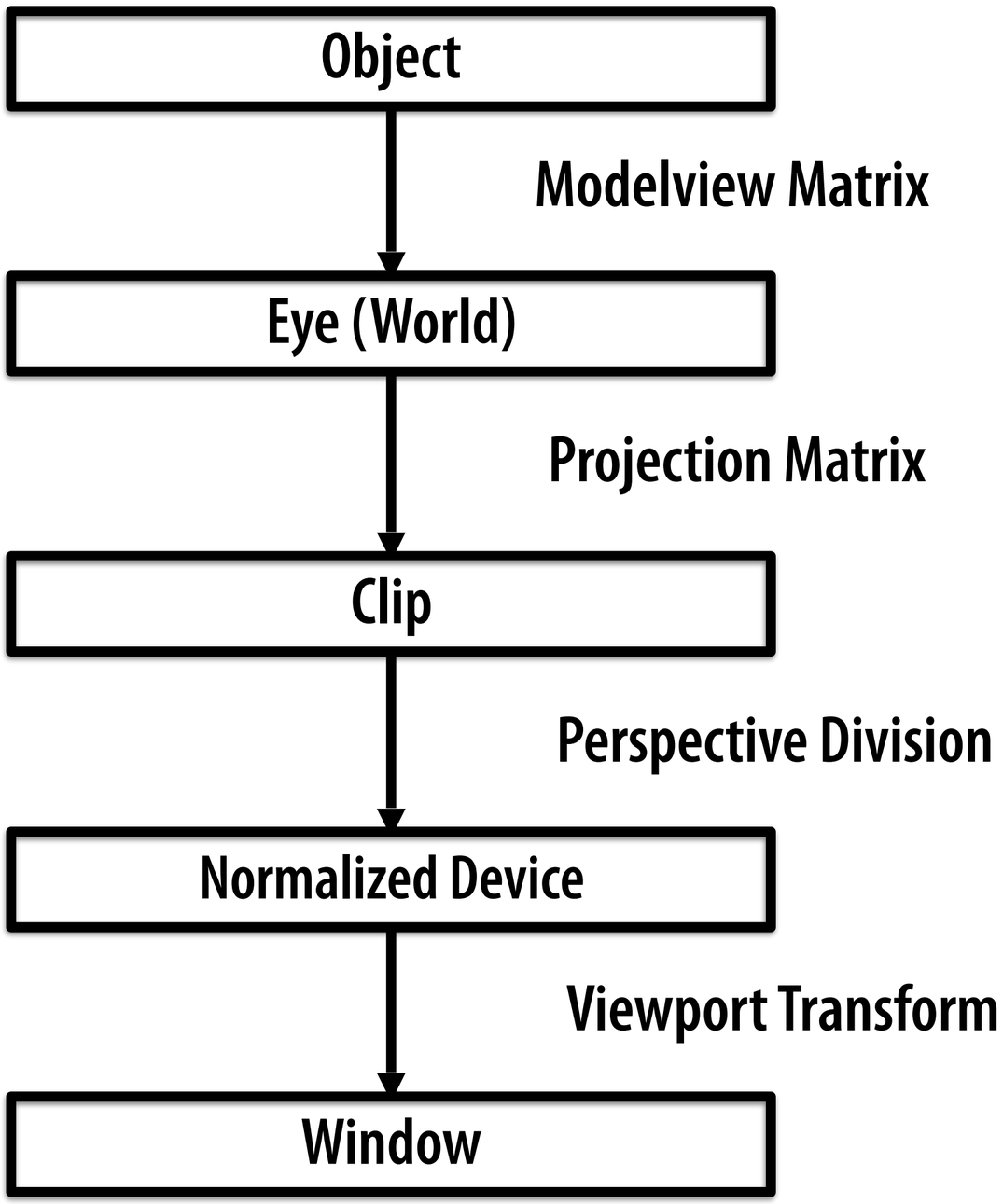
■ glFrustum

$$\begin{bmatrix} \frac{2 \cdot zNear}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2 \cdot zNear}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & -\frac{2 \cdot zFar \cdot zNear}{zFar-zNear} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

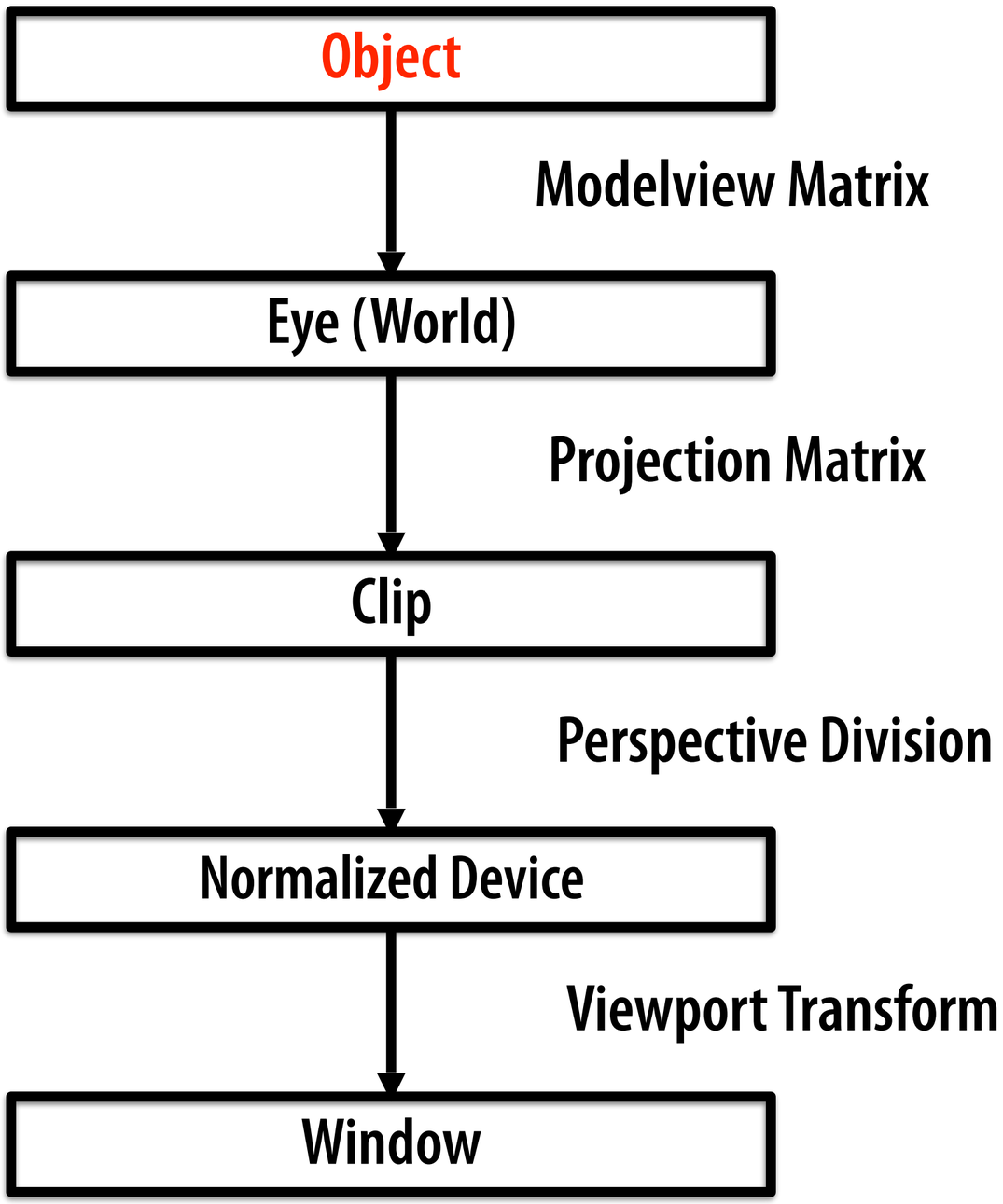
■ gluPerspective

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2 \cdot zFar \cdot zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Coordinate System Transforms

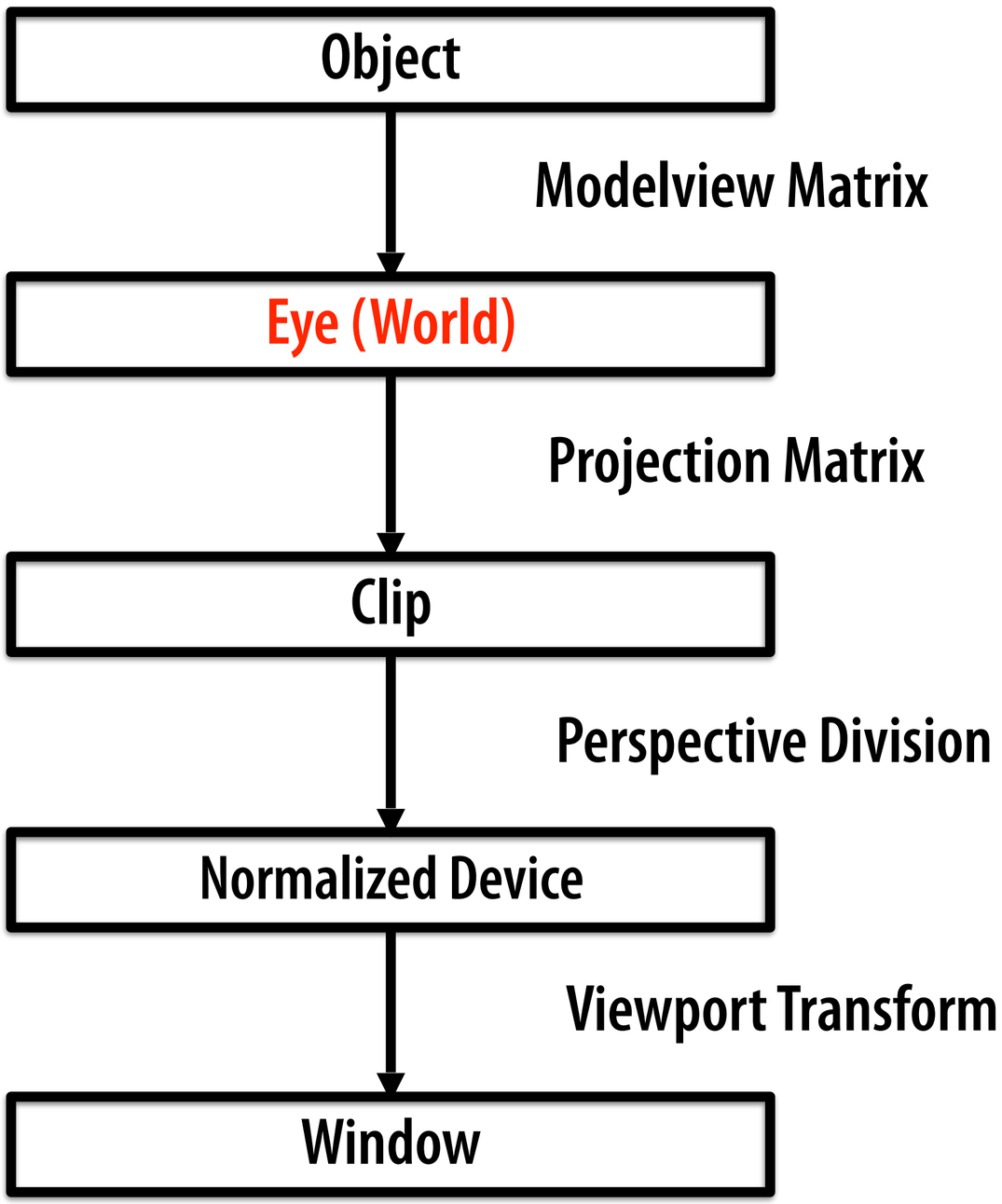


Coordinate System Transforms



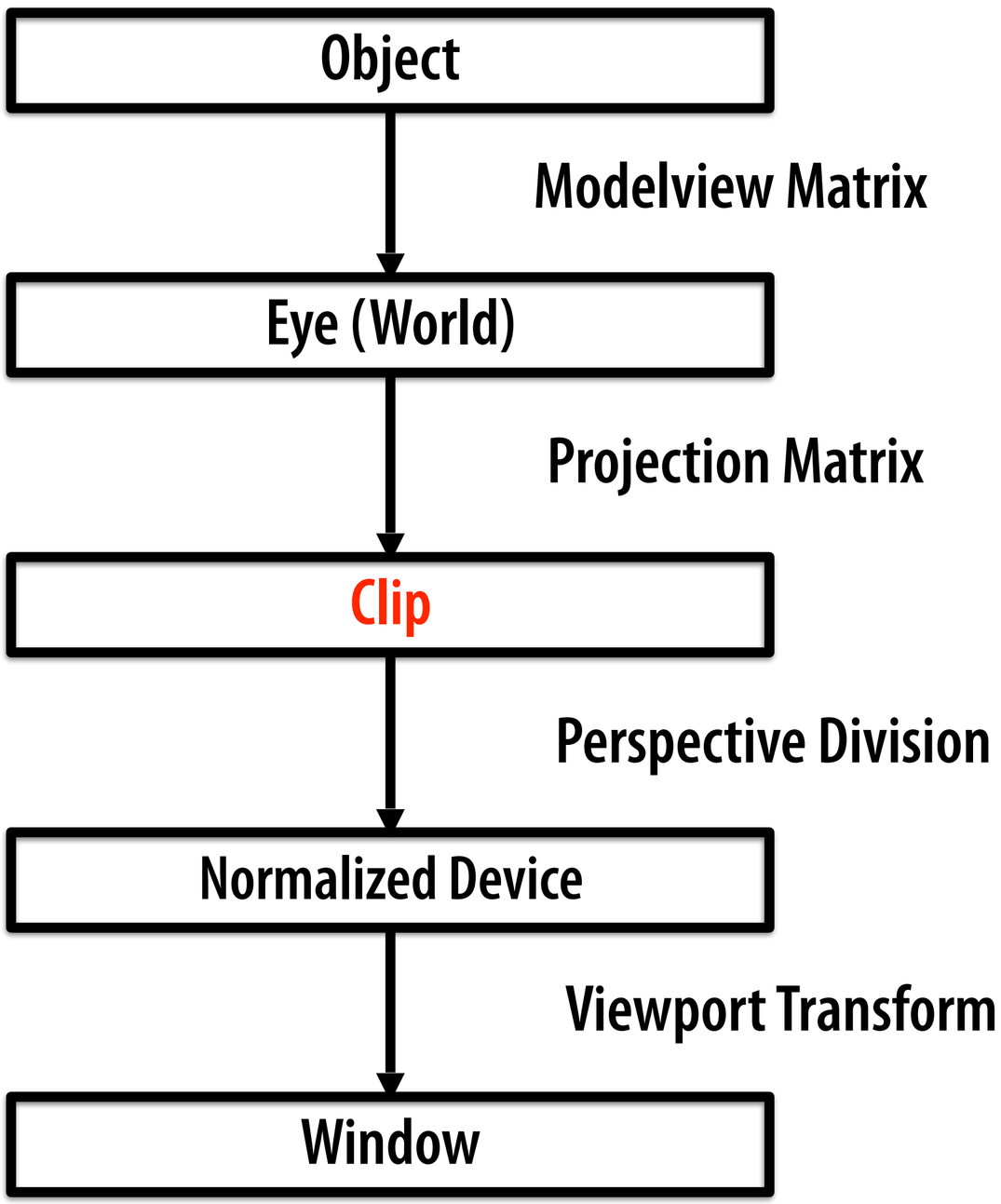
- Object space—natural coordinate system of models. Why? Easiest to model in this space.

Coordinate System Transforms



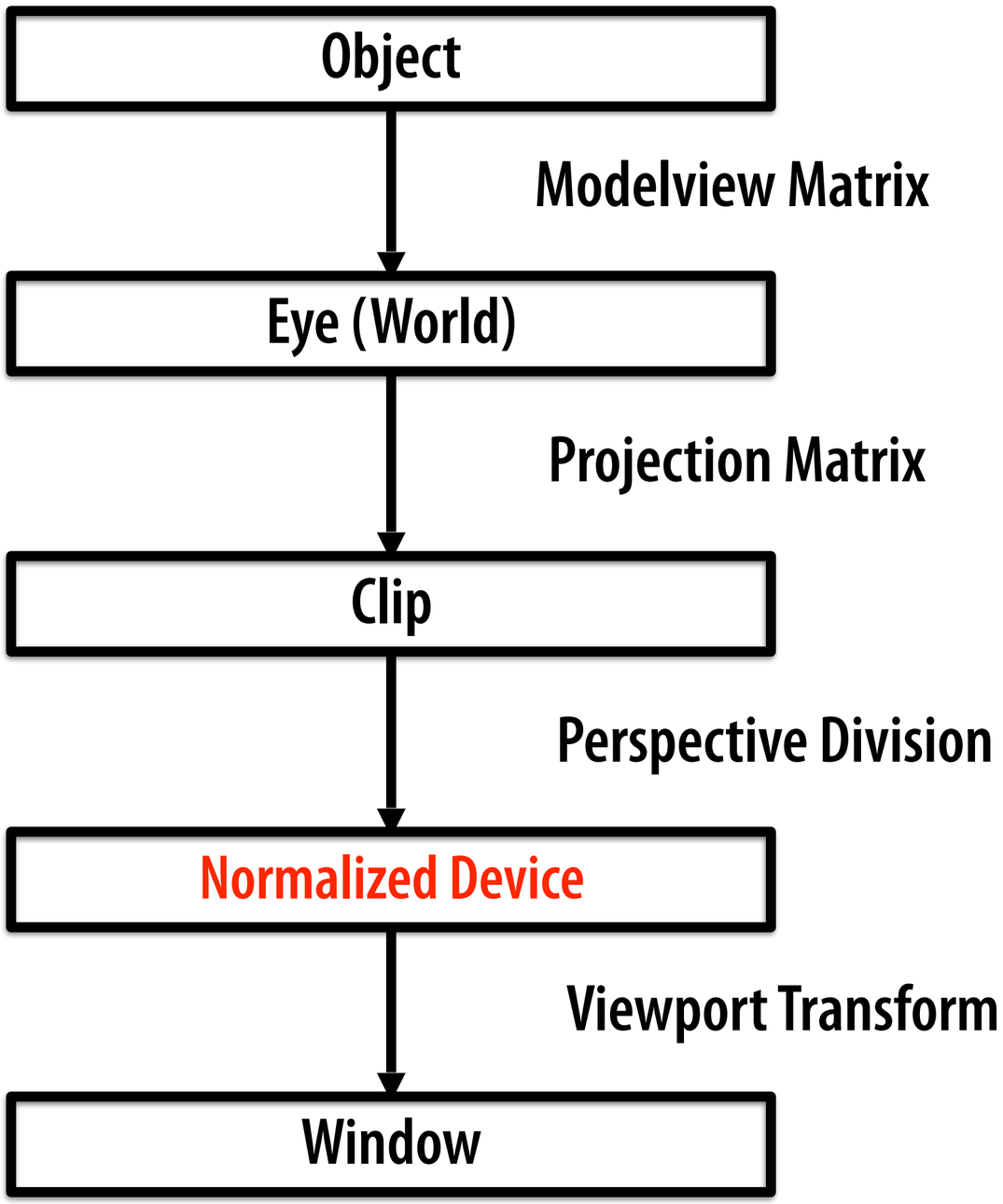
- Eye space—common space for all objects
- Why? Most suitable for lighting calculations
  - This is less used today (modern renderers back-project lights into object space and calculate there)
- User-specified clipping planes applied here

Coordinate System Transforms



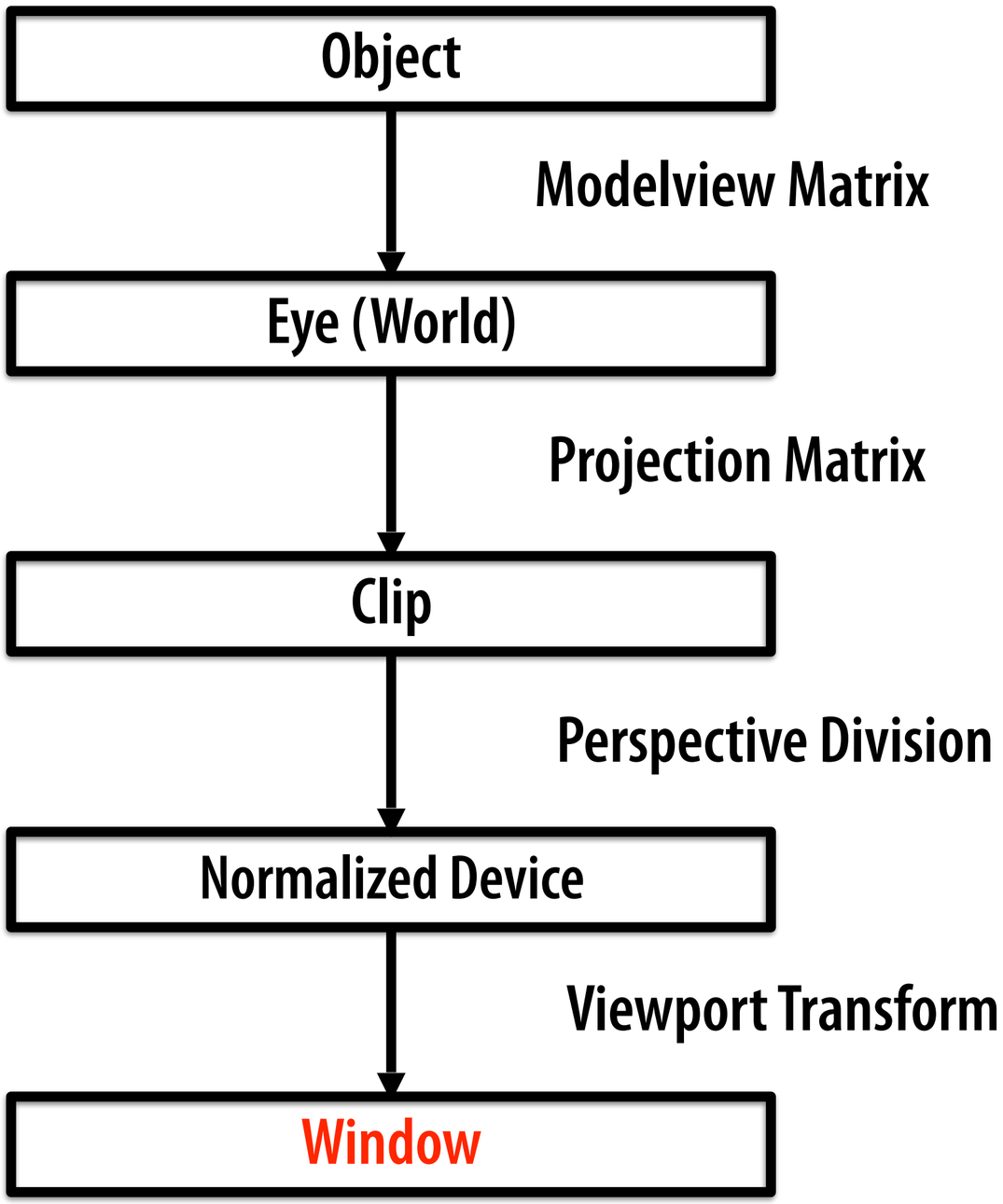
- Clip space—vertices are 4-vectors (have homogeneous *w*)
- Why? Clipping most efficient here—avoids perspective divide
- Vertex programs output positions in clip space
- Triangles assembled at this stage
- View volume clipping calculation against L/R/U/D/hither/yon: $\{x,y,z\}$ in $(-w, w)$

Coordinate System Transforms



- Normalized Device Coordinates
- $\{x,y,z\}$ in (-1, 1)
- Why? Culling most efficient here

Coordinate System Transforms



- Window/screen space
- Map {*x*,*y*}:

  (-1, -1) to (1, 1) →

  (0, 0) to (*w*, *h*)

Vertex Operations

- Vertex operations apply to vertexes independently

Transform coordinates and normal

- Model → world
- World → eye

Normalize the length of the normal

Compute vertex lighting

Transform texture coordinates

vertex program

Transform to clip coordinates

Divide by w

Apply affine viewport transform

Primitive Operations

- Primitive assembly
- Clipping
- Backface cull

Transform coordinates and normal

- Model -> world
- World -> eye

Normalize the length of the normal

Compute vertex lighting

Transform texture coordinates

Transform to clip coordinates

Assemble vertexes into primitives

Clip primitives against frustum

Divide by w

Apply affine viewport transform

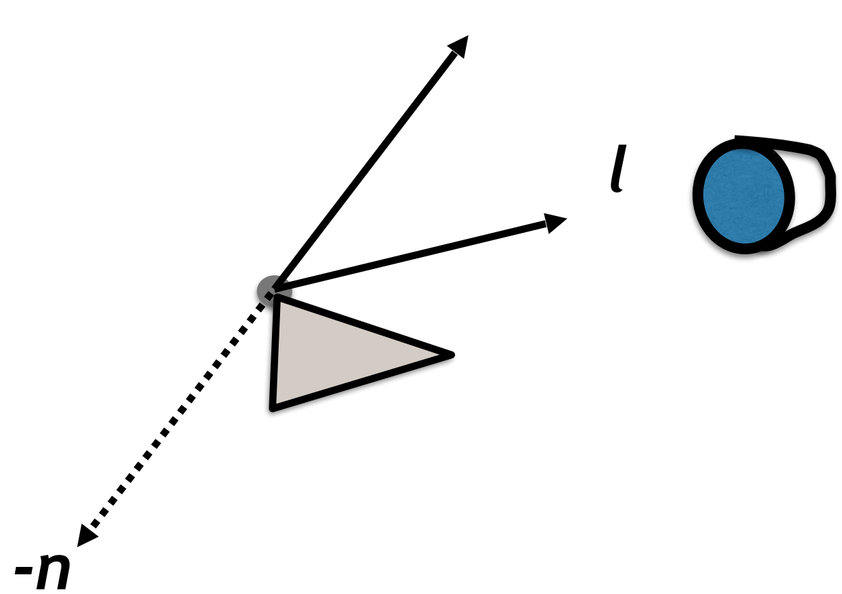
Eliminate back-facing triangles

Normal Transform

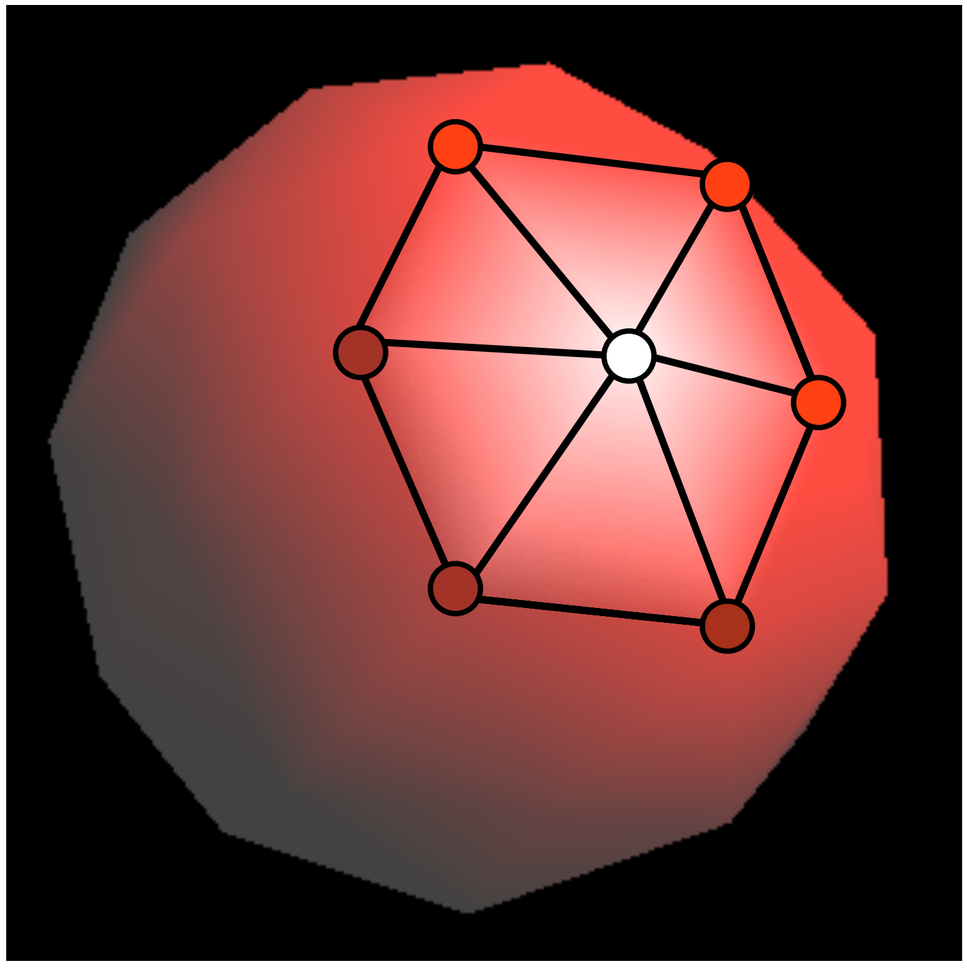
- Points and directions are duals; normals are directions
- Transform with inverse M (actually M_{3x3}^{-1})
- Approaches to acquiring M^{-1} include
  - Maintain separately, or
  - Compute when coordinate matrix is changed, or
  - Force specification by application
- Why transform normals? (Can lighting be computed in model coordinates?)
  - Model matrix may not be rigid
  - But yes, this is what's commonly done
- Why normalize normals to unit length?
  - Lighting equations require unit length
  - Non-rigid model matrix distorts lengths
  - Requires reciprocal square root operation

Fixed-function Lighting

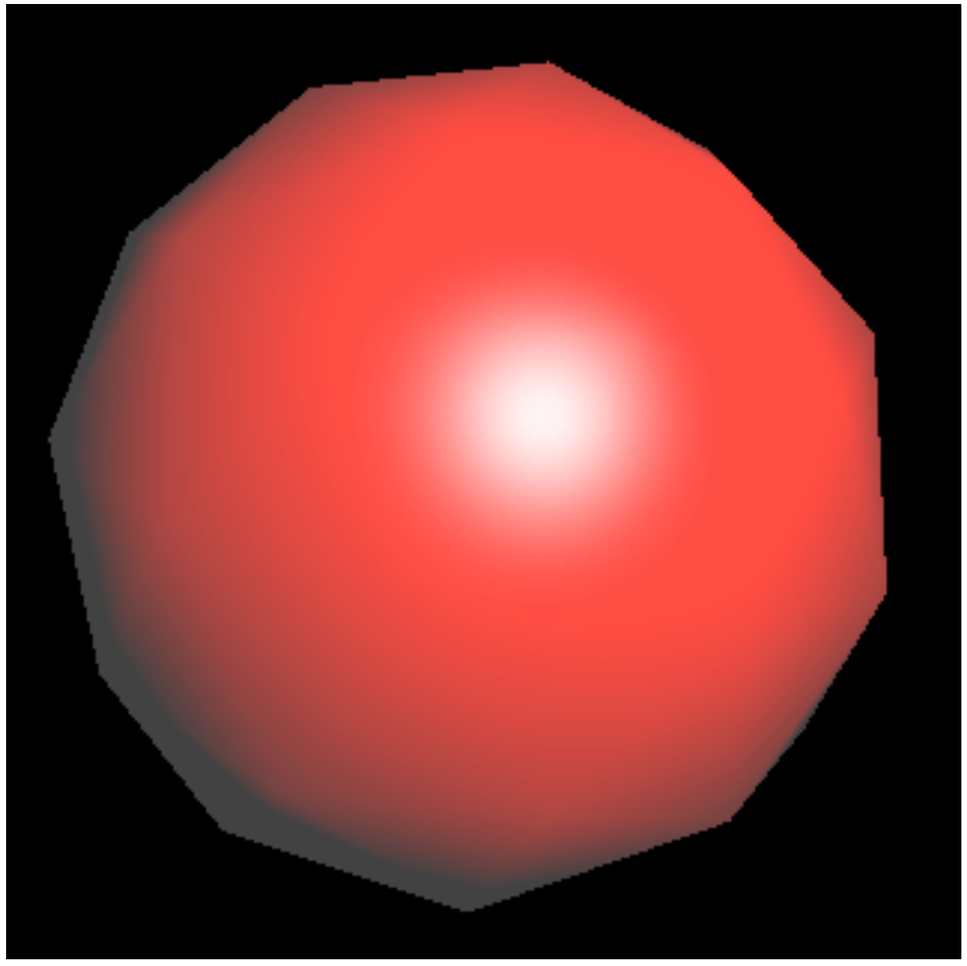
- Color = $e_{cm} + a_{cm}\, a_{cs} + \sum [a_{cm}\, a_{cli} + (n \cdot l)\, d + (n \cdot h_i)^s]$
  - Emissive: Color in absence of light
  - Ambient: Component of light not dependent on direction
  - Diffuse: "Flat", nonshiny color of light
  - Specular: Shiny highlights
- Simple $n \cdot l$ evaluation
  - Plus ambient and specular
  - Multiple lights …
  - Remember, n is not a facet normal
- Possibly two-sided



Example per-vertex computation: lighting



Per-vertex lighting computation



Per-vertex normal computation, per pixel lighting

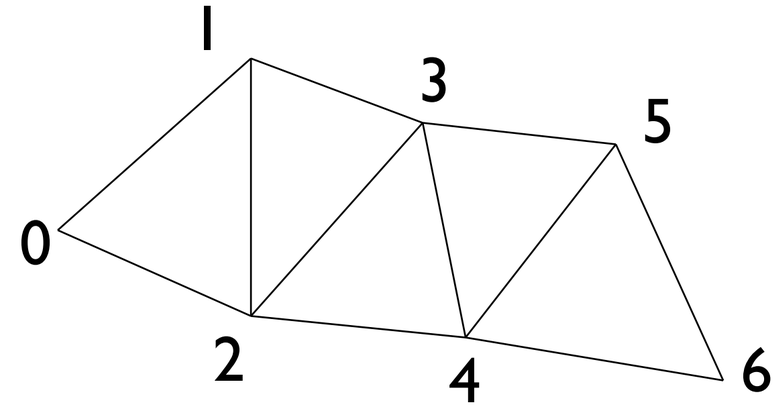
- Per-vertex data: surface normal, surface color
- Uniform data: light direction, light color

Primitive Assembly

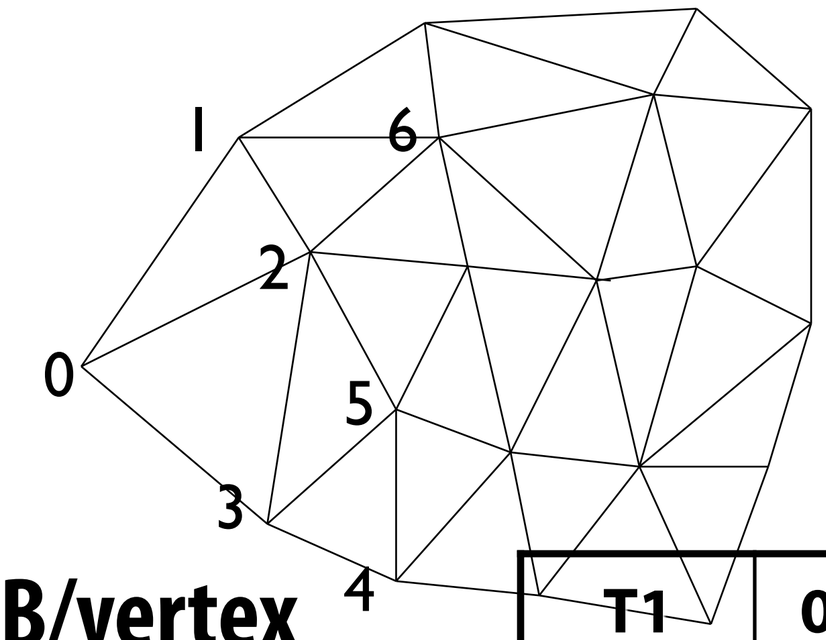
- Assemble based on application commands
  - Independent or strip or mesh or …
- Decompose to triangles
  - Prior to clipping to maintain invariance
- Algorithm properties
  - Fixed execution time (good)
    - All vertex operations up to this point have this property
  - Vertex interdependencies (bad)

Reusing Vertices

- Triangle strips
  - Ideal: 3x reduction
  - Straightforward to generate from higher order geometry



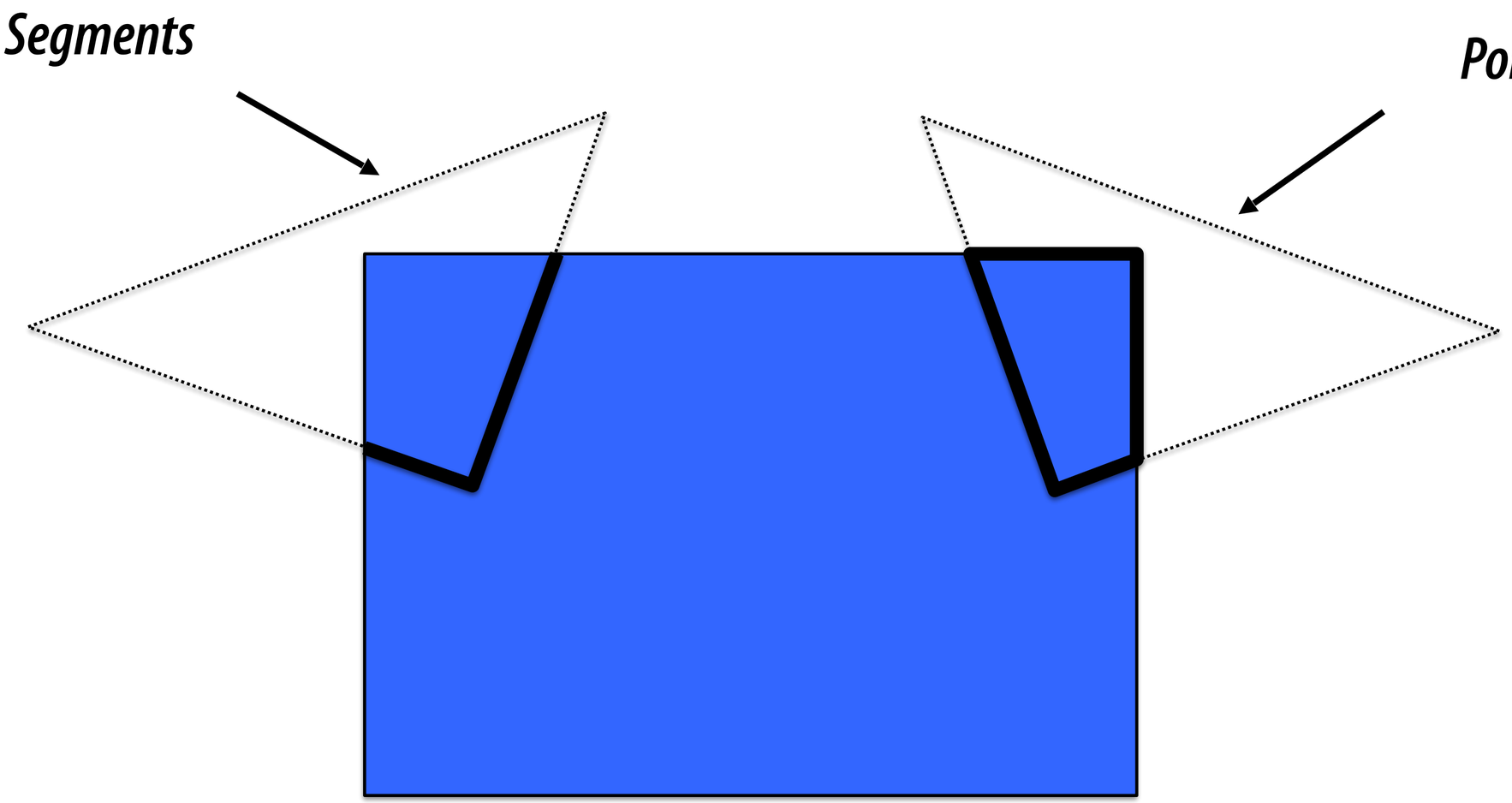
- Indexed triangle strips
  - Direct3D
  - Indexed vertices
  - Savings: ~1–2B/index vs. ~12–36B/vertex



T1	0,1,2
T2	0,2,3
T3	2,3,5
T4	2,5,6

Clipping

- Two types
  - Point, line: eliminates geometry
  - Polygon: eliminates and introduces edges



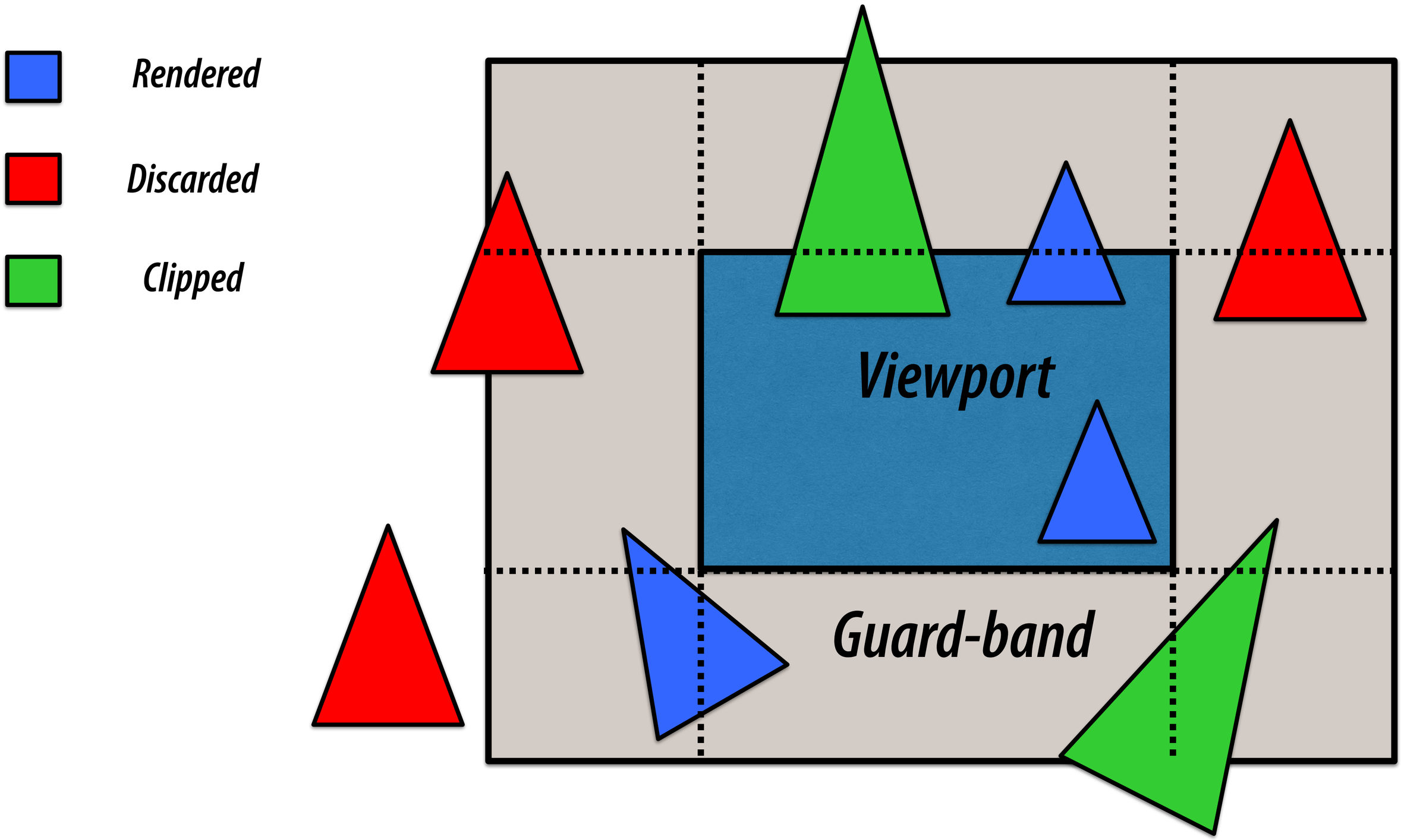
Clipping

- Two types
  - Point, line: eliminates geometry
  - Polygon: eliminates and introduces edges
- Algorithm properties
  - Vertex interdependencies (bad)
    - Data-dependent execution (worse)
      - Variable execution time (substantially different)
      - Variable code paths
  - If you know "scan" as a parallel algorithm, think about the "allocate" use of scan

Guard-Band Clipping

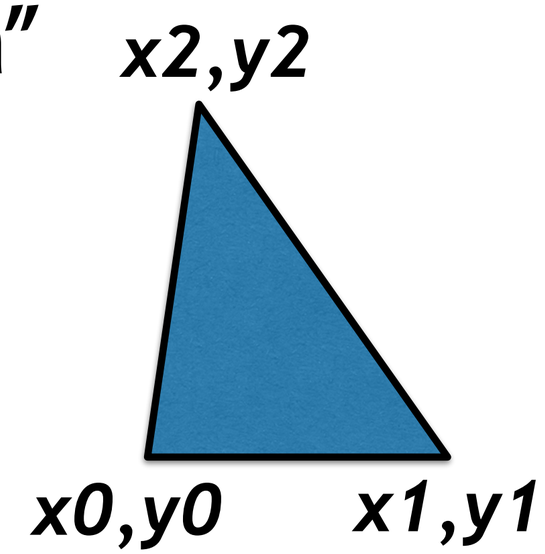
- Expand clipping frustum beyond desired viewport
  - Near and far clipping are unchanged
  - Frustum clip only when necessary
- Ideal triangle cases:
  - Discard if outside viewport, else
  - Render without clipping if inside frustum, else
    - Rasterizer must scissor well for this to be efficient
  - Clip triangles
    - That cross both viewport and frustum boundaries
    - That cross the near or far clip-planes
- Operation is imperfect, but conservative

Guard-Band Clipping



Backface Cull

- Facet facing toward or away from viewpoint?
  - No facet normal (other APIs?)
  - Use sign of primitive's window coordinate "area"
    - Remember, only triangles are planar
- Use facing direction to
  - Select lighting result (for *n* or –*n*)
  - Potentially discard the primitive
- Advance in sequence to improve efficiency?



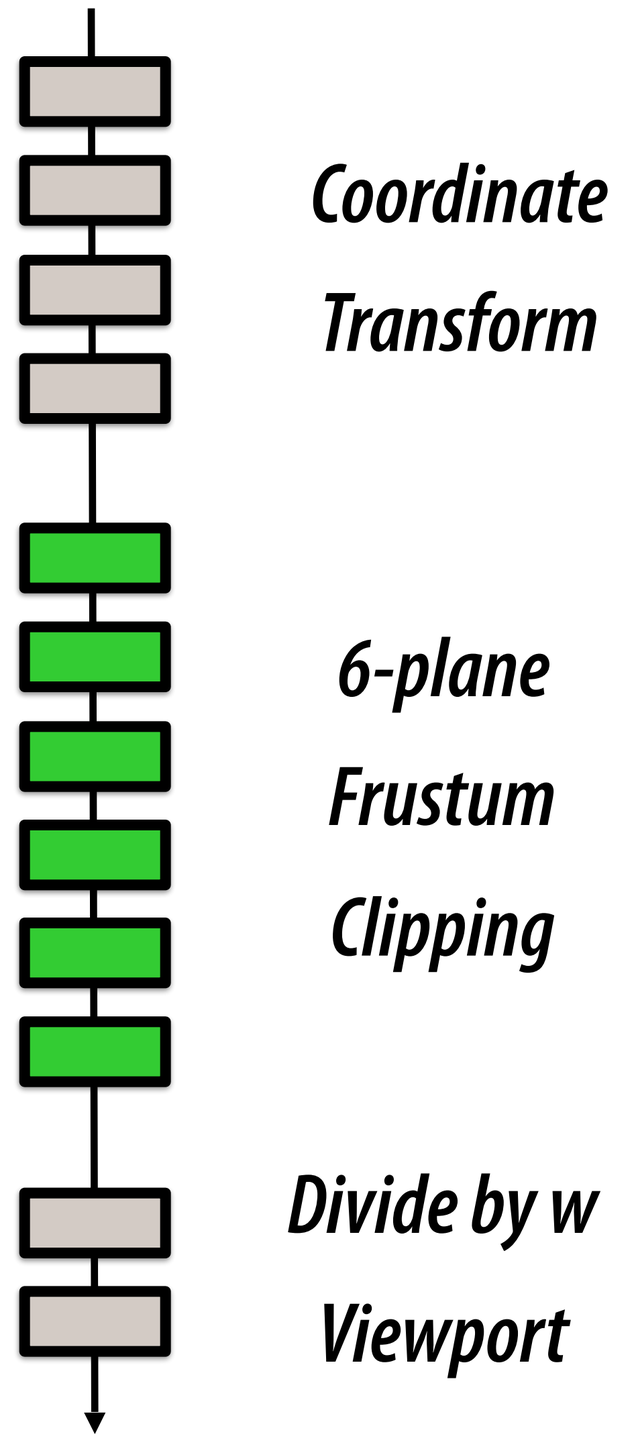
$$\textit{Triangle area} = \frac{(x0y1 - x1y0) + (x1y2 - x2y1) + (x2y0 - x0y2)}{2}$$

Some Examples

- Systems
  - Clark Geometry Engine (1983)
  - Silicon Graphics GTX (1988)
  - Silicon Graphics RealityEngine (1992)
  - Silicon Graphics InfiniteReality (1996)
  - Modern GPU (c. 2001)
- What we'll look at
  - Organization of the geometry system
  - Distribution of vertex and primitive operations
  - How clipping affects the implementation

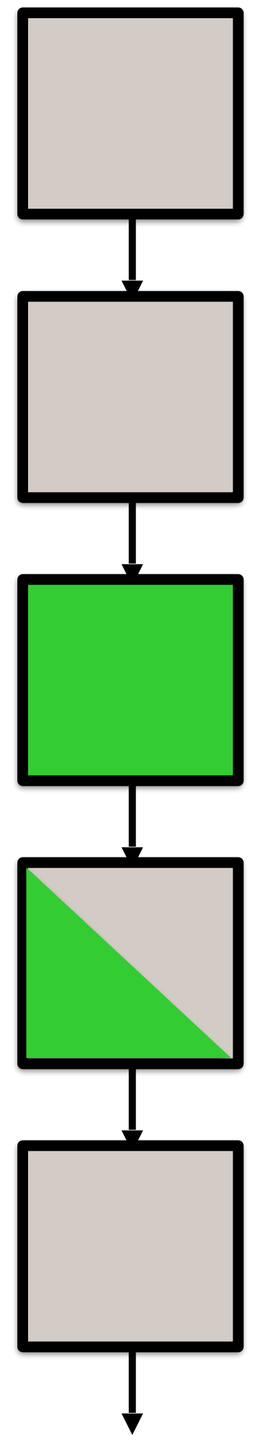
Clark Geometry Engine (1983)

- 1st generation capability
- Simple, fixed-function pipeline
  - Twelve identical engines
  - Soft-configured at start-up
- Clipping allocated 1/2 of total 'power'
  - Performance invariant (good)
  - Typically idle (bad)



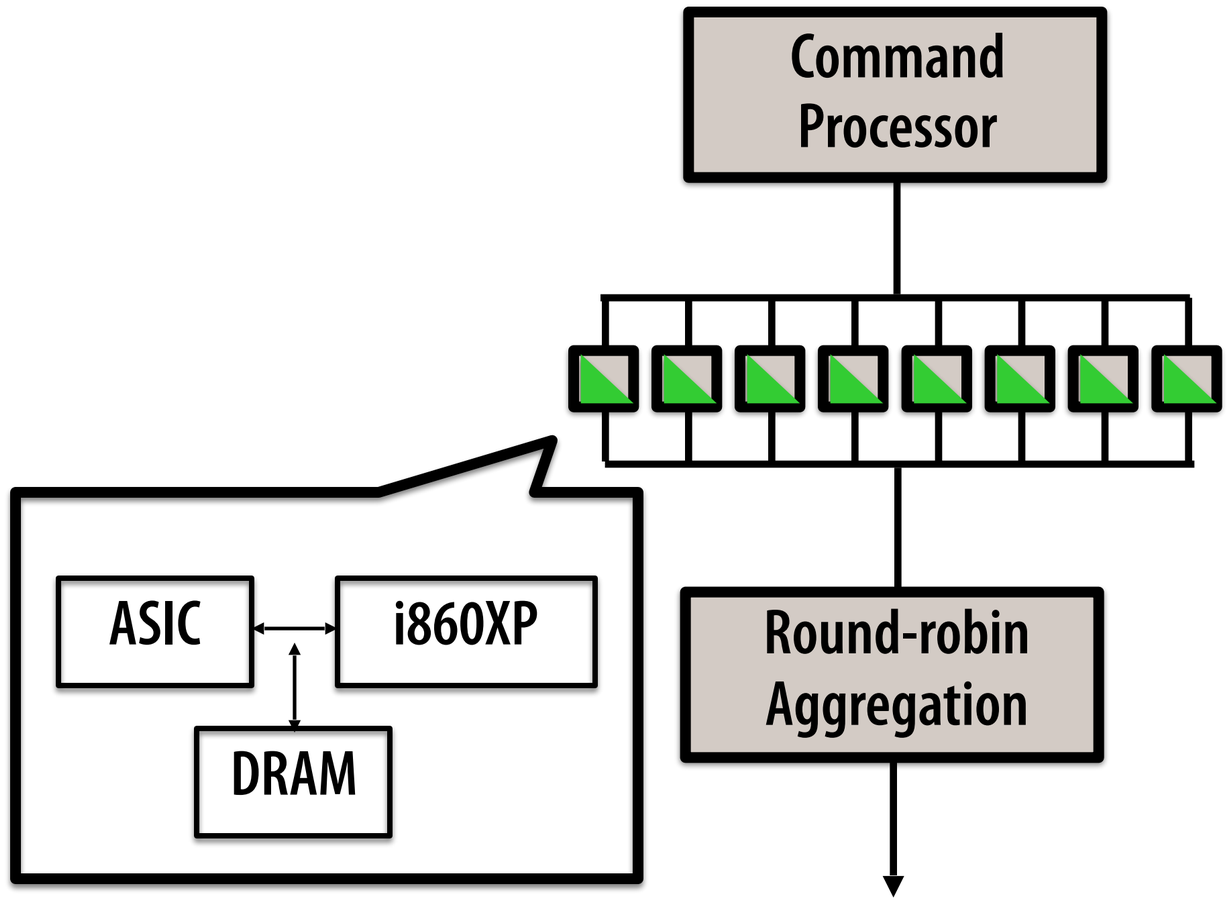
GTX Geometry Engine (1988)

- 2nd generation capability
- Variable functionality pipeline
  - 5 identical engines
  - Modes alter some functions
    - Load balancing is difficult
- Clipping allocated 1/5 of 'power'
  - Clip testing is the constant load
  - Actual clipping
    - Slow pipeline execution (bad)
      - out of balance
    - Typically isn't invoked (good)



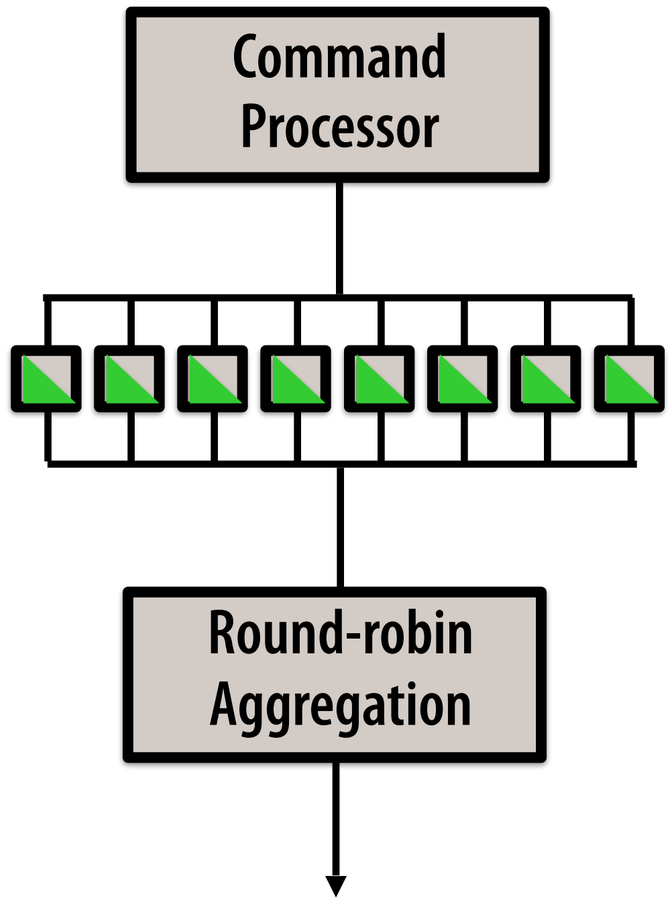
RealityEngine Geometry (1992)

- 3rd generation capability
- Variable-functionality MIMD organization
  - Eight identical engines
  - Round-robin work assignment
  - Good 'static' load balancing
- Command processor
  - Splits large strips of primitives
  - Shadows per-vertex state
  - Broadcasts other state
- Primitive assembly
  - Complicates work distribution
  - Reduces efficiency of strip processing



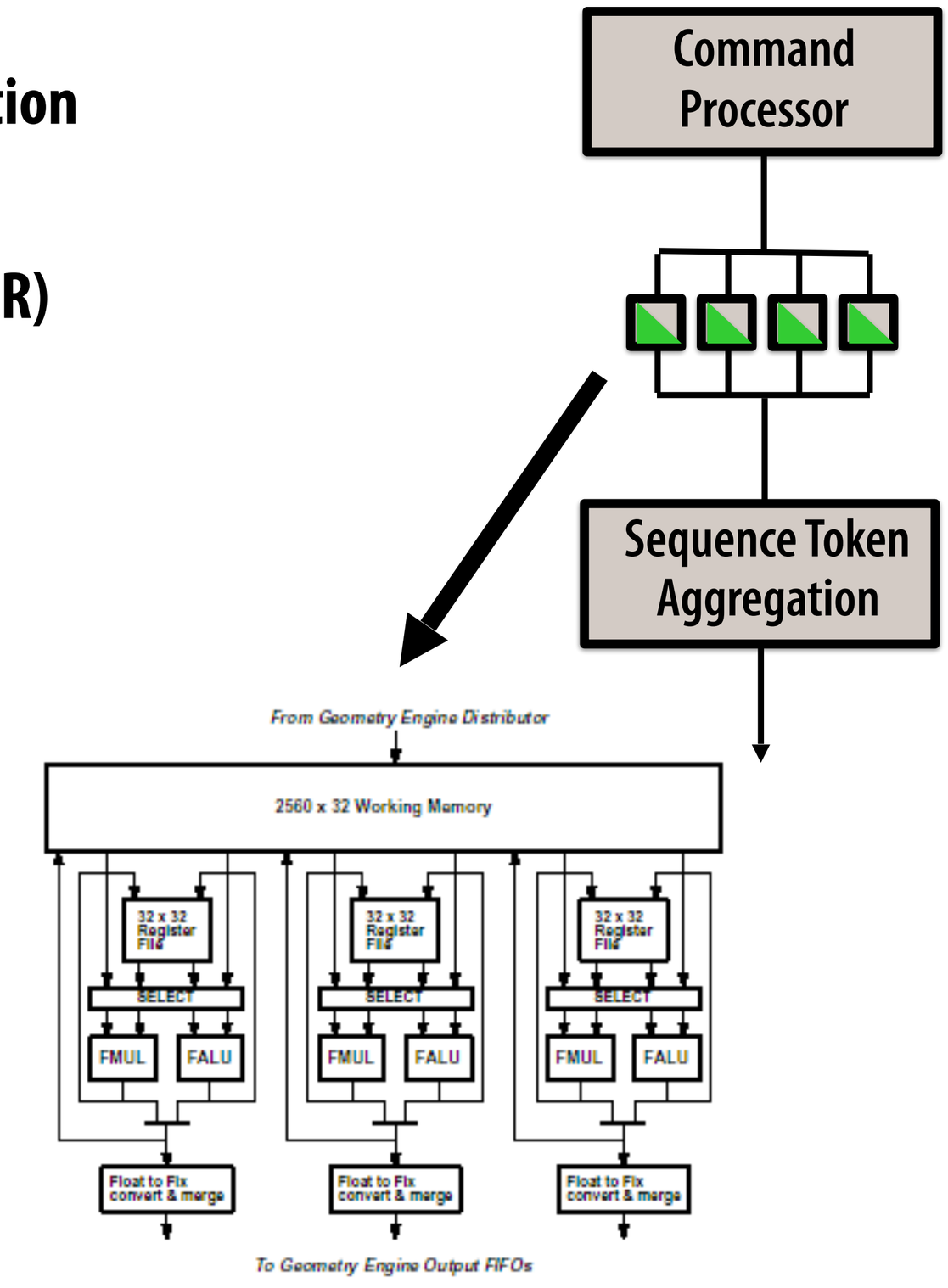
RealityEngine Clipping

- Clipping introduces data-dependent (dynamic) load
  - Cannot be predicted by CP
- Dynamic load balance accomplished by:
  - Round-robin assignment
  - Large input and output FIFOs
  - For each geometry processor
  - Sized greater than (*n*) *long*/*typical*
- Large work load per processor
  - Minimize the *long*/*typical* ratio
  - Unlike pipeline processing



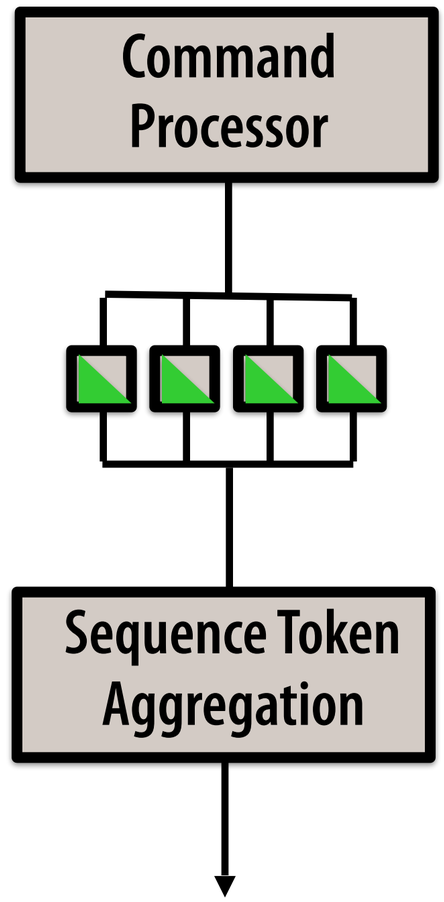
InfiniteReality Geometry (1996)

- 3rd generation capability
- Variable-functionality MIMD organization
  - Four identical (SIMD) engines
  - Least-busy work assignment (not RR)
  - Good 'static' load balancing
- Command processor
  - Splits large strips of primitives
  - Shadows per-vertex state
  - Broadcasts other state
- Geometry Engine
  - Custom ASIC—3 FP cores, 9-ported 2560 word memory
  - 4 pipeline stages



InfiniteReality Clipping

- Dynamic load balance accomplished by:
  - Least-busy assignment
  - Even larger FIFOs
  - Large work load per processor
- Likelihood of clipping reduced by
  - Guard-band clipping algorithm



GPU Geometry Engine (c. 2001)

- Utilizes homogeneous rasterization algorithm
  - No clipping required
  - Hence no primitive assembly prior to rasterization
    - Push backface cull to rasterization setup
    - This is where triangle area is computed anyway
    - I've heard NVIDIA has gone back to clipped rasterizat'n
- All geometry engine calculations are
  - On independent vertexes—easy work distribution
  - Not data dependent—minimal code branching
- Allows efficient, SIMD geometry engine implementation

Programmability?

Display Lists

- Package of commands, bundled into single list
  - Unmodifiable
- Examples:
  - Reusable (static) geometry
  - Matrix operations
  - Raster bitmaps / images
  - Light / lightmodel / material specifications
- Work well with client-server
- Storage on GPU?
- Optimization by driver?

Vertex Buffers/Arrays

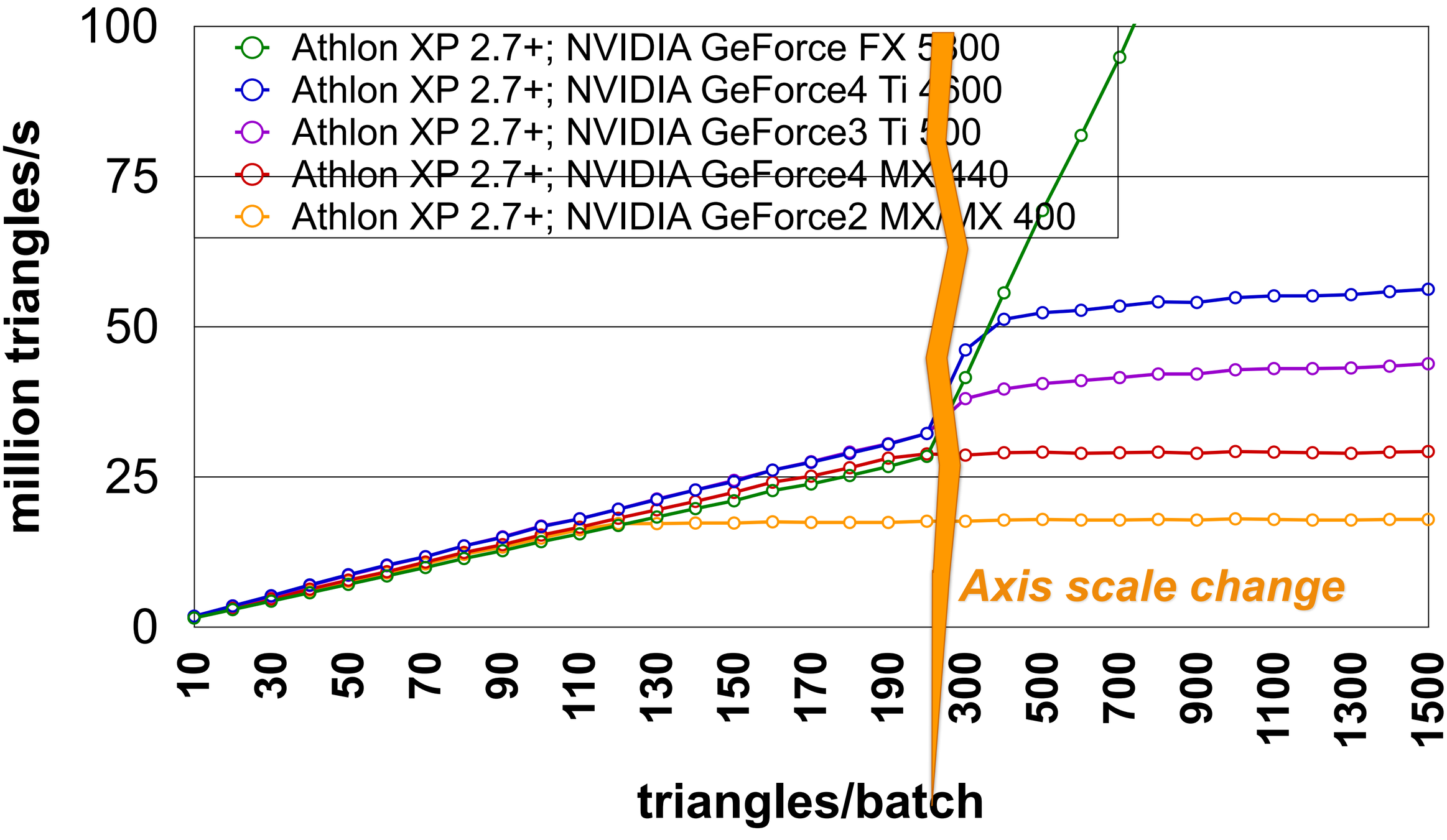
```
  vtx = (GLfloat *) malloc (numvtx*2*sizeof(GLfloat));
color = (GLfloat *) malloc (numvtx*3*sizeof(GLfloat));
index = (GLuint *)  malloc (numvtx*sizeof(GLuint));

/* Set up the pointers */
glVertexPointer(2, GL_FLOAT, 0, vtx);
glEnableClientState(GL_VERTEX_ARRAY); // deprecated
glColorPointer(3, GL_FLOAT, 0, color);
glEnableClientState(GL_COLOR_ARRAY);

glDrawElements(GL_TRIANGLE_STRIP, (ntri+1)*2,
               GL_UNSIGNED_INT, index);
```

- Big picture: 1) Data goes into special memory; 2) reduce CPU overhead of call

Measured Batch-Size Performance



CPU Limited?

- Yes, at < 130 tris/batch (avg) you are
  - completely,
  - utterly,
  - totally,
  - 100%
  - CPU limited!
- CPU is busy doing nothing, but submitting batches!

You get X batches per frame,

X mainly depends on CPU spec

25k batches/s @ 100% 1GHz CPU

Up to you!

Anything between 1 to 10,000+ tris possible

If small number, either

Triangles are large or extremely expensive

Only GPU vertex engines are idle

Or

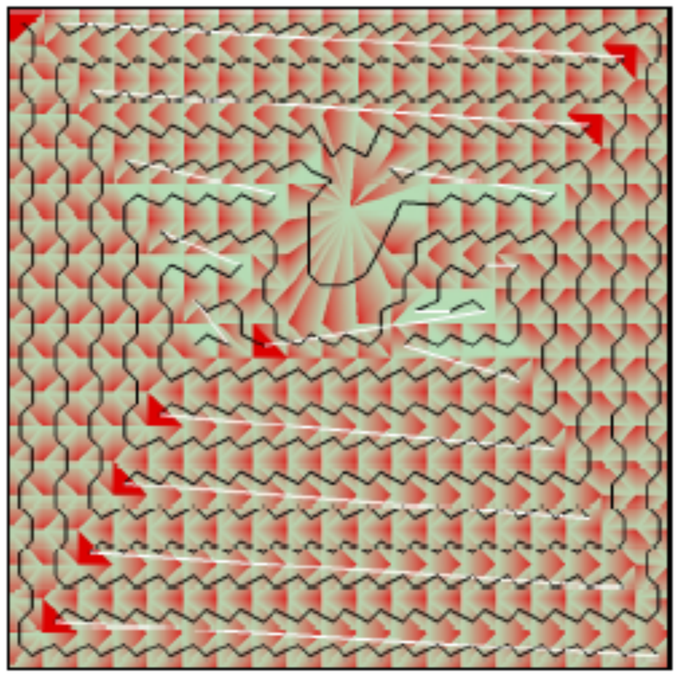
Game is CPU bound, but don't care because you budgeted your CPU ahead of time, right?

GPU idle (available for upping visual quality)

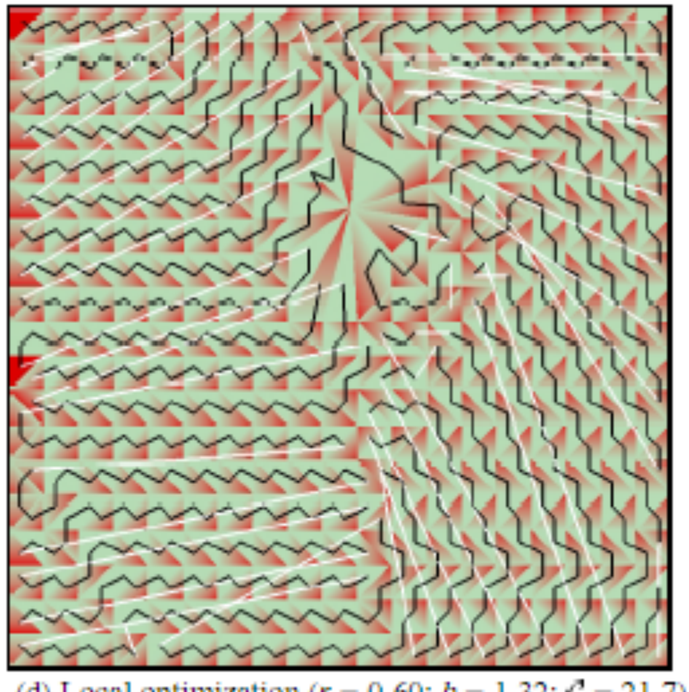
Vertex Caches

- "Optimization of Mesh Locality for Transparent Vertex Caching", Hugues Hoppe, Siggraph '99

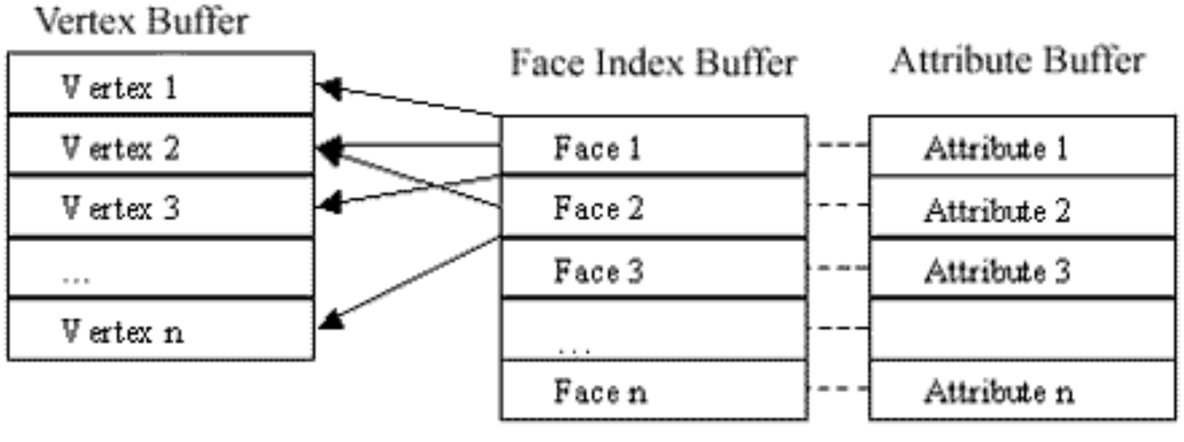
  Reduces redundant T&L cost
- 16? entry FIFO cache ('99)
- Driver must tessellate
- 0.60 cache misses/tri



(b) Traditional strips ($r = 0.99$; $b = 1.29$; $\mathcal{C} = 34.3$)



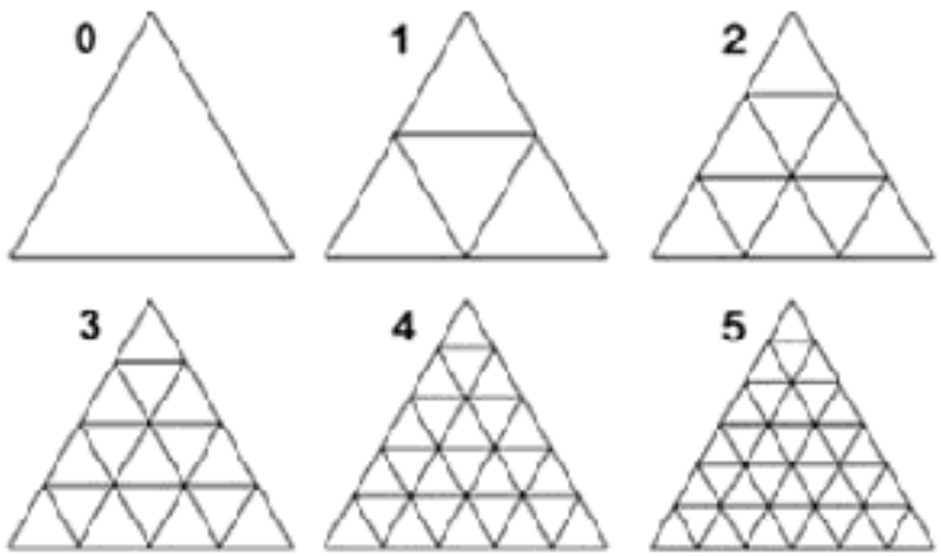
(d) Local optimization ($r = 0.60$; $b = 1.32$; $\mathcal{C} = 21.7$)



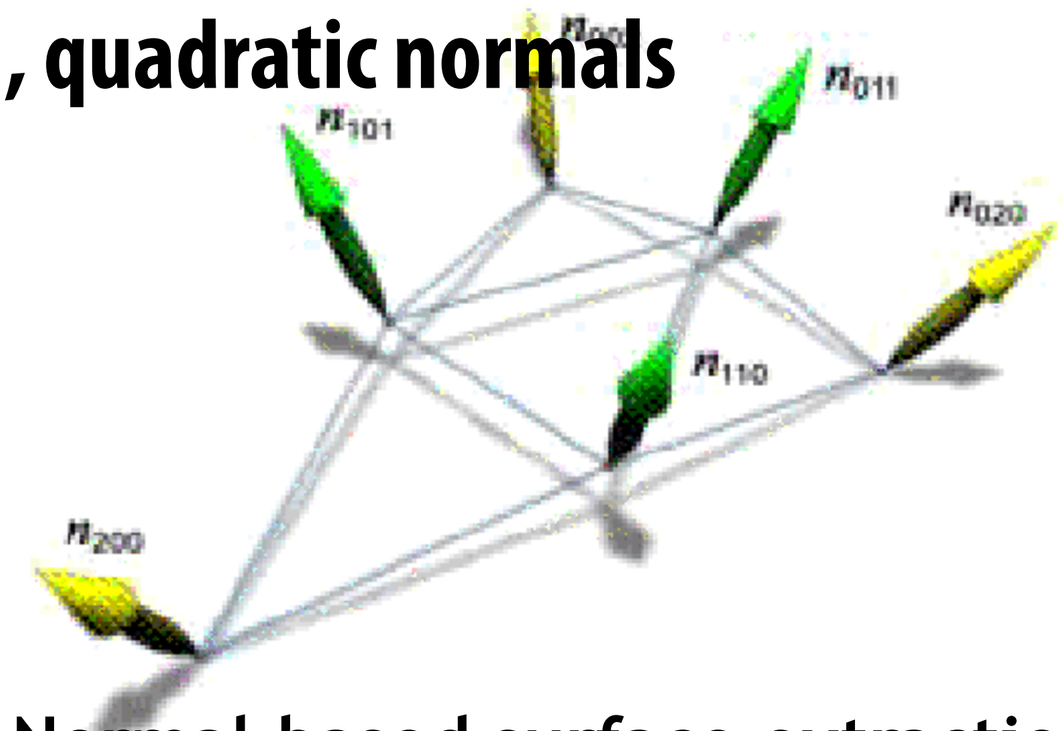
I know these are fake

ATI's TRUFORM Technology

- Vlachos et al., "Curved PN Triangles", I3D 2001
- Requires minimal setup
  - Single mode (triangle edge subdivision count)
  - No geometric data
- Operates on triangles as normally submitted
  - Cubic geometry interpolation, quadratic normals



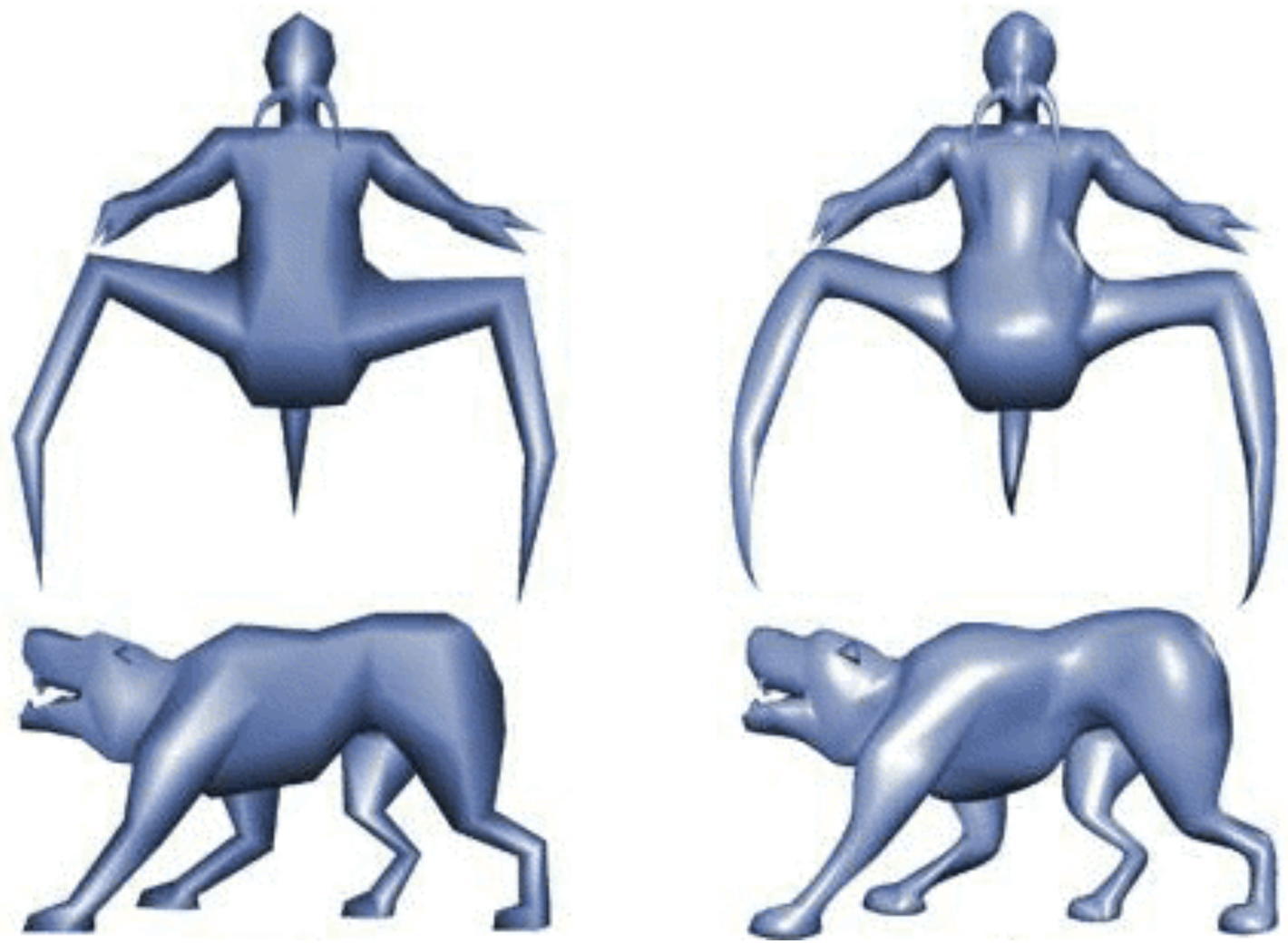
Edge Subdivision Count



Normal-based surface extraction

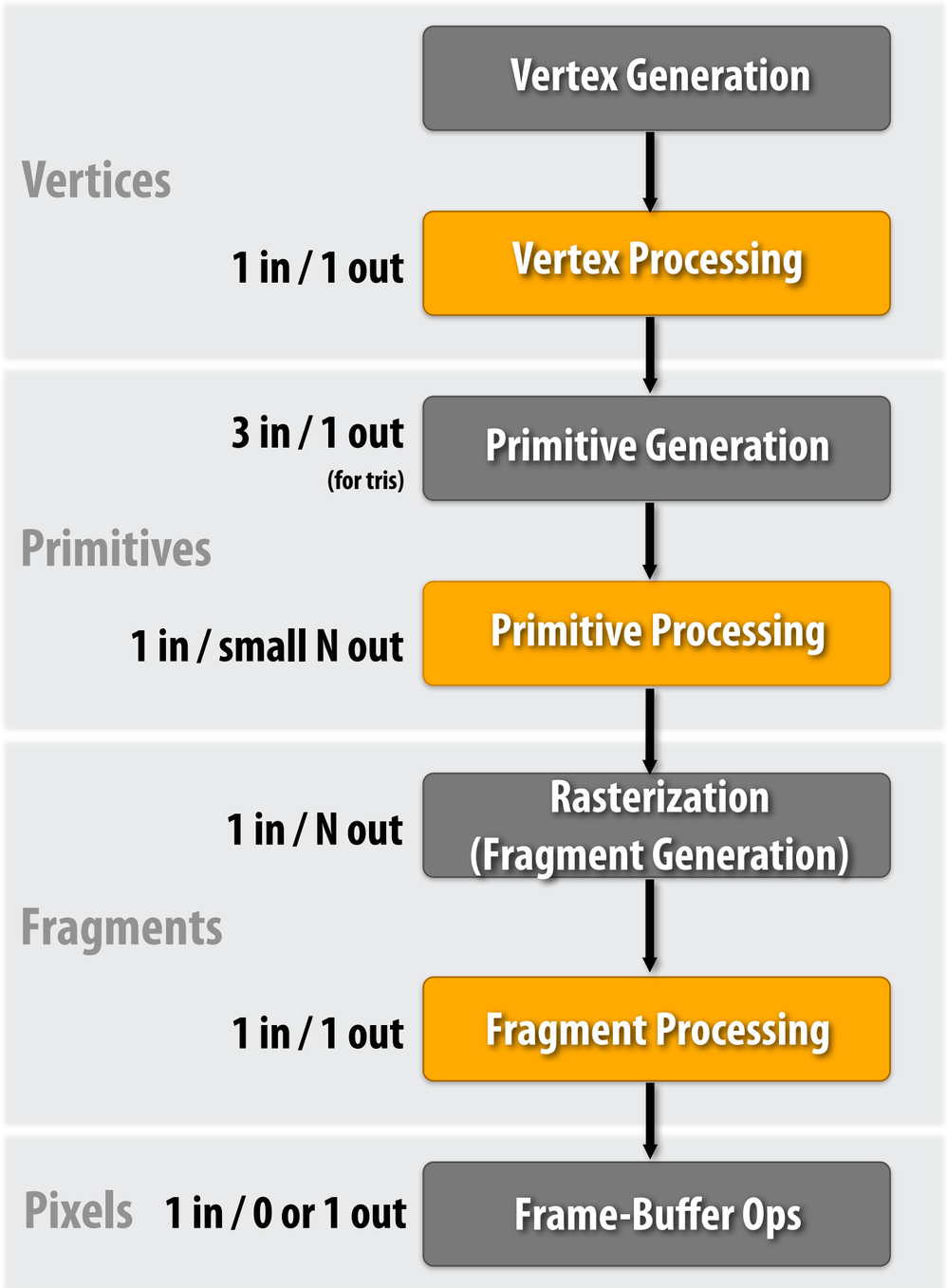
ATI's TRUFORM Technology

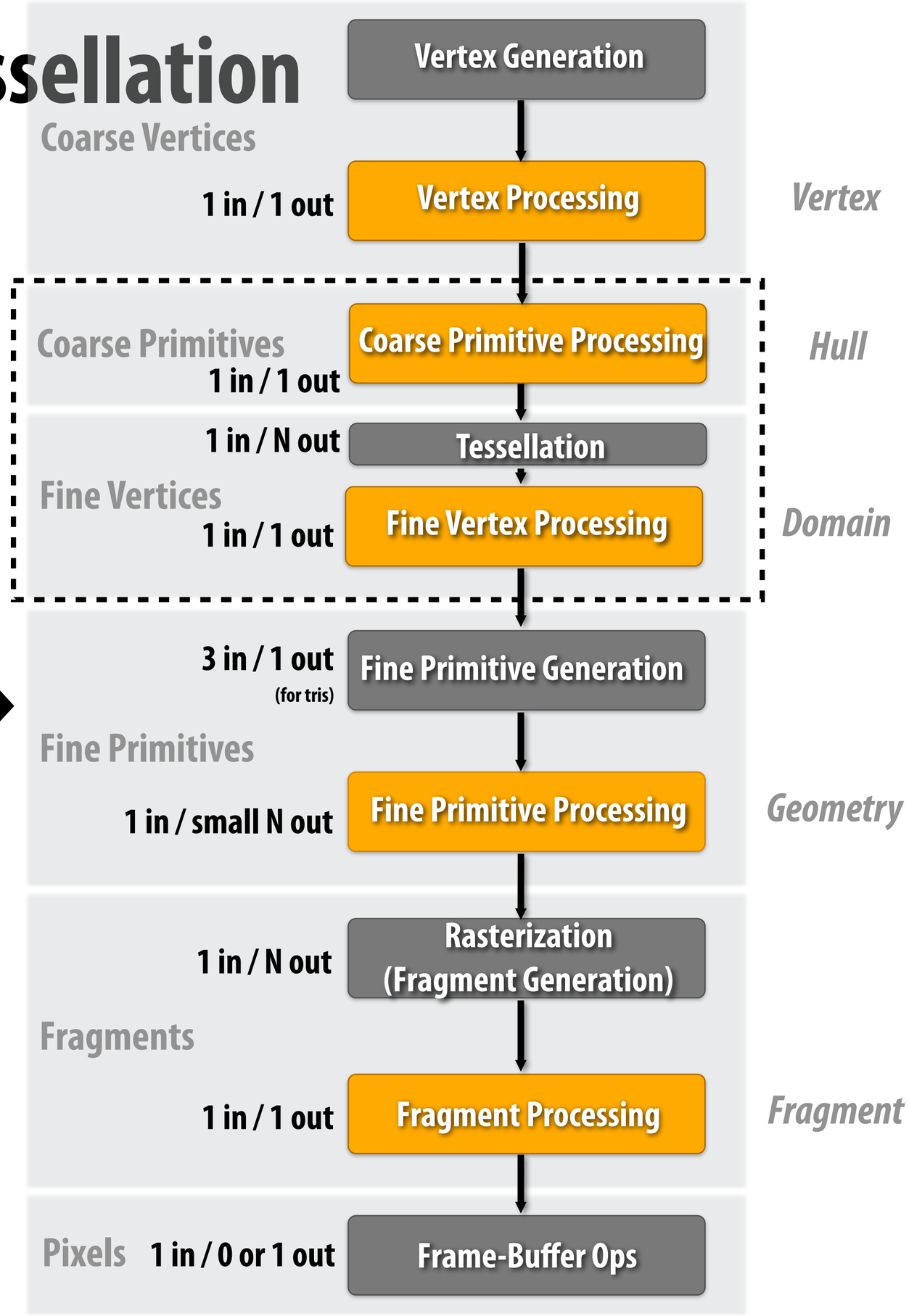
- Improves silhouettes
- Improves lighting over per-vertex (per fragment?)



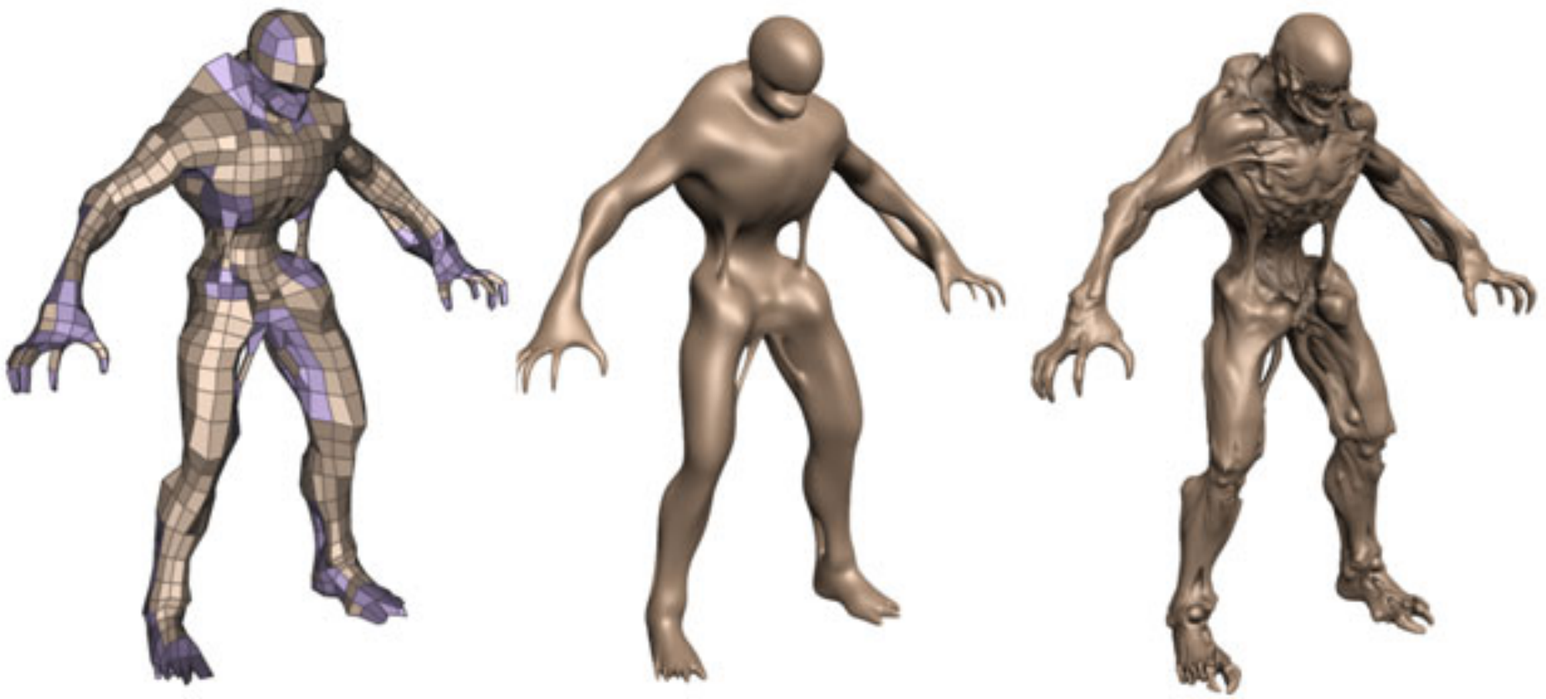
Graphics pipeline with tessellation

- **Five programmable stages in modern pipeline (OpenGL 4, Direct3D 11)**





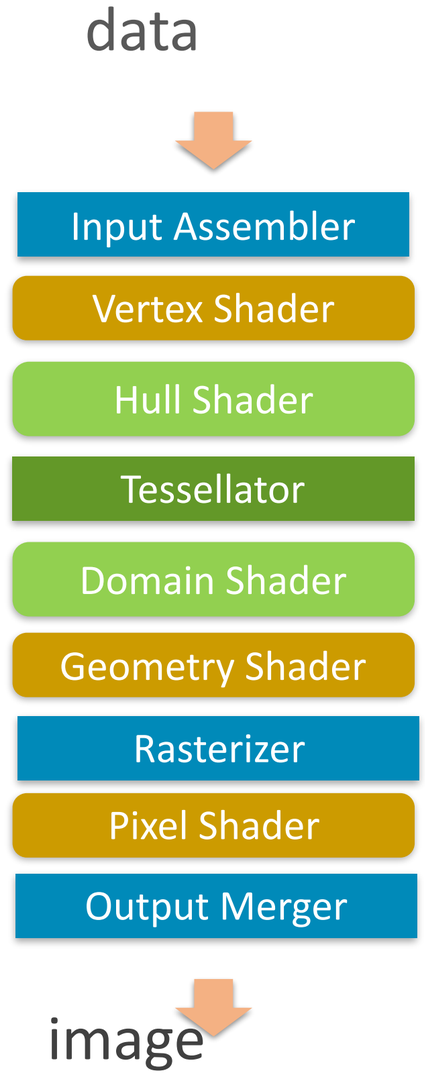
HW Tessellation



http://www.nvidia.com/object/tessellation.html

DirectX 11

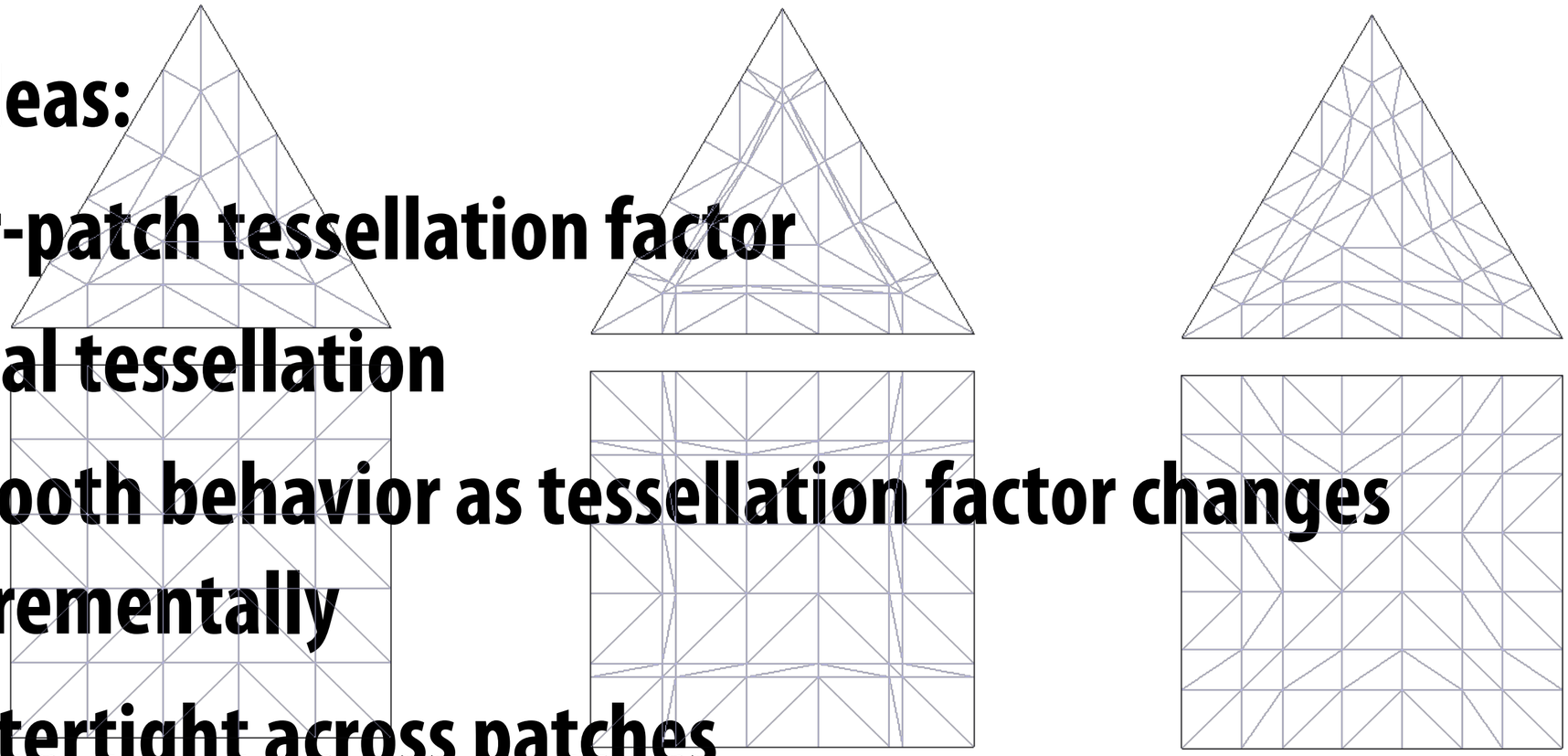
- Hardware Tessellation
  - 3 pipeline stages
  - support for higher order surface patches
- **Hull-Shader Stage** - A programmable shader stage that produces a geometry patch (and patch constants) that correspond to each input patch (quad, triangle, or line).
- **Tessellator Stage** - A fixed function pipeline stage that creates a sampling pattern of the domain that represents the geometry patch and generates a set of smaller objects (triangles, points, or lines) that connect these samples. (32b floating-pt + 16b fixed-pt)
- **Domain-Shader Stage** - A programmable shader stage that calculates the vertex position that corresponds to each domain sample.

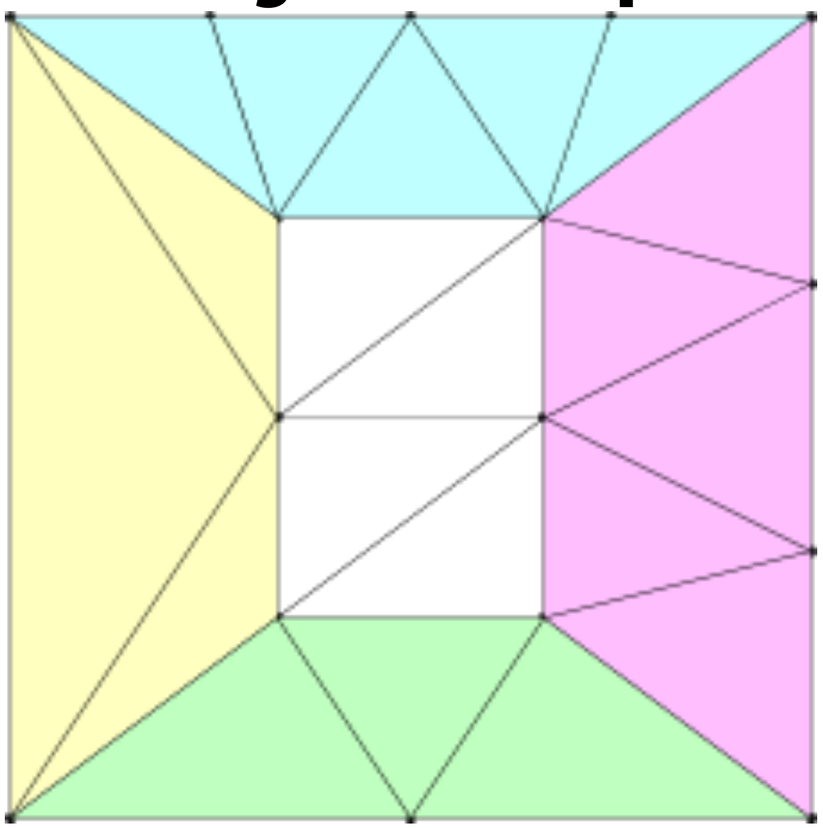


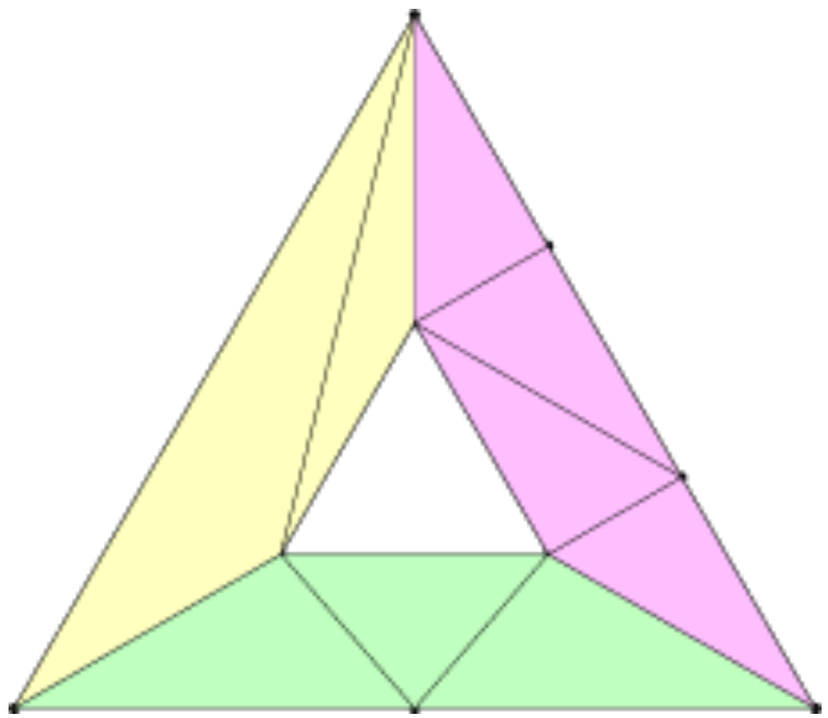
thanks to Charles Loop for these slides

Hardware Tessellator

- Key ideas:
  - Per-patch tessellation factor
  - Local tessellation
  - Smooth behavior as tessellation factor changes incrementally
  - Watertight across patches







http://fgiesen.wordpress.com/2011/09/06/a-trip-through-the-graphics-pipeline-2011-part-12/