Lecture 11:

Juggling the Pipeline

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

Format for presentations

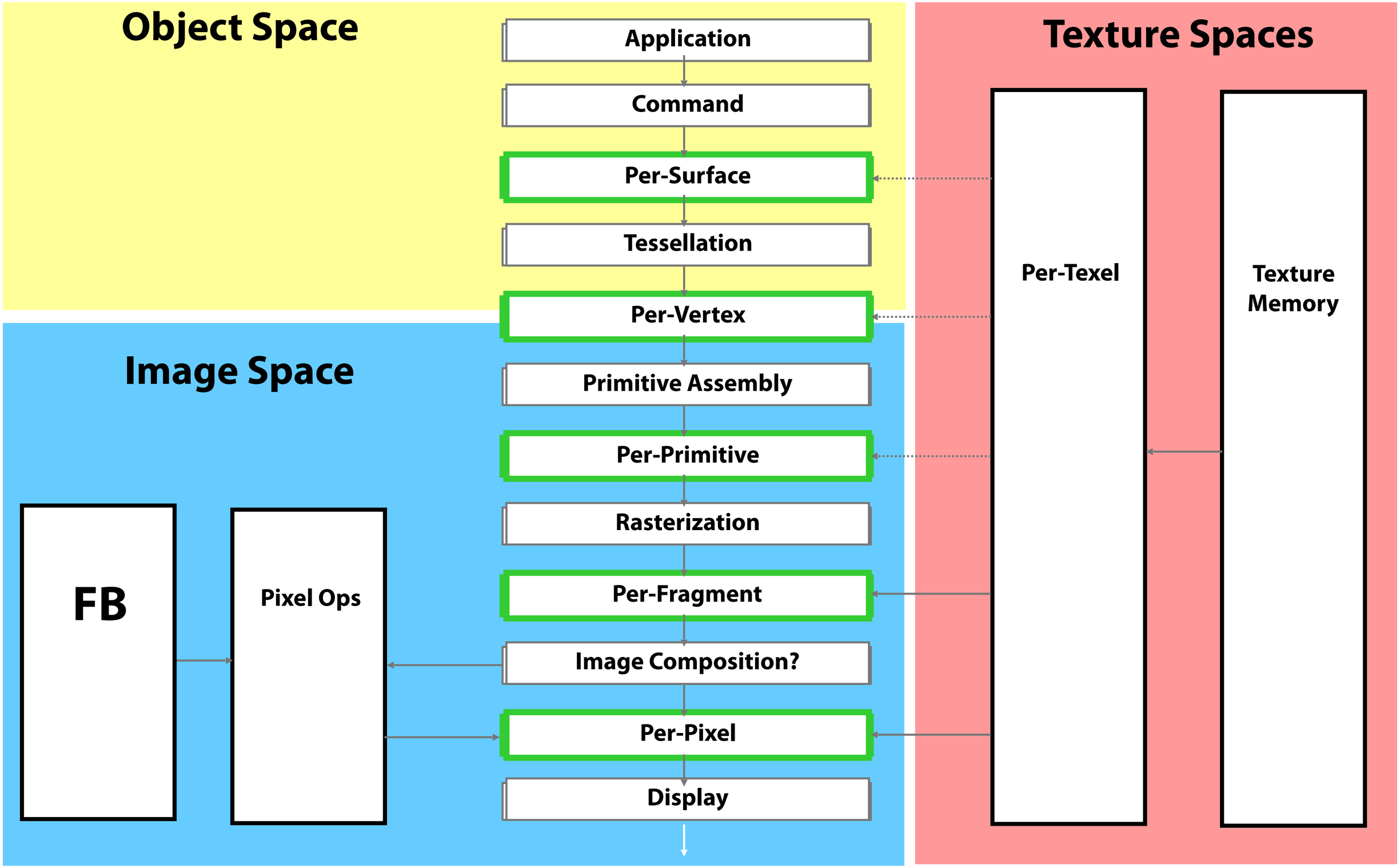
- Suggestion: 3 slides per group:
  - Motivation: why is your problem important?
  - Approach: how did you go about solving the problem?
  - Results: what did you learn?
- Presentations must be self-contained (not OK to say "well, we just haven't finished this yet, it'll be in the final report")
- Multi-student group: Both students must speak

Final project presentations (next Th)

- Thursday:
  - Each group has FOUR minutes
  - Do not go over FOUR minutes
  - Turn in a set of PDF slides (SmartSite) BY 8 am Thursday
  - I will assemble them into a presentation
  - Do not go over FOUR minutes
  - Do not go over FOUR minutes

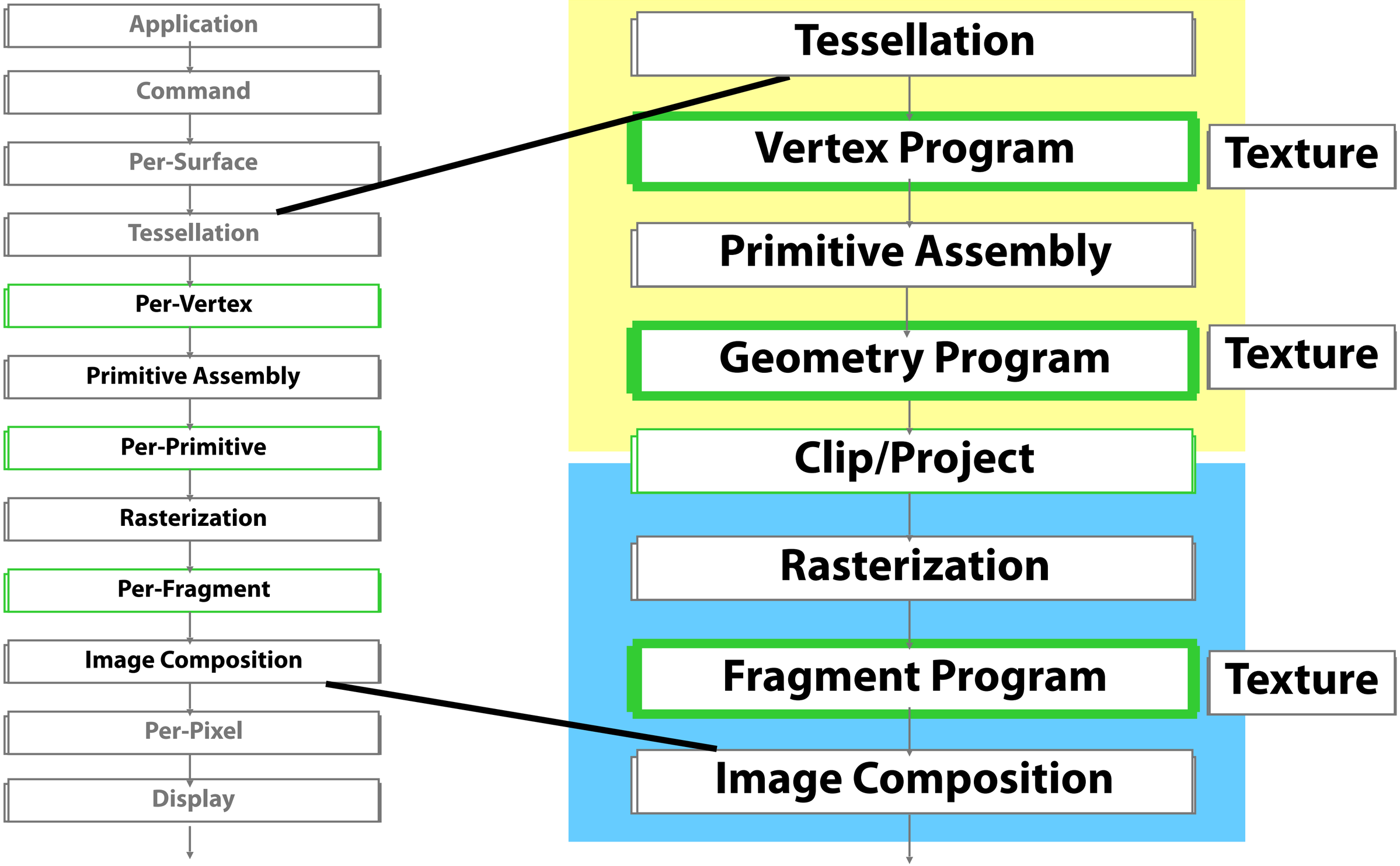
- Reports are also due Thursday
  - You will be happy that the class is done on Thursday
  - Your teaching staff appreciates brevity
  - What did you learn? (What did you learn about graphics systems?)

Generalized Graphics Pipeline?



[From Akeley and Hanrahan, Real-Time Graphics Architectures]

OpenGL 3.2 in Generalized Pipeline



Design alternatives

Hidden-surface elimination

- painters algorithm (host) = hide-first
- hierarchical z-buffer = hide-(partially)-middle
- z-buffer = hide-last

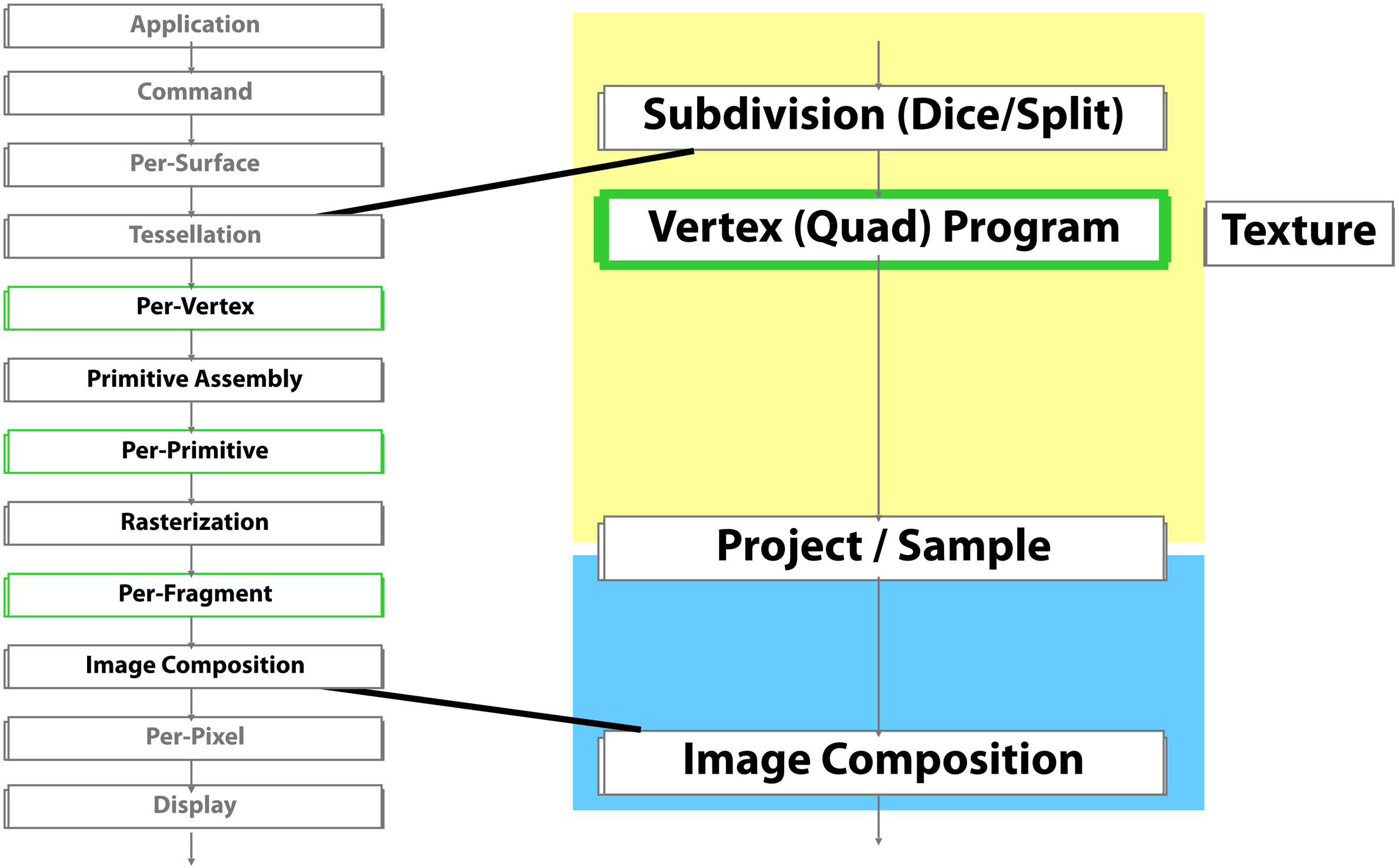
Texturing

- Fragment textures = texture-last
- Vertex textures = texture-first

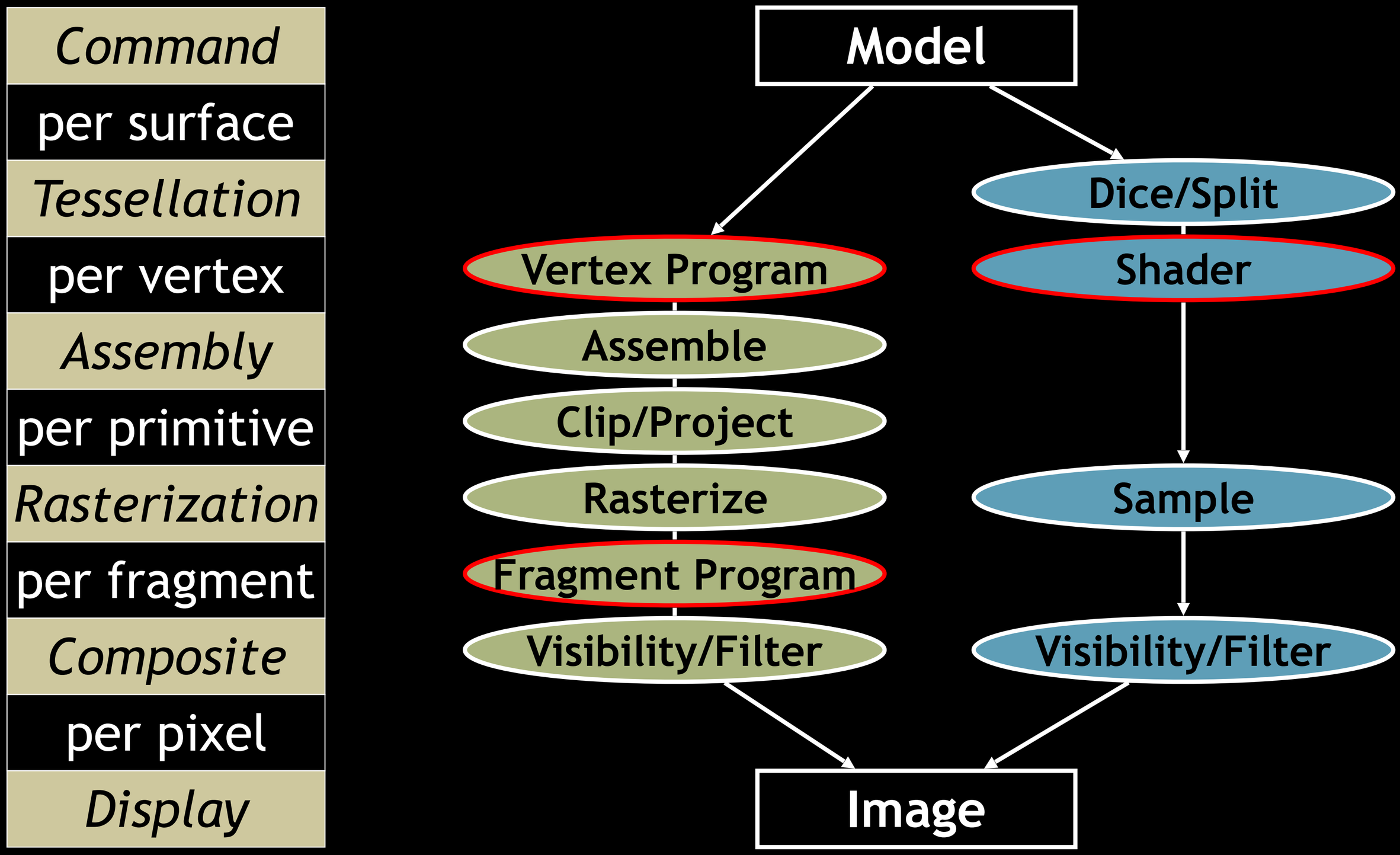
Shading

- Vertex shading = shade-first or shade-first-vertex
- Fragment shading = shade-last-fragment
- Deferred shading = shade-last-pixel

Reyes In Generalized Pipeline



OpenGL vs. Reyes



Summary of Pipeline Differences

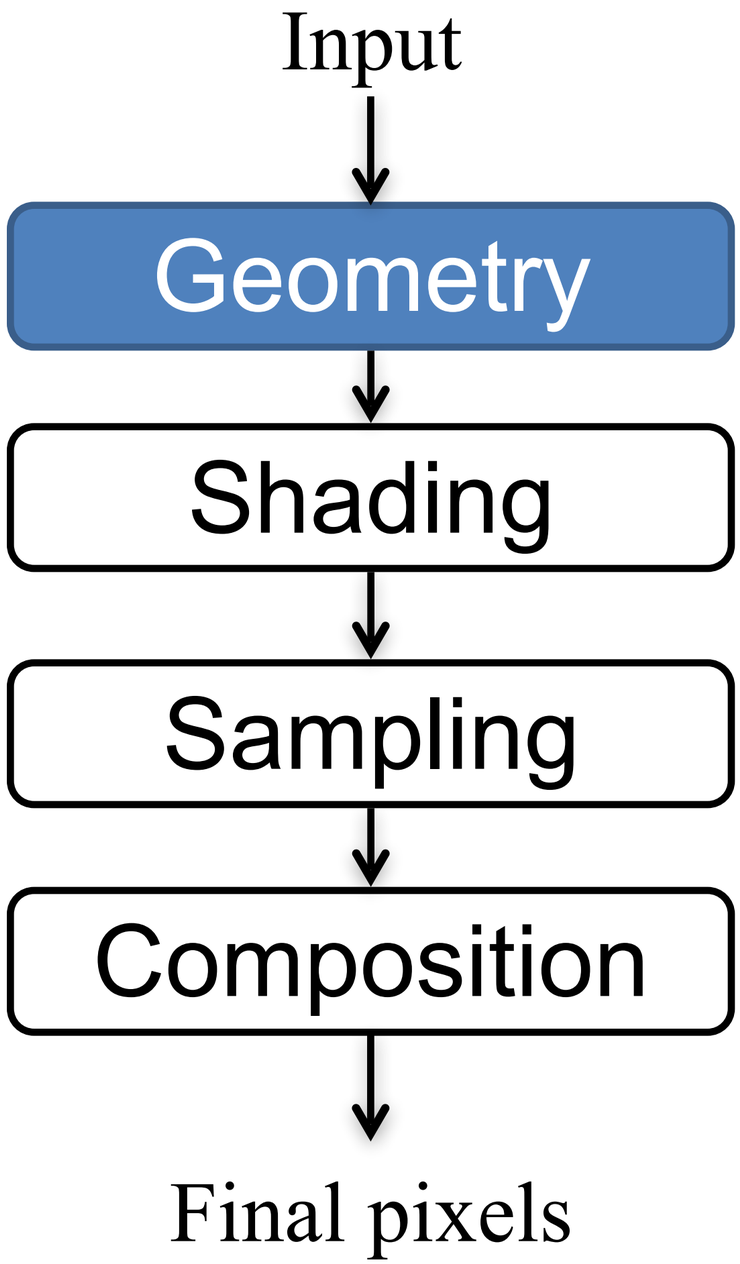
- **Shading and texturing**
  - OpenGL: *n* shaders, *m* coordinate spaces
  - Reyes: Single shader, single coordinate space
- **Sampling vs. rasterization**
  - OpenGL: rasterizes arbitrary-sized triangles
  - Reyes: samples bounded-sized quads
- **Tessellation**
  - OpenGL: tessellates in host or at compile time
  - Reyes: tessellates dynamically as part of pipeline

Shading and Texturing

- **OpenGL**
  - Vertex shading: eye space
  - Fragment shading: screen space
    - Textures require filtering (mipmapping, 8 samples/access)
    - Imagine OpenGL: mipmapped scenes are > 2x slower than point-sampled
    - Factoring advantageous for large triangles, but must support two shading units
- **Reyes**
  - Vertex/quad shading: eye space
    - Coherent access textures: samples are properly filtered
  - Gain ability to shade before pixel coverage calculation: motion blur, depth of field

Step 1: Geometry

Convert input surfaces to micropolygons



Thanks to Anjul Patney for these slides from his Siggraph Asia 2008 course.

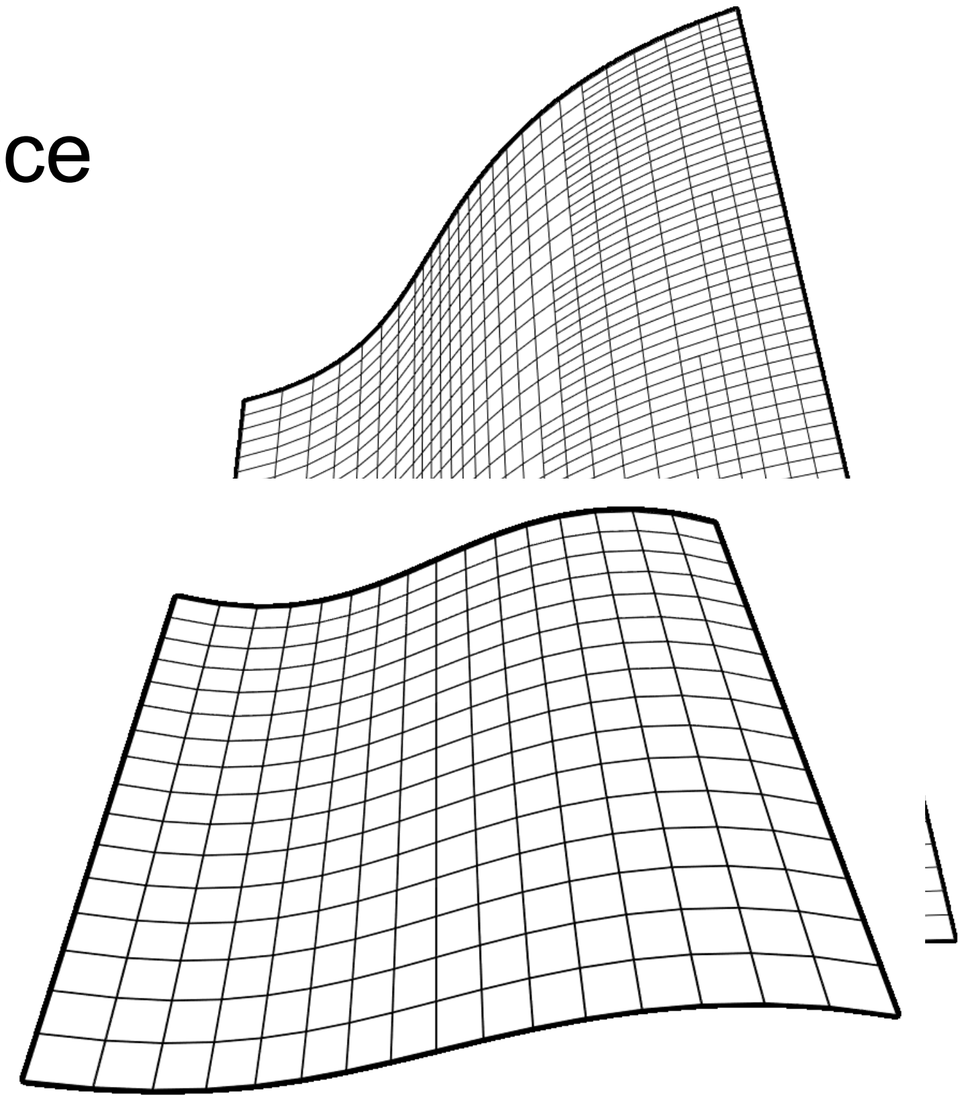
Input

- Higher-order surfaces
  - Bézier surfaces
  - NURBS
  - Subdivision surfaces
- Displacement-mapped
- Animated

Hand image courtesy: Tamy Boubekeur, Christophe Schlick

Task – Split and Dice

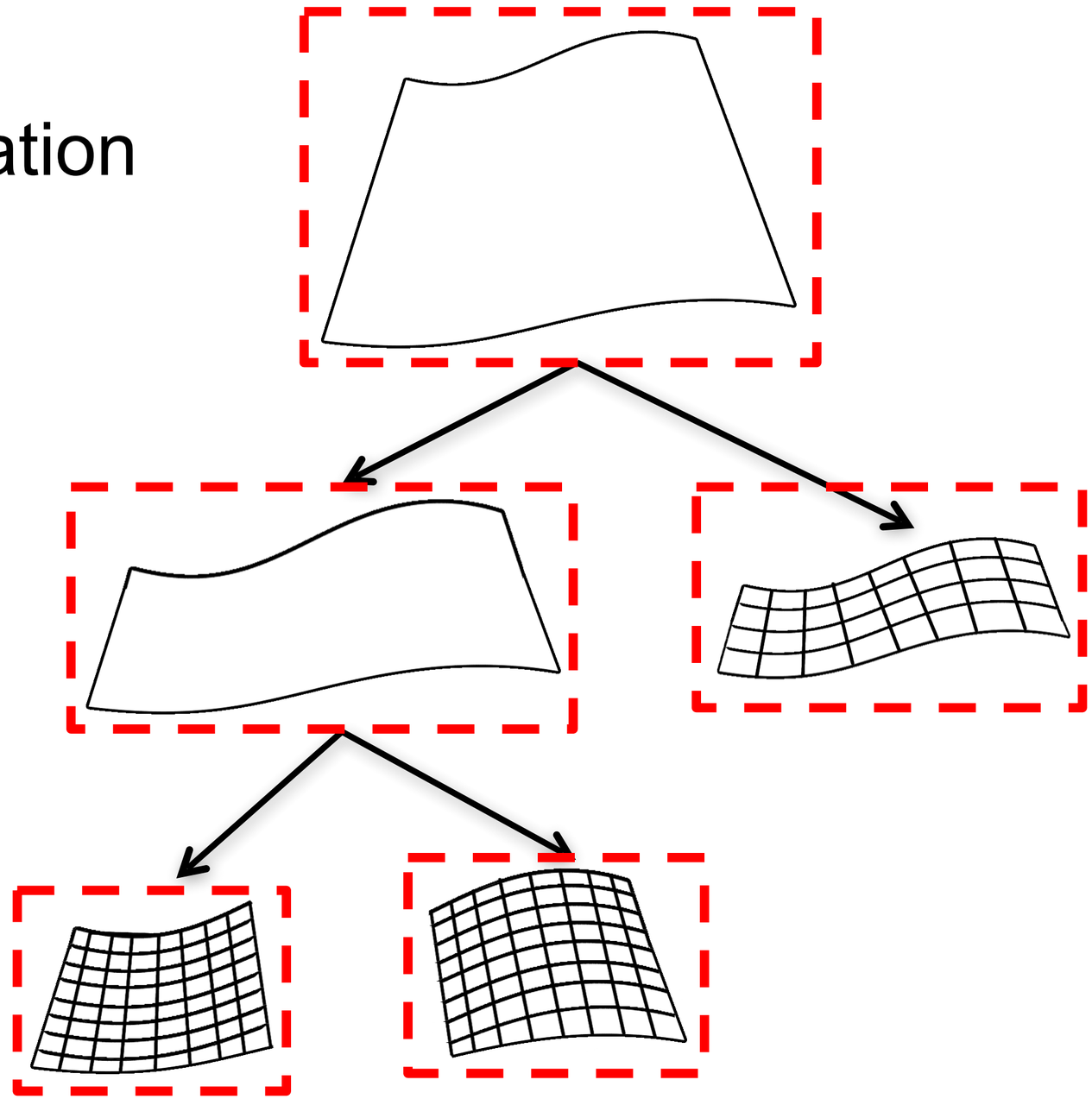
- Adaptively subdivide the input surface



- Tessellate when small enough
- Rinse and repeat

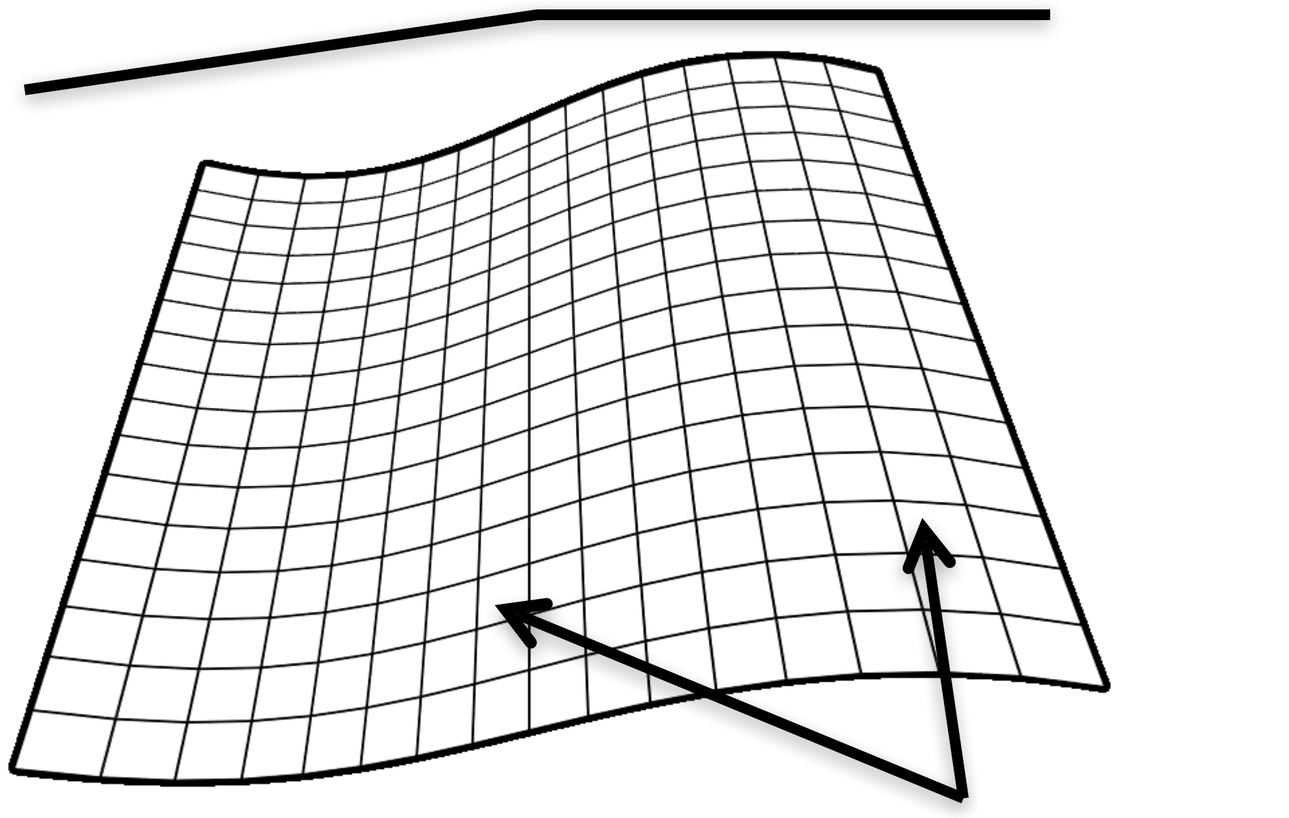
Challenges

- Recursive, irregular computation
  - Bad for parallelism, SIMD
- Too many micropolygons
  - Limited memory



Geometry Output – Unshaded Grids

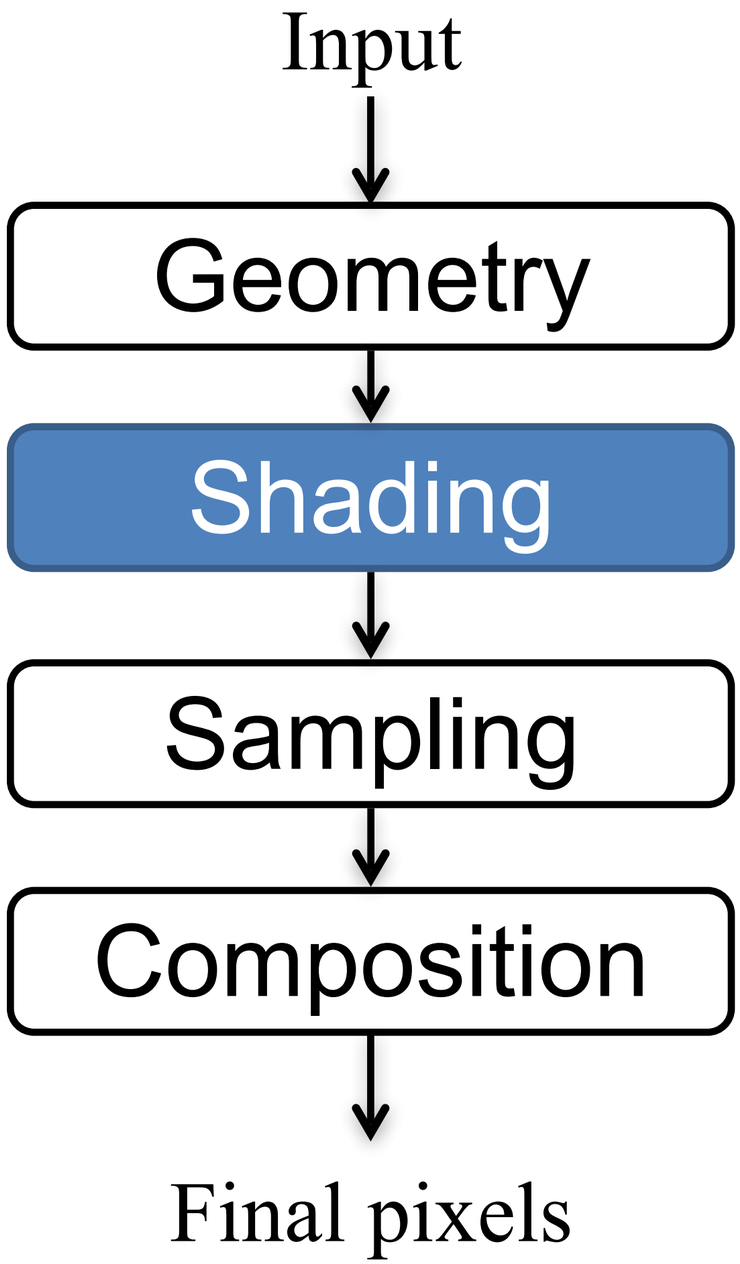
1 Grid



Micropolygons

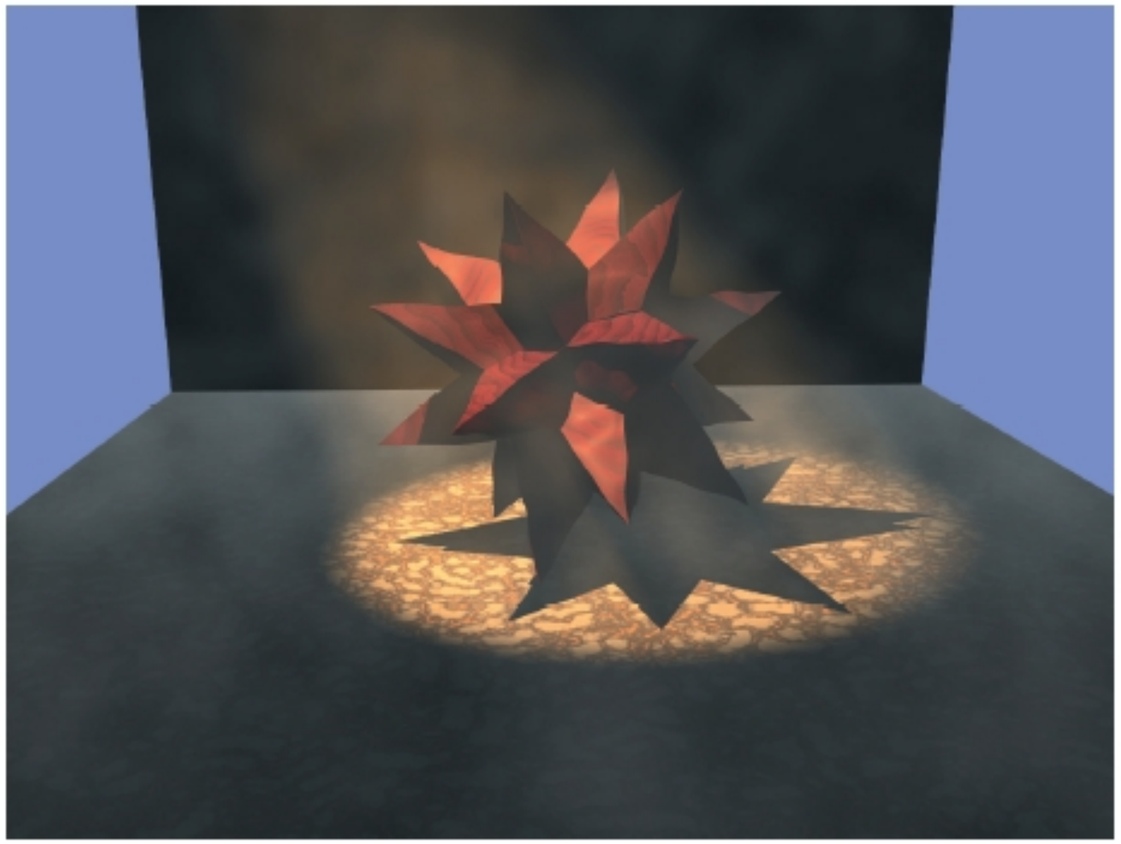
Step 2: Shading

Decide colors for grid micropolygons



Task

- Run shader(s) for each grid
  - Displacement
  - Surface
  - Light
  - Volume
  - Imager



- Good behavior
  - Highly parallel, SIMD friendly
  - Good locality behavior

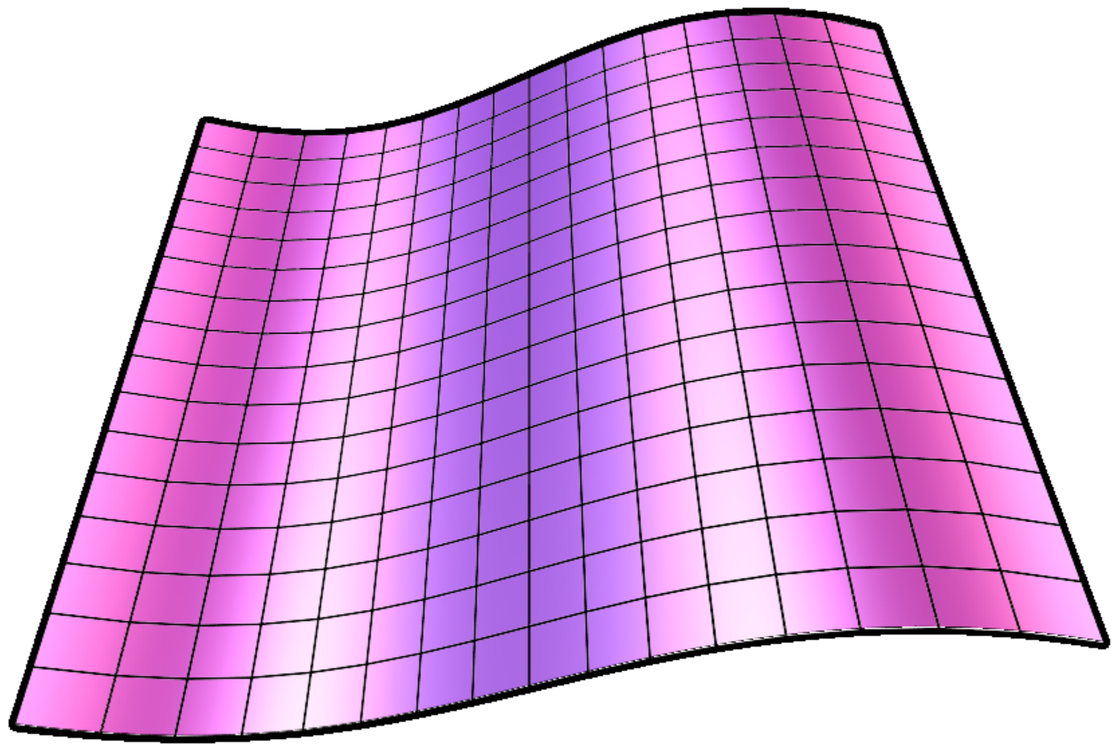
Image courtesy: Saty Raghavachary

Challenges

- Massively parallel is great
  - But is it good enough?

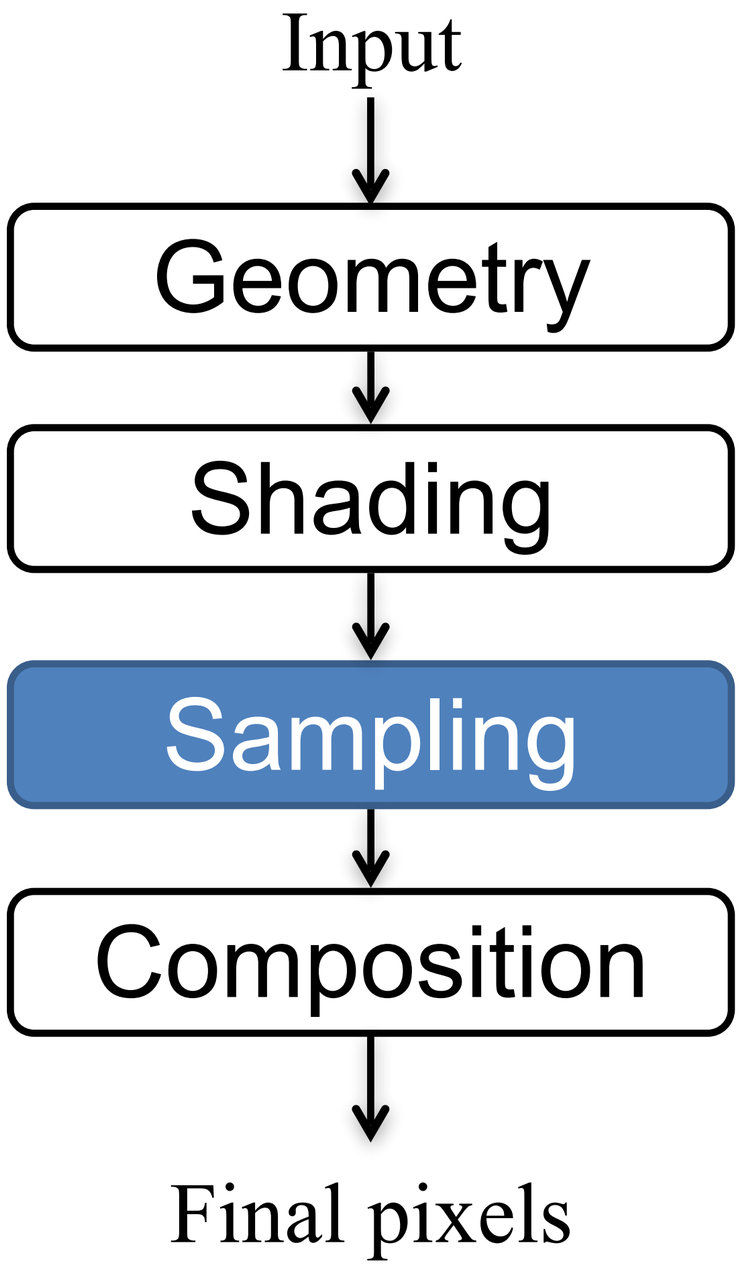
- Shaders can be complex
  - Too many instructions, conditionals
  - Global illumination
  - File I/O
  - Arbitrary texture fetches

Output – Shaded Grids

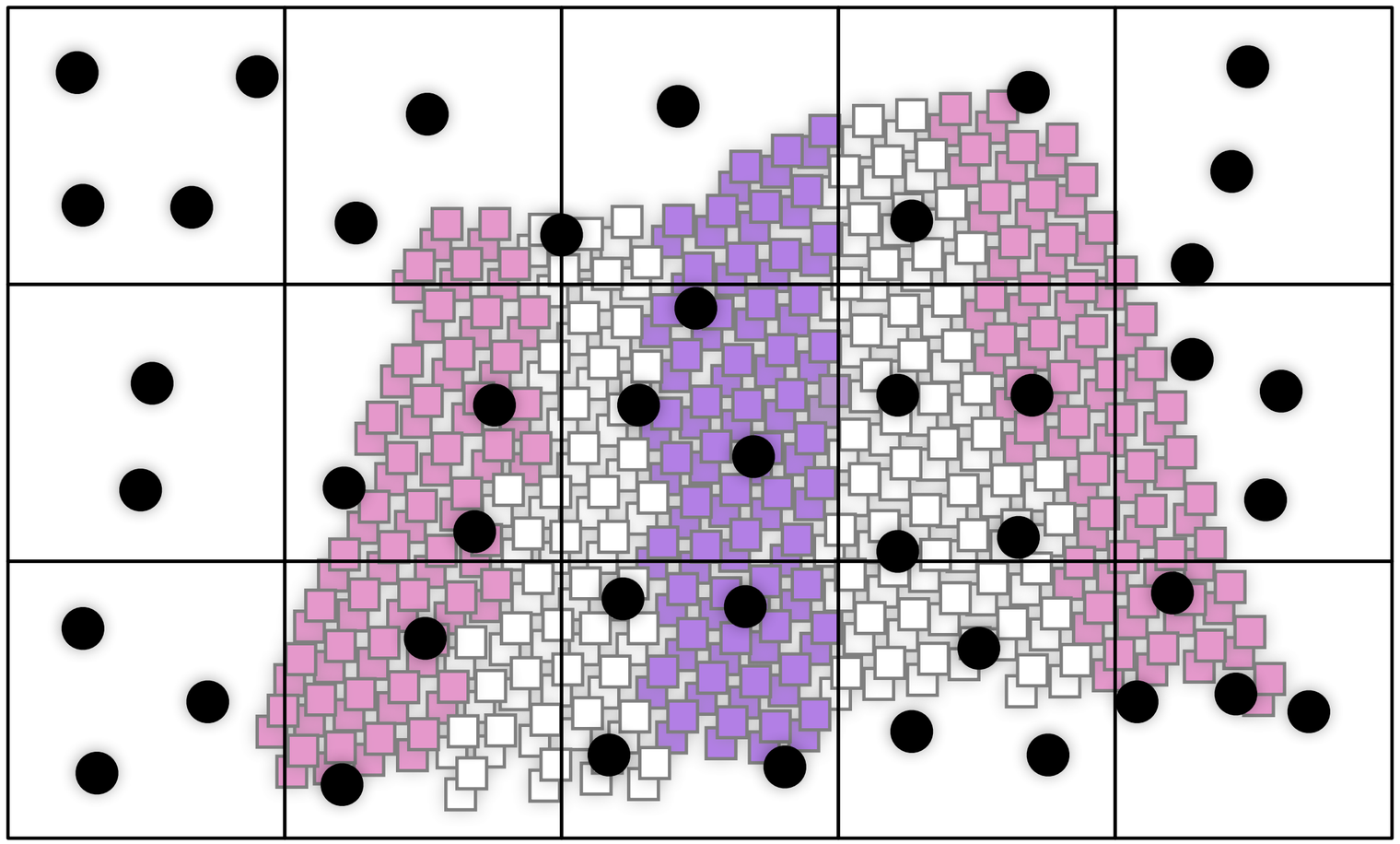


Step 3: Sampling

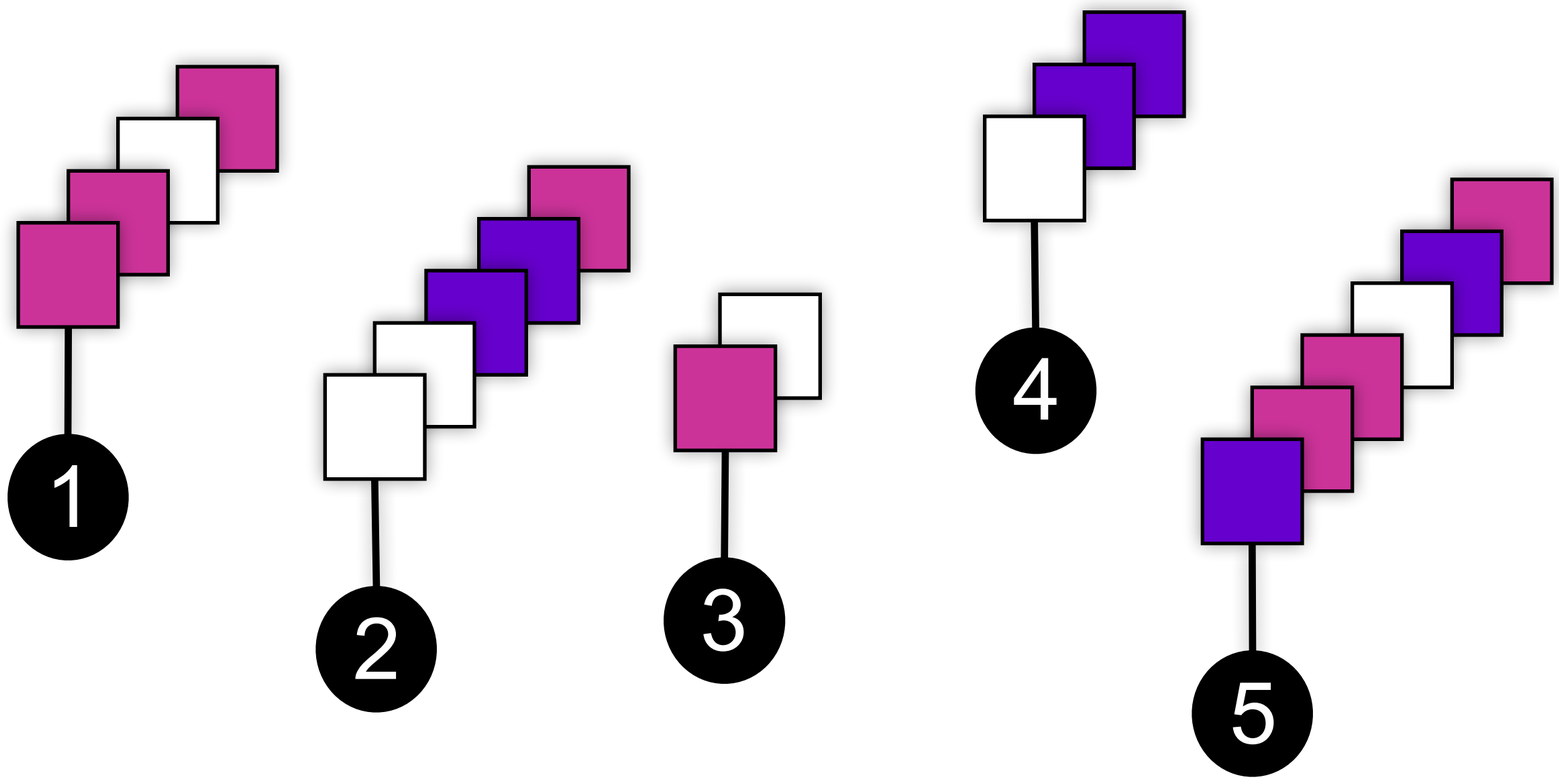
Collect stochastic samples of micropolygons



Task



Samples



Challenges

- Generate samples
  - Jittered grid
  - Parallel Poisson sampling [Wei 2008]

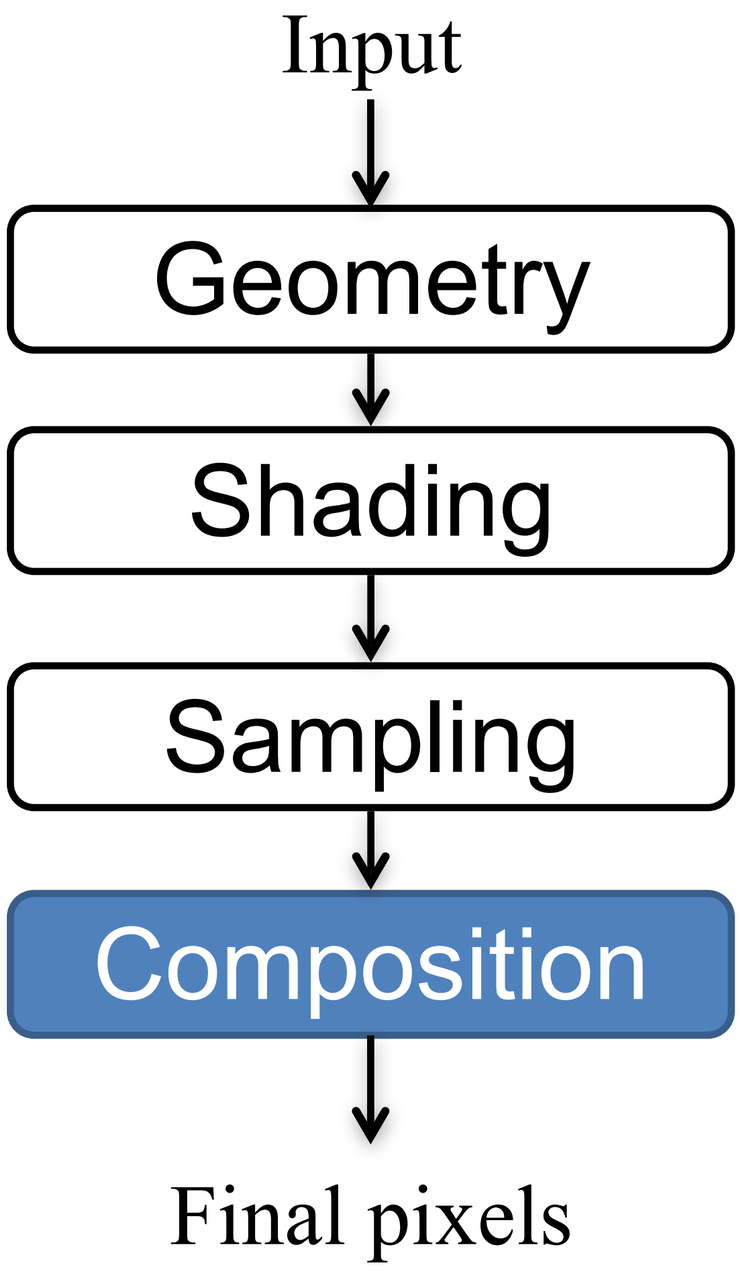
- For each sample, find all intersecting micropolygons
  - Raycast or Rasterize?

- Output: A (depth-sorted?) list of samples

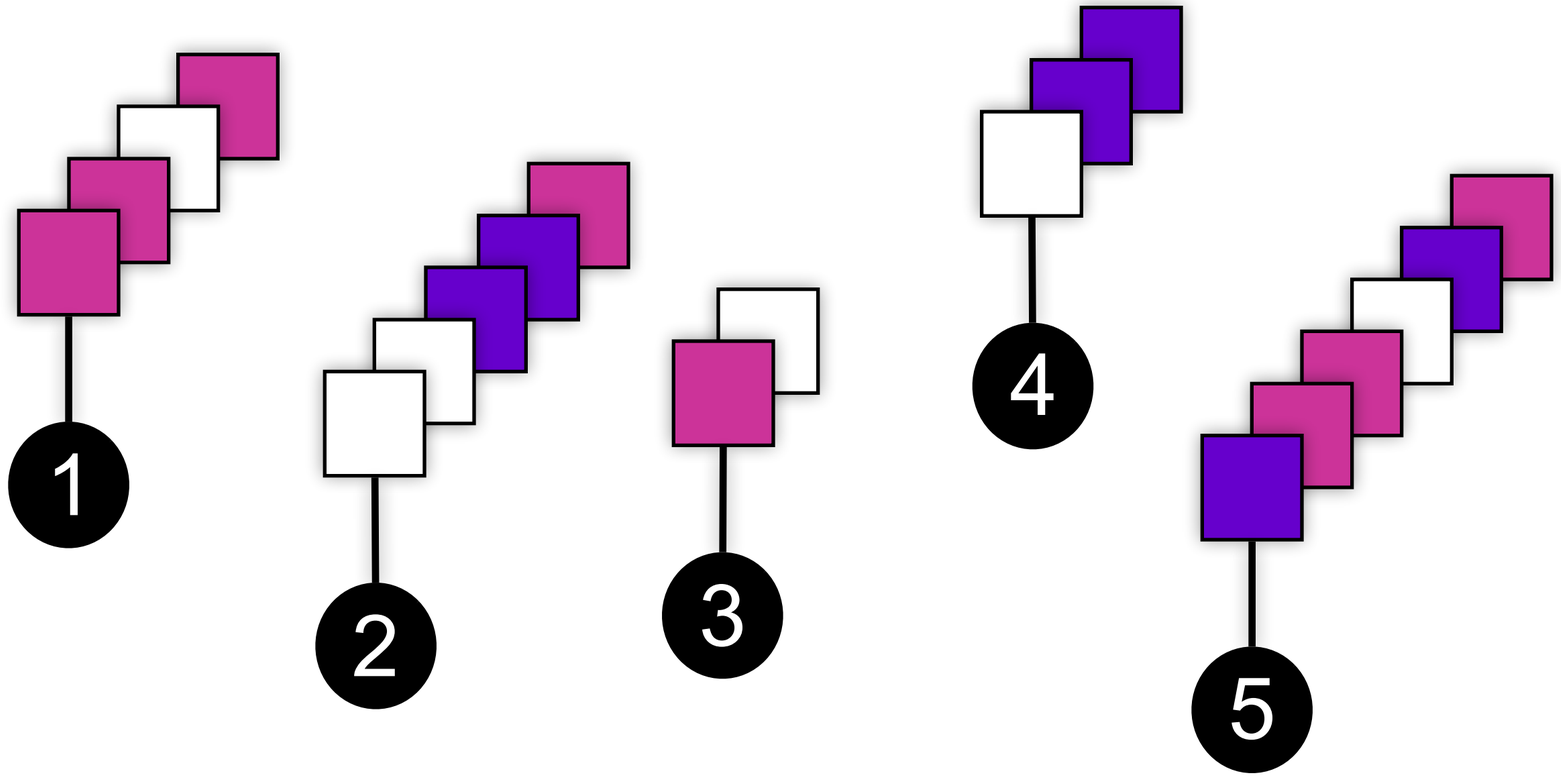
Step 4: Image Composition

Blend samples to get colors

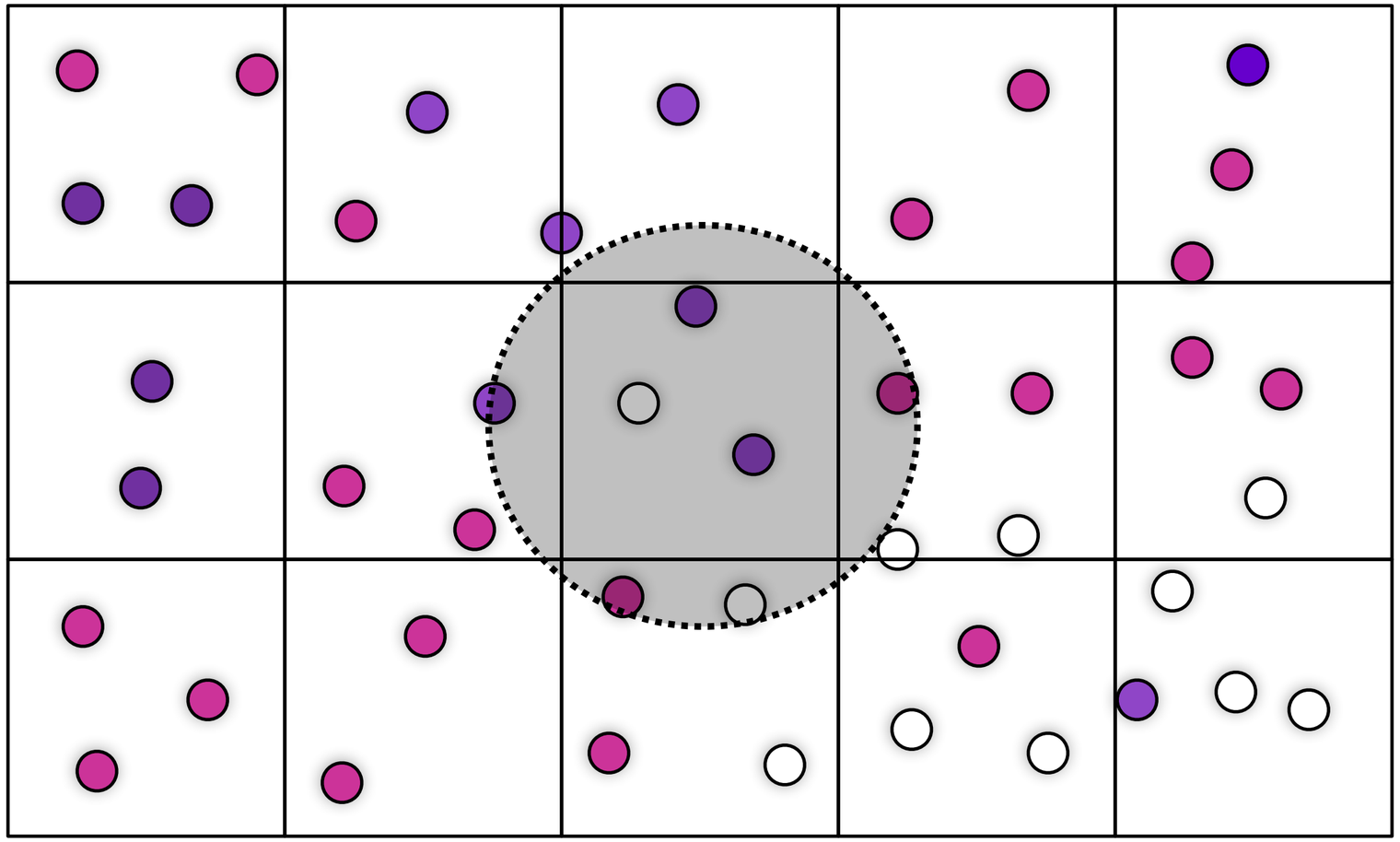
Combine colors to get pixels



Task 1: Blend



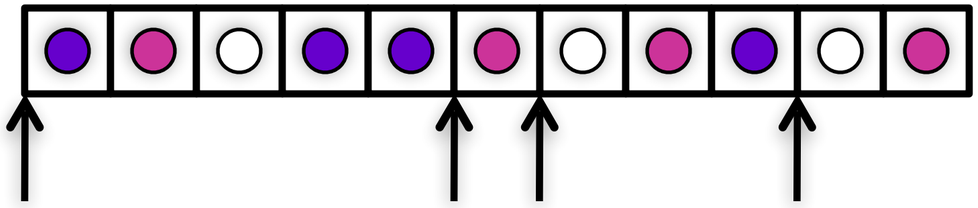
Task 2: Filter to get pixel colors



Challenges

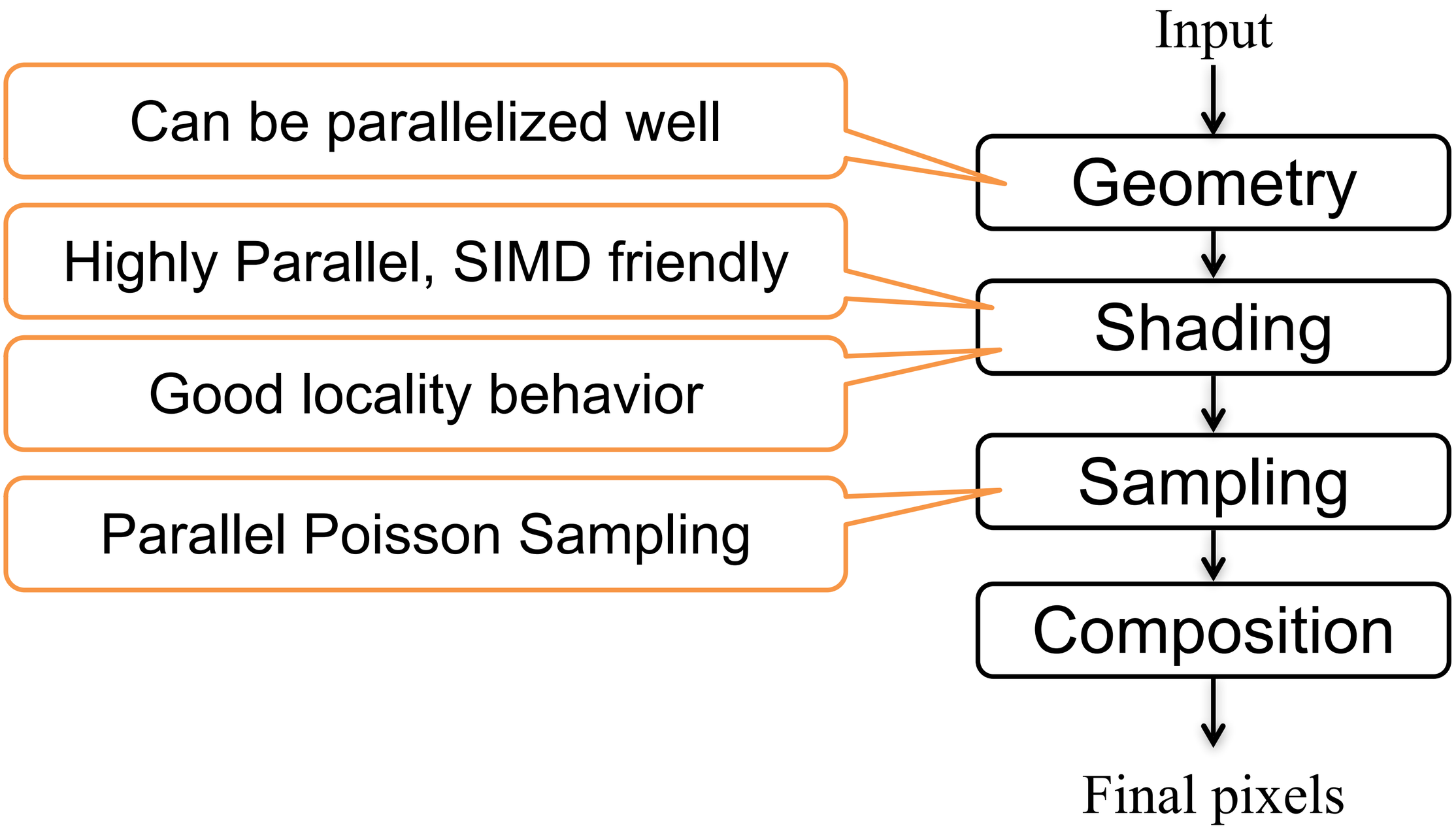
- Represent the irregular work-list
  - Traditionally: linked-list per sample (arbitrary size)

- Sort and Reduce
  - Unequal work-items

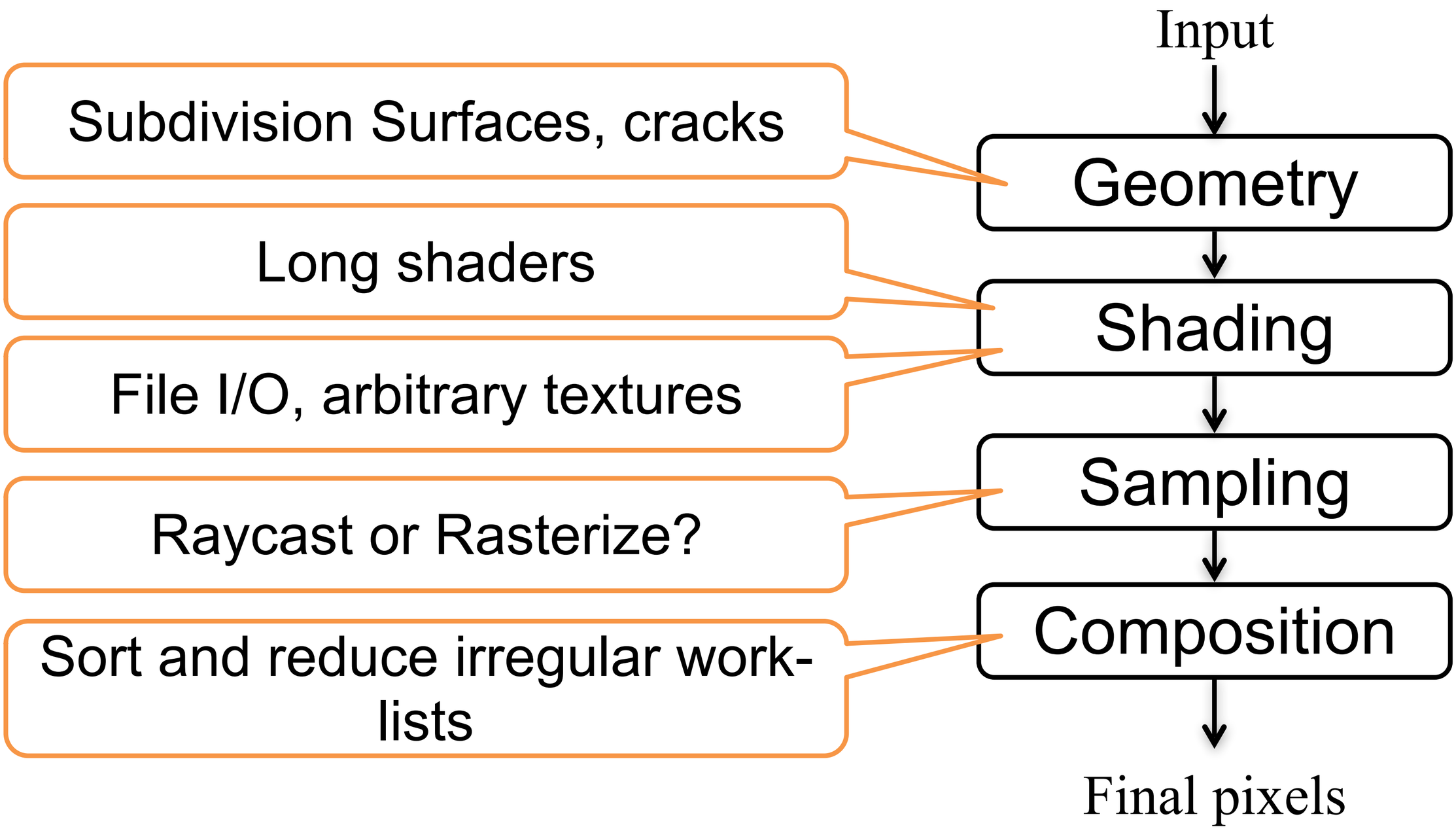


- Generate and apply filter kernels
  - Box
  - Gaussian

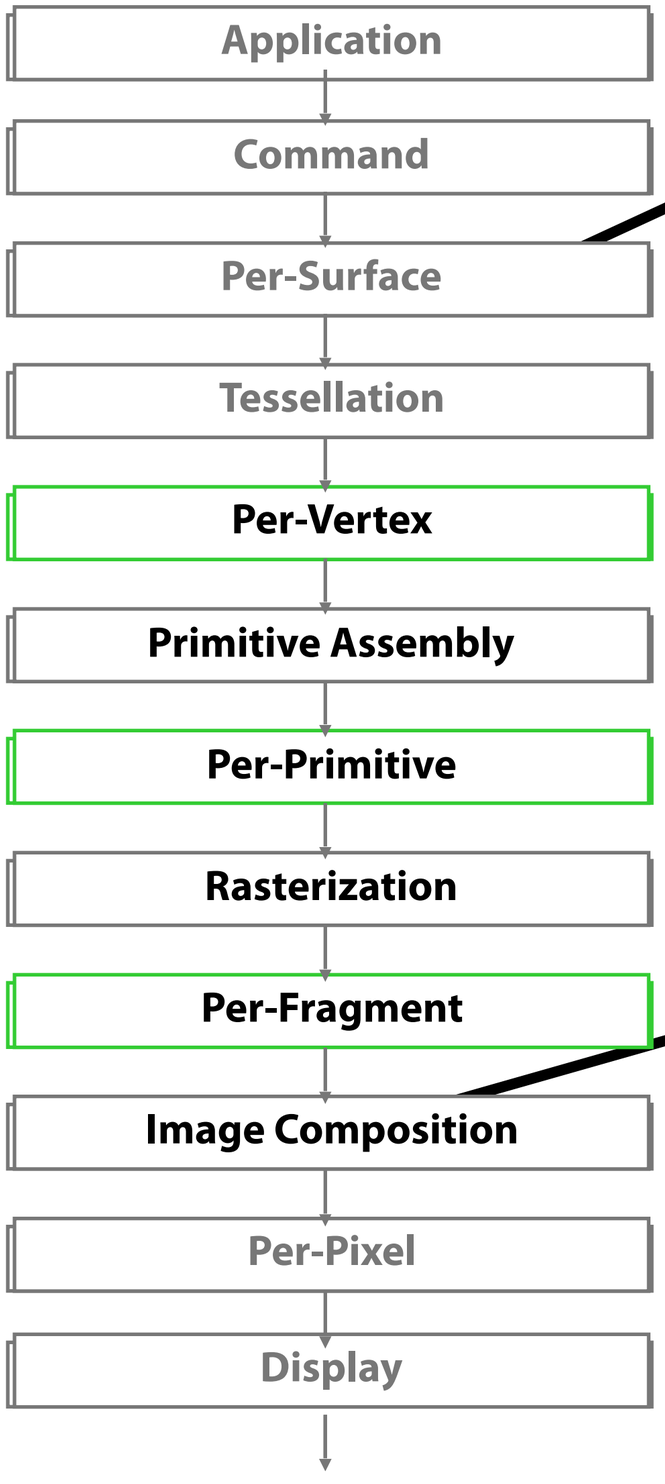
Summary: What is easy?

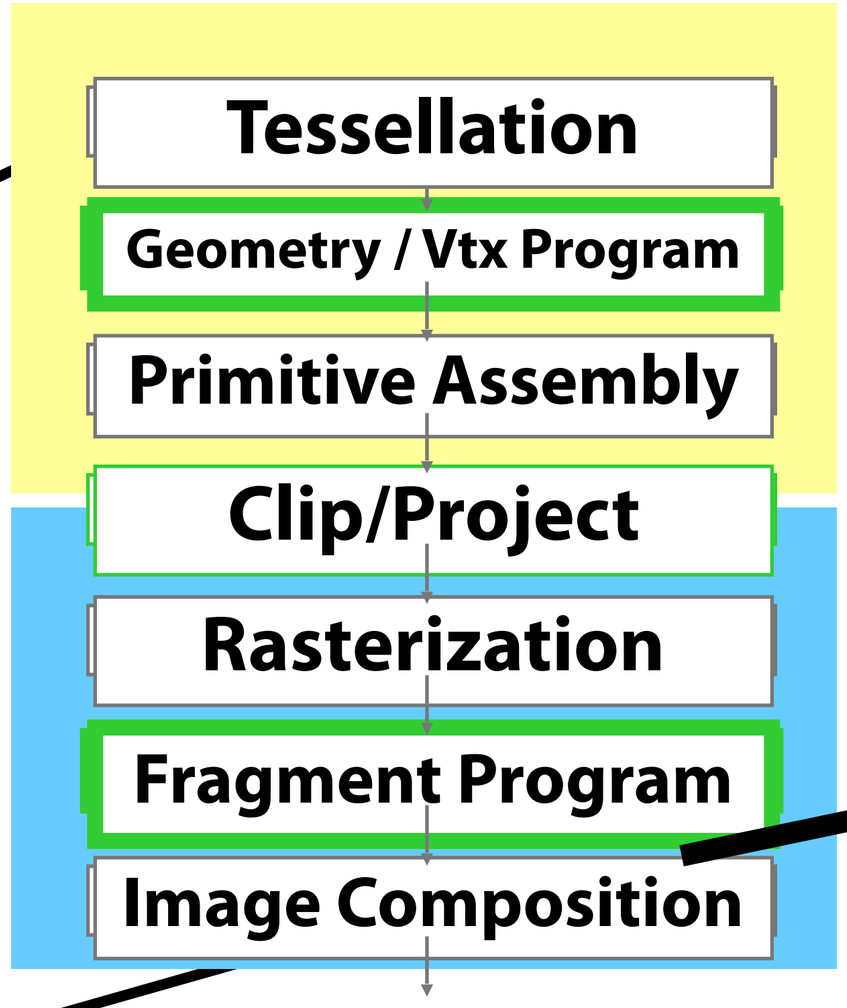


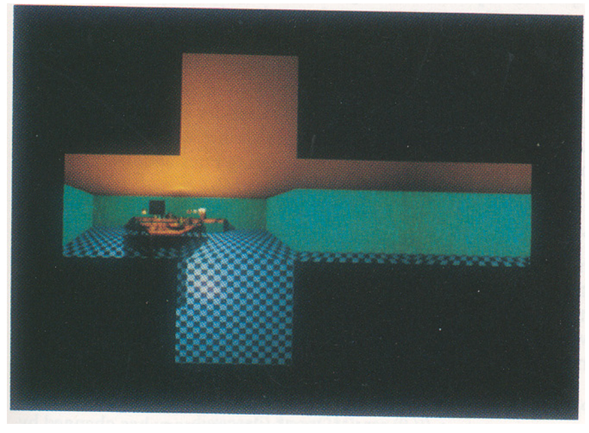
Summary: What is hard?

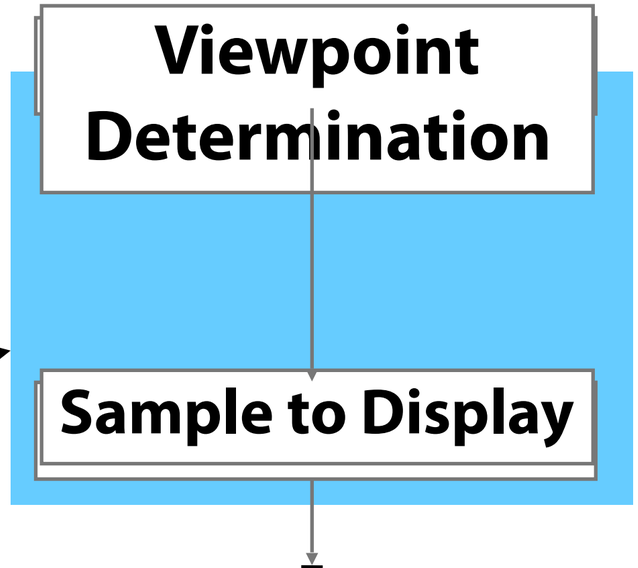


Address Recalculation Pipeline in Generalized Pipeline







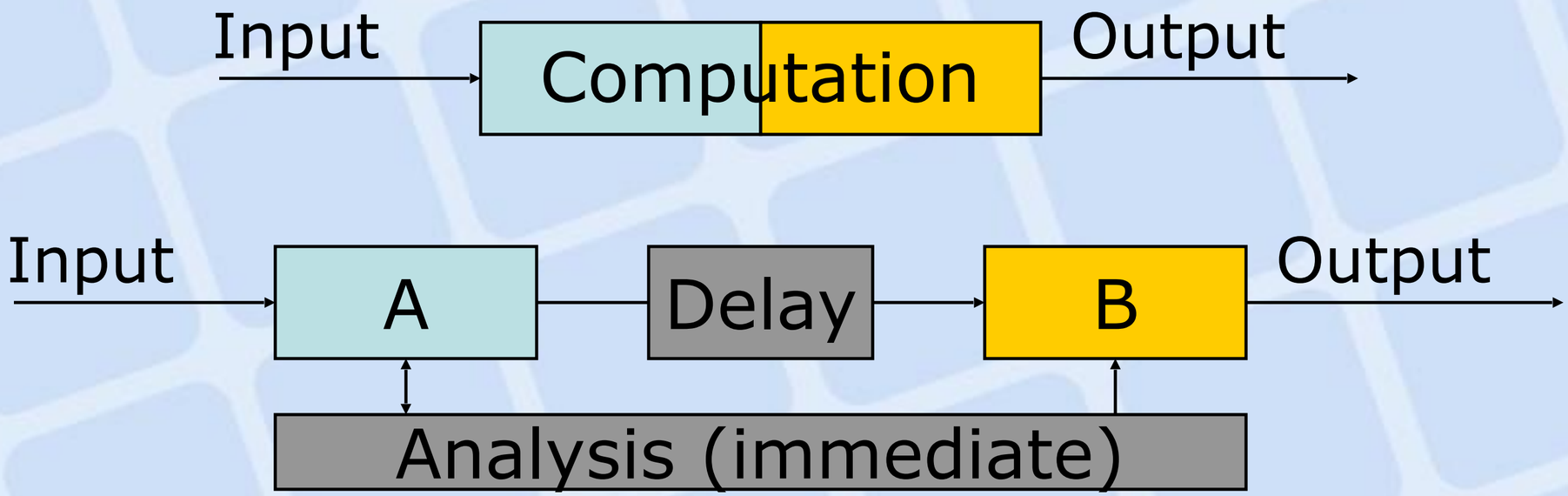


Priority rendering with a virtual reality address recalculation pipeline, Regan and Pose, Siggraph '94

Delay Streams for Graphics Hardware

- Timo Aila
- Ville Miettinen
- Petri Nordlund

Extending the Stream Processing Model



- Split computation into two separate parts
- Why?
  - peek into future (think about Tetris)
  - better decisions: avoid redundant computations

Delay Streams for Graphics Hardware

Introduction

Applications

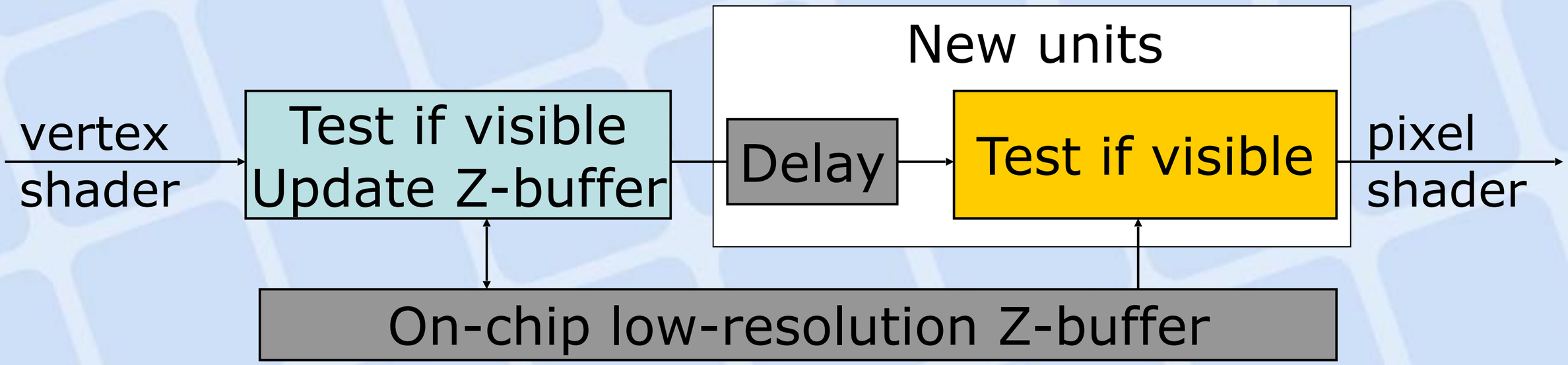
- Occlusion Culling
- Detecting Discontinuity Edges
- Order-Independent Transparency

Occlusion Culling

- Object-Level Culling
  - occlusion queries, available in hardware
  - requires substantial application assistance

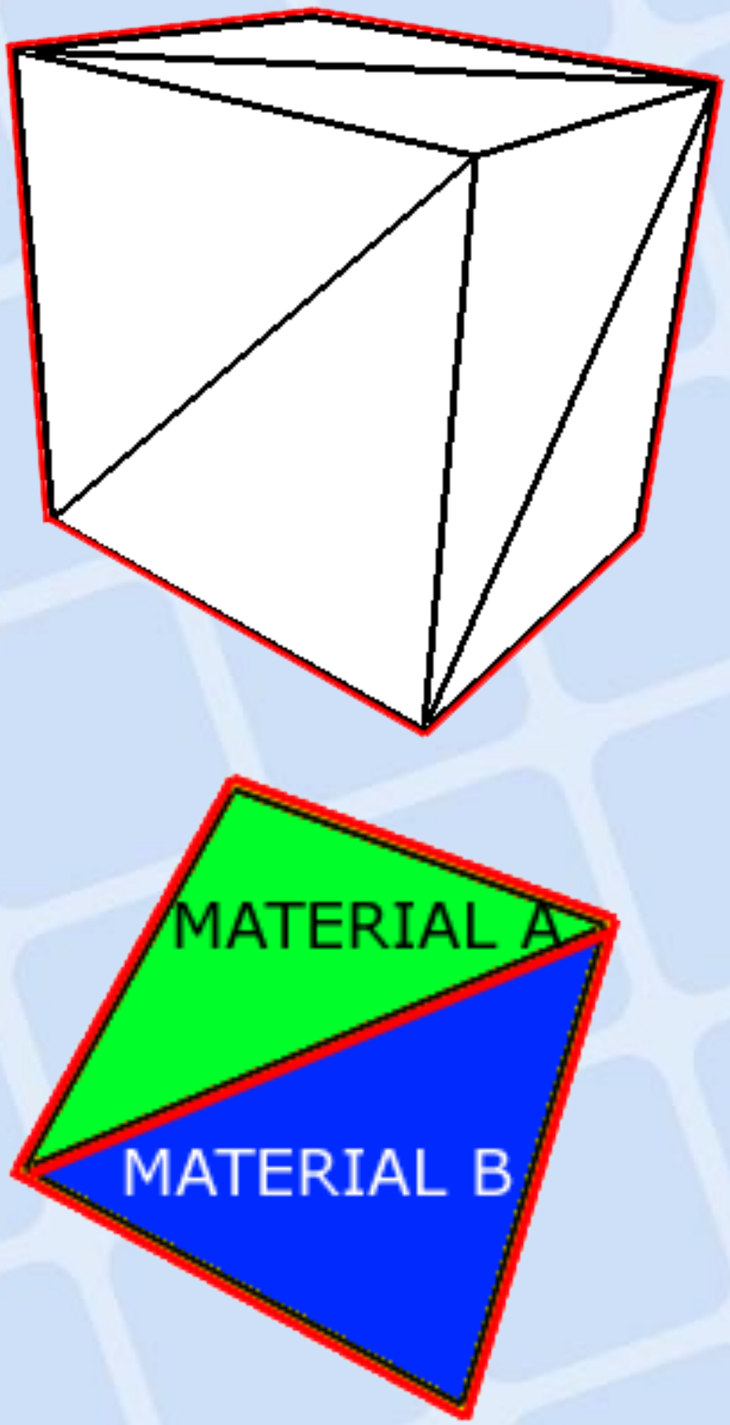
- Primitive-Level Culling
  - cull hidden triangles and blocks of pixels before pixel shading
  - used in most current graphics hardware
  - transparent to application

Idea: Delay Streams



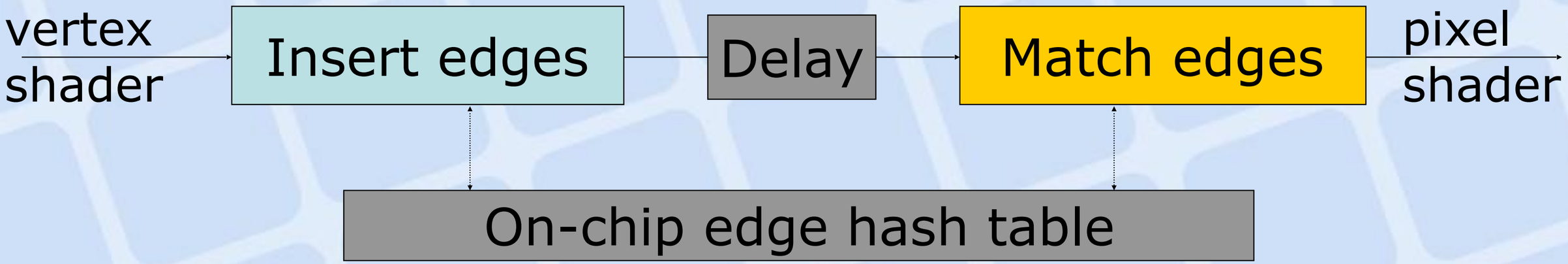
- Between vertex and pixel processors
  - sliding window of 50K-200K non-culled triangles
  - stores compressed vertices and render state changes
- Triangle rendered if visible after the delay
  - may thus be culled by primitives submitted **after** it
  - does **not** change the rendering order

Detecting Discontinuity Edges



- Discontinuity edges (red)
  - edges shared by front- and back-facing triangle
  - non-shared edges
  - edges at material boundaries
- Find using geometric hashing
  - vertex hash key formed from vertex attributes (position, color, texture coordinates..)
  - edge hash key formed from two vertex hash keys

Detecting Discontinuity Edges in Hardware



- Heuristic approach: detect most shared edges, classify rest as discontinuity edges
- Shared edge detected if both triangles in delay stream at same time
- Holds 3K tris; exact results on many models
- Misclassifications rare and not critical

No antialiasing

4x4 supersampling

Discontinuity edges (DEs)

DE supersampling

PCU: The Programmable Culling Unit

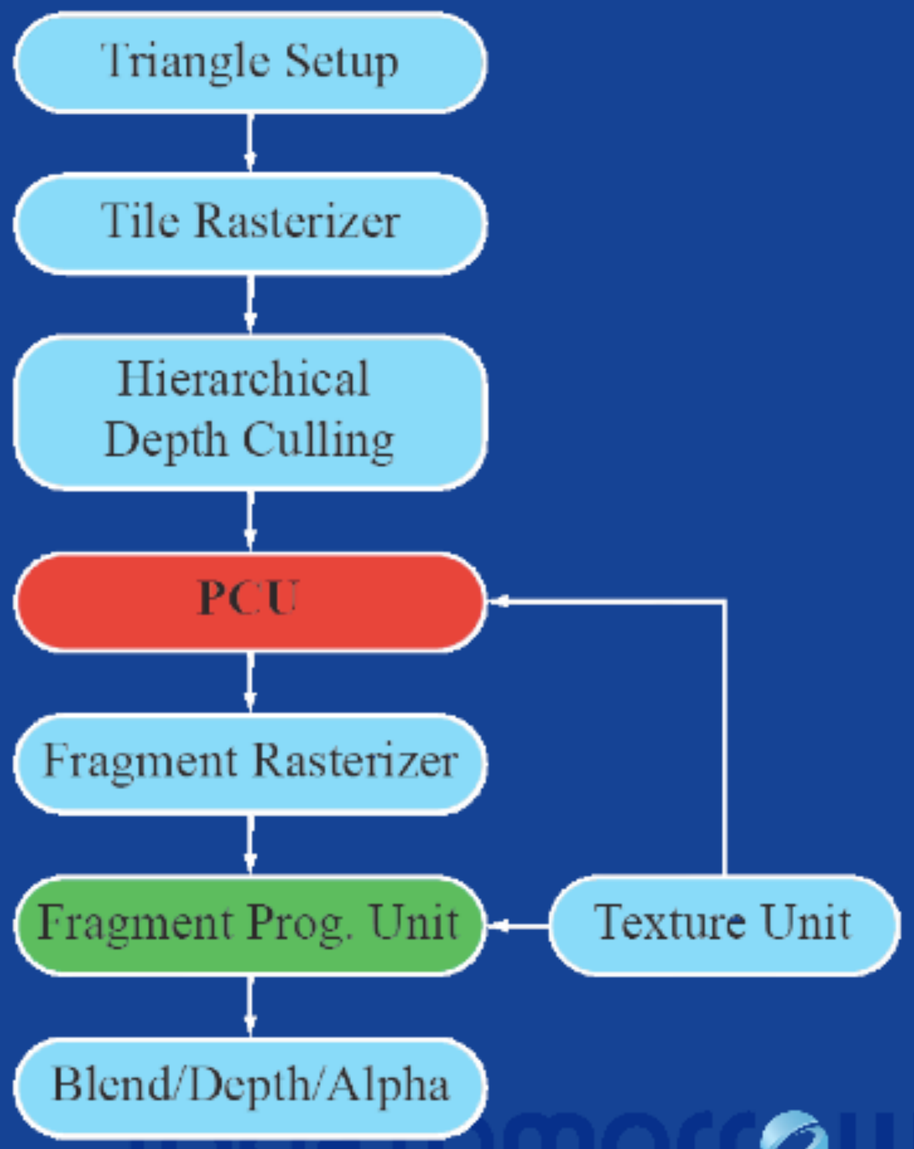
SIGGRAPH2007

Jon Hasselgren Tomas Akenine-Möller

Slides shamelessly downloaded from Jon’s website. Thanks Jon!

The programmable culling unit (PCU)

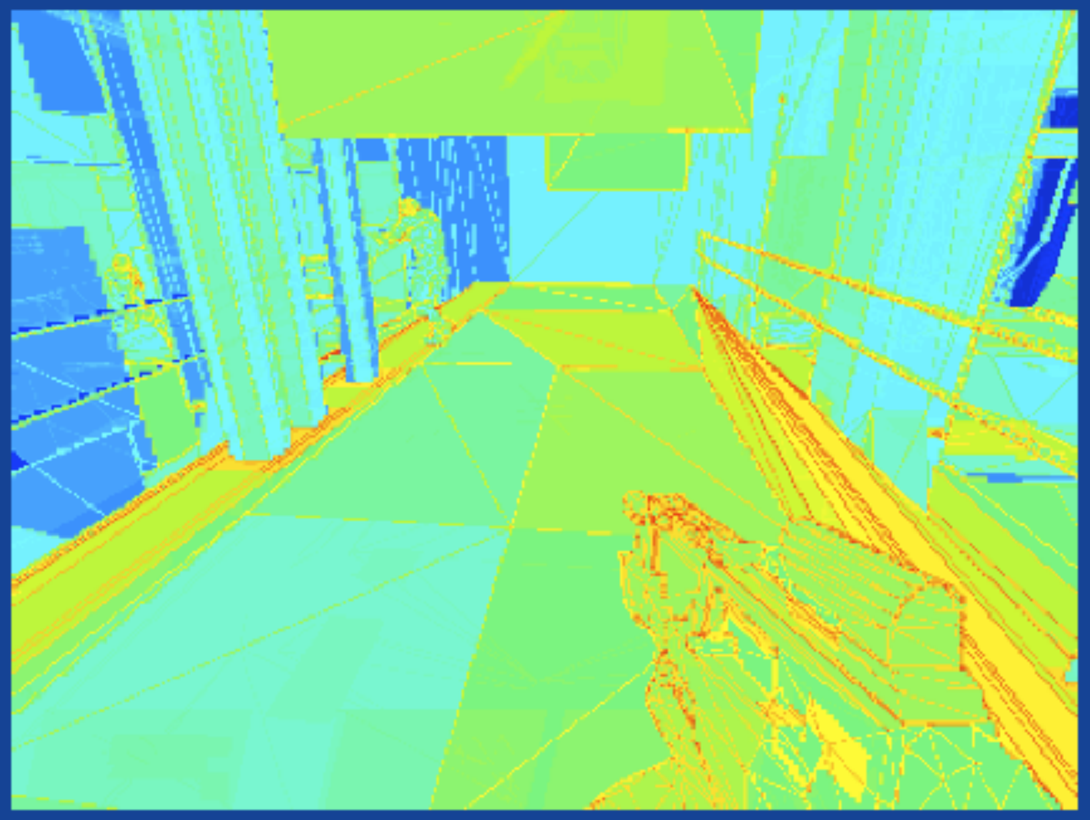
- We want to add hardware support for programmable culling
- New unit in the pipeline
  - Operates on tiles of pixels
  - Executes a culling program
  - Cull or process the tile



Does it work in practice?

Quake 4

Normal Rasterization



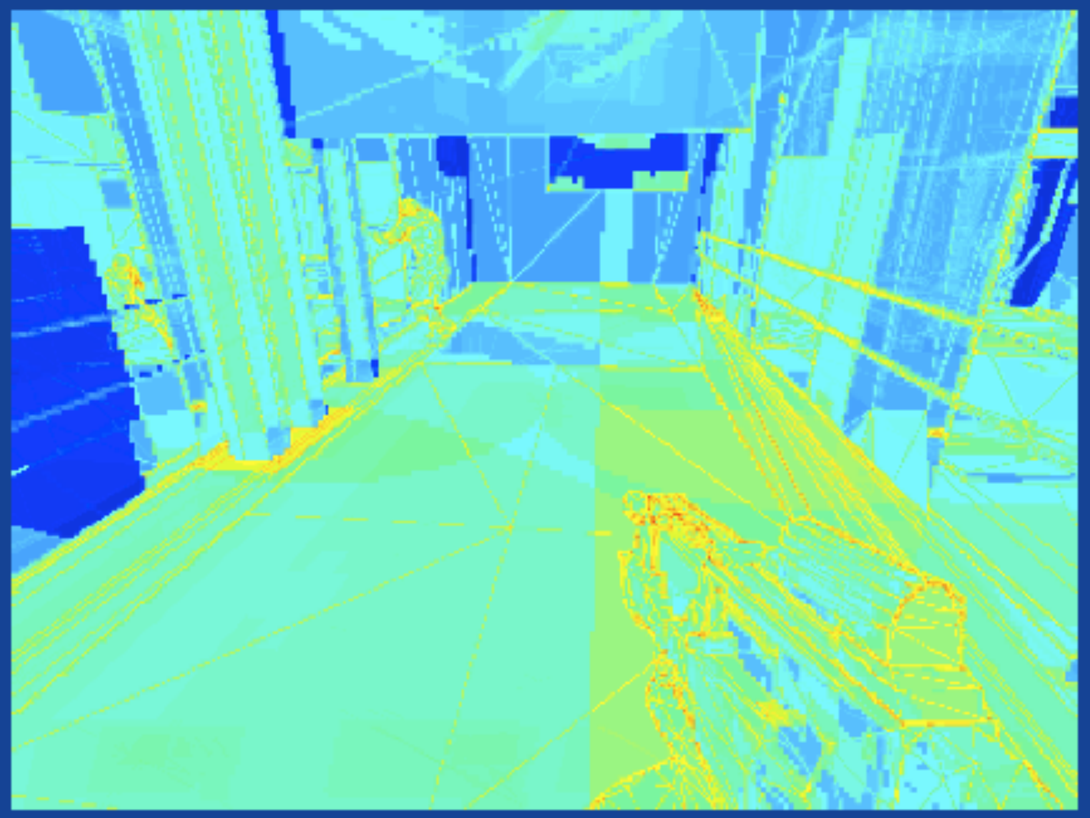
45 M Instructions

Does it work in practice?

Quake 4

Exact same result

Programmable Culling



30 M Instructions

Example: Details

- Computes diffuse pixel lighting
  - Dot product for diffuse lighting
  - Texture for material constant
  - KIL instruction to terminate ”backfacing” fragments

DP3	d,**n**,**l**
KIL	$d < 0$
TEX2D	c, t0, r1
MUL	out.col, d, c

Culling with the fragment program

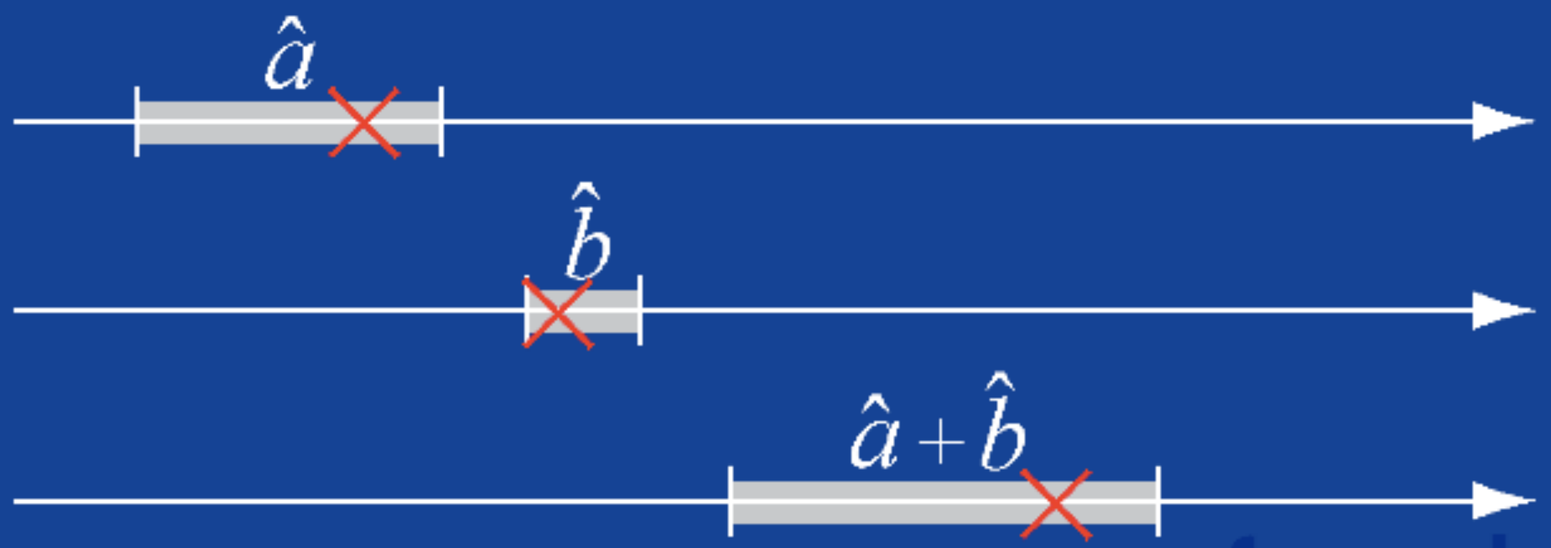
- Prove a KIL instruction is triggered for all fragments in a tile
  - Prove that d < 0 for all fragments
  - Need to be able to evaluate the dot product per tile

DP3	d,**n**,**l**
KIL	$d < 0$
TEX2D	c, t0, r1
MUL	out.col, d, c

Bounded Arithmetic

- We use interval arithmetic [Moore66] to evaluate the shader over a whole tile.
  - *All instructions* are redefined on intervals

$$\texttt{ADD}\ c,a,b \iff c = a+b \qquad \texttt{ADD}\ \hat{c},\hat{a},\hat{b} \iff \hat{c} = \hat{a}+\hat{b}$$

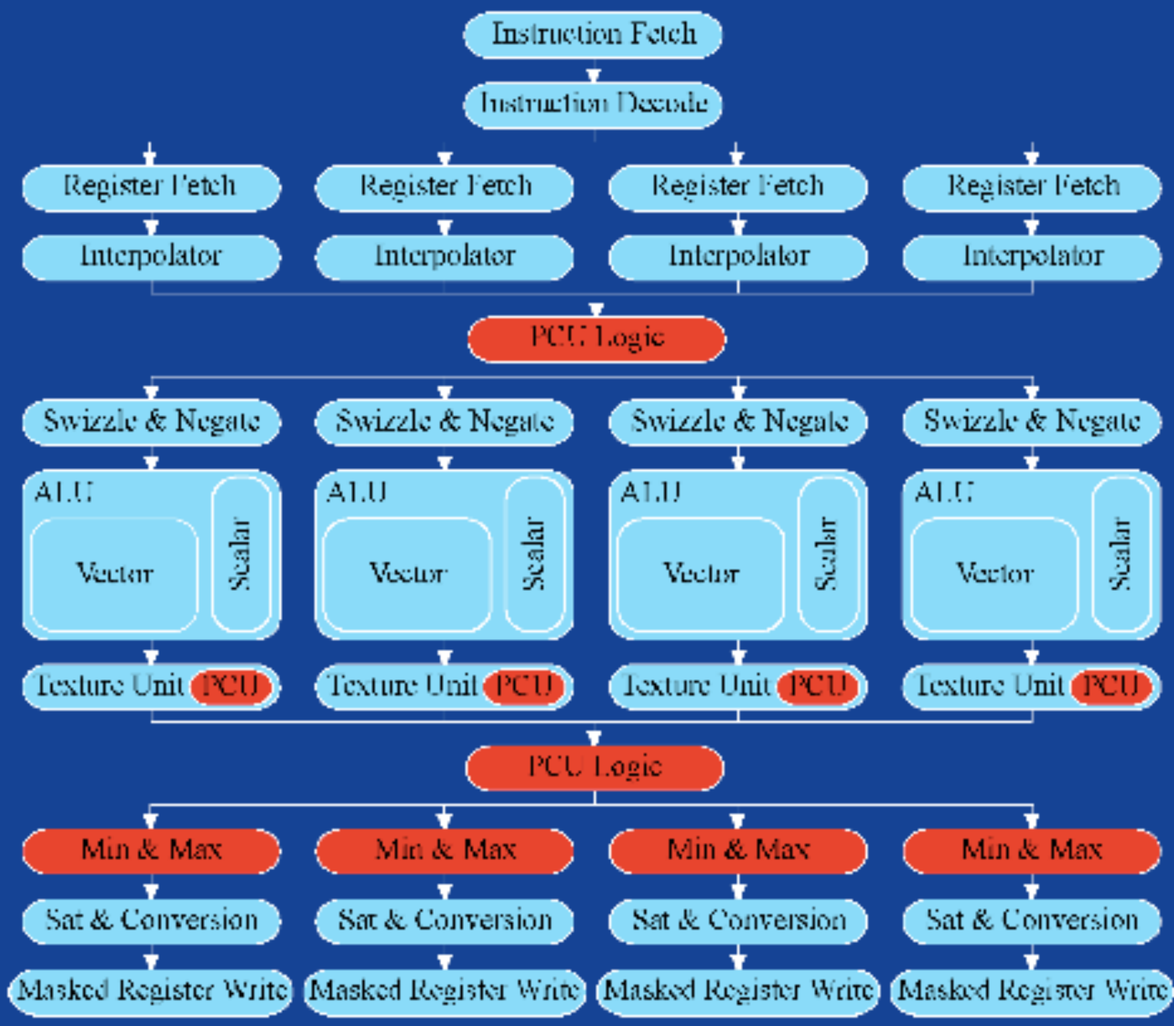


Culling algorithm

- First we compute bounding intervals for all varying (interpolated) program inputs
- Next we execute the fragment program using interval instructions
  - When a KIL is executed, we can cull the entire tile if the condition is unambiguously true

Hardware Summary

- Unified shader unit
  - Cull programs
  - Fragment programs
  - Simple to extend
- 10% extra hardware in our implementation
  - Many possible tradeoffs
  - Re-use existing resources



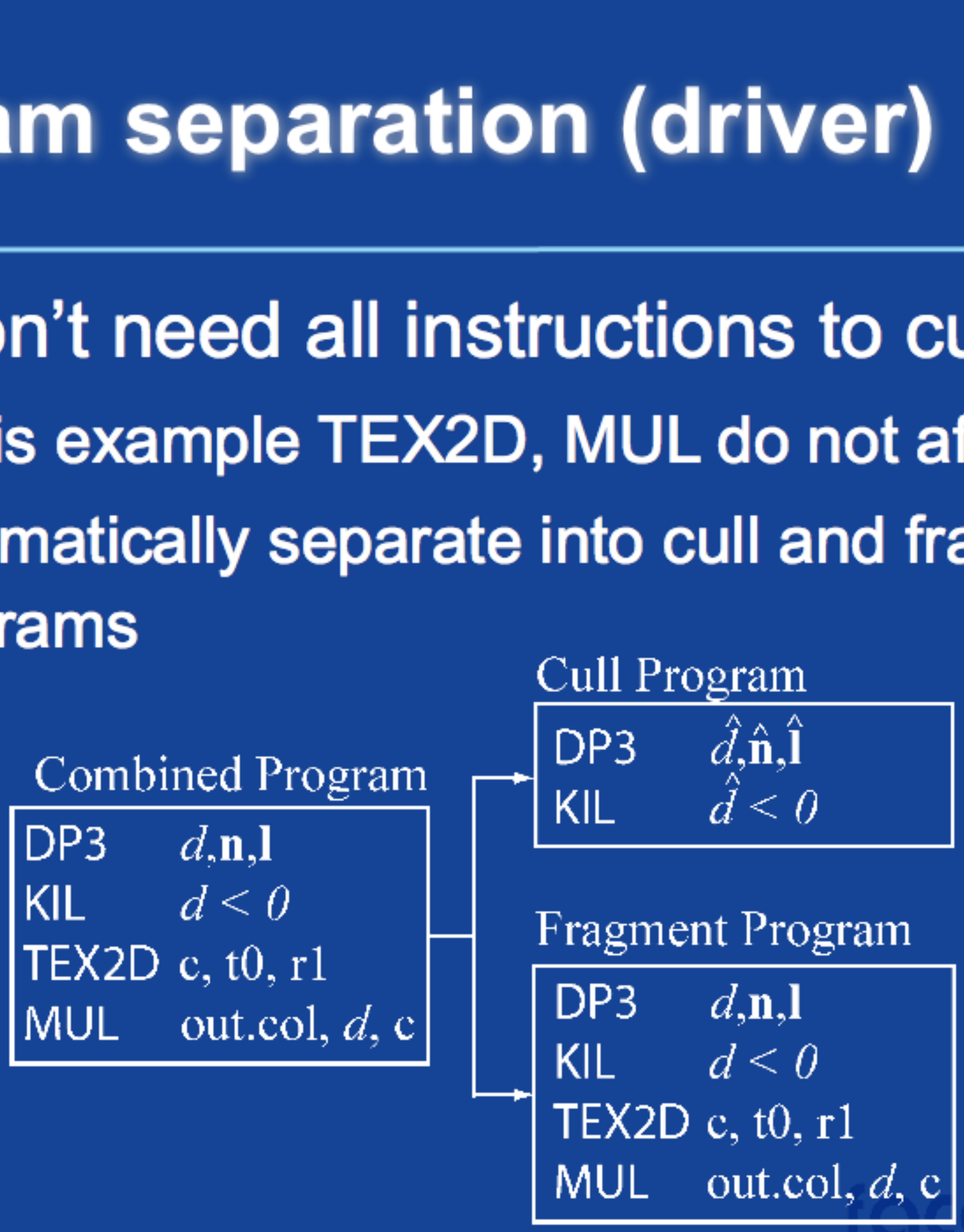
Driver optimizations

- In this form, the PCU is fully functional
  - But we execute the exact same program for fragment processing and culling
  - Many instructions will not contribute to culling

- We perform a program separation (optimization) process
  - Done by the driver when compiling the program

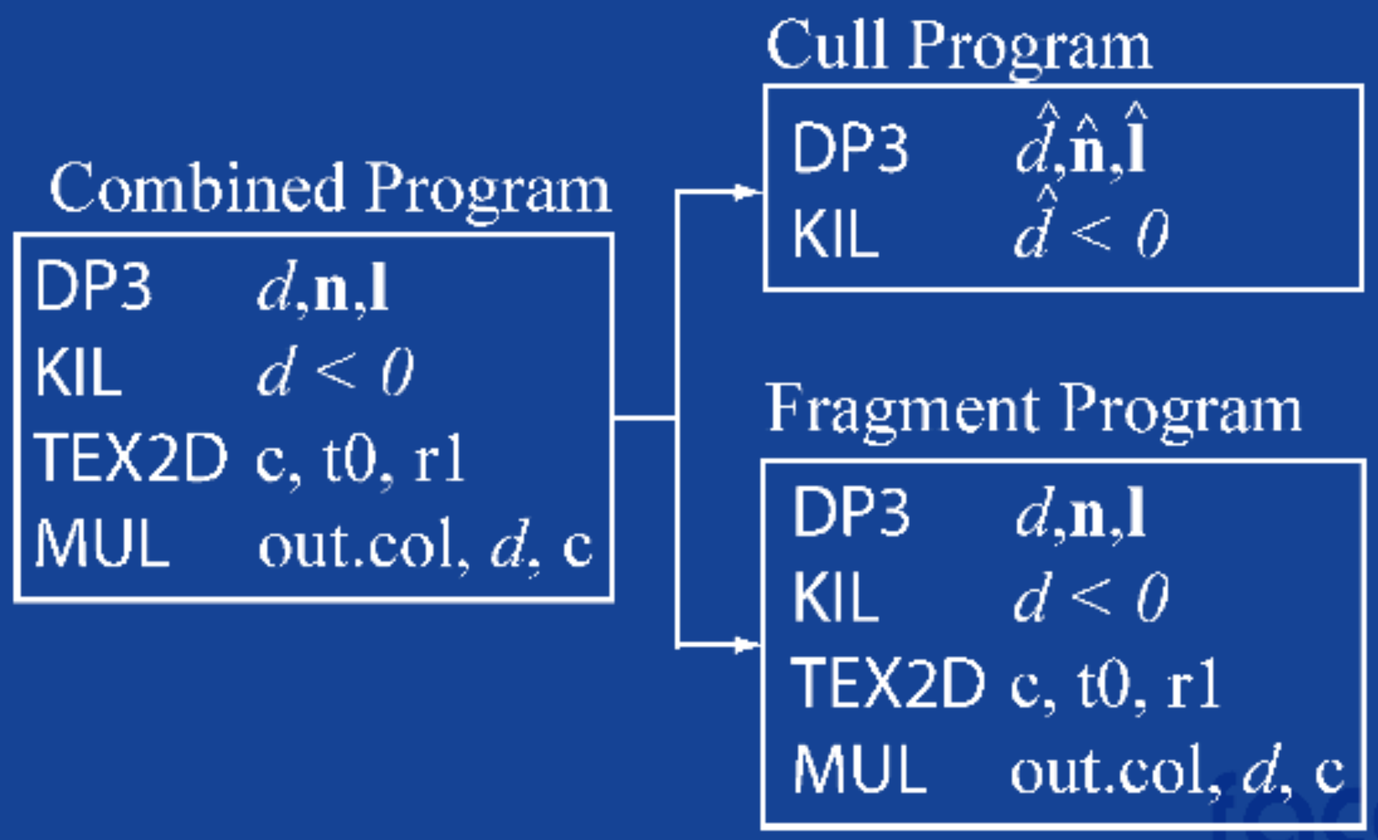
Program separation (driver)

- We don’t need all instructions to cull a tile
  - In this example TEX2D, MUL do not affect culling
  - Automatically separate into cull and fragment programs



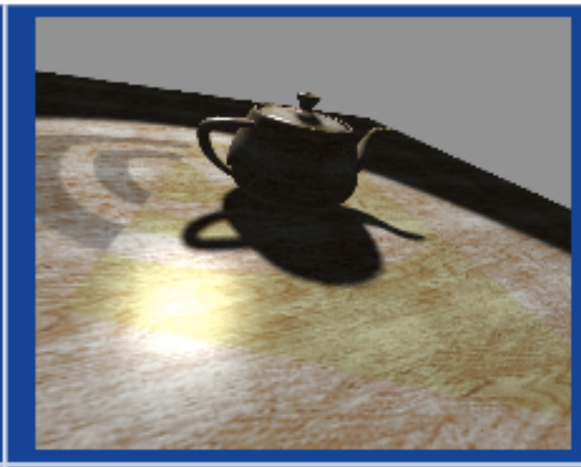
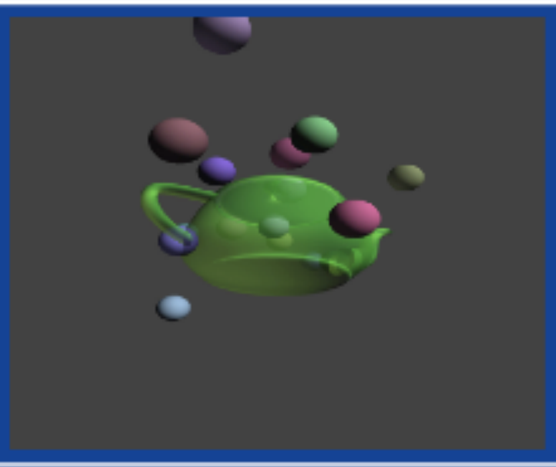
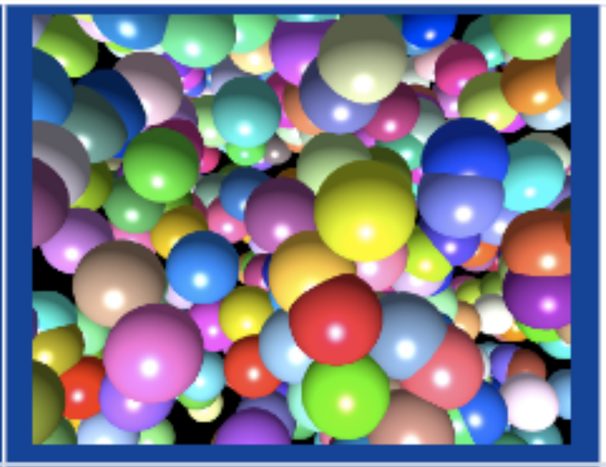
Dead Code Elimination (driver)

- Mark the following as live code, and do DCE
  - Fragment program: KIL, color, depth output
  - Cull program: KIL



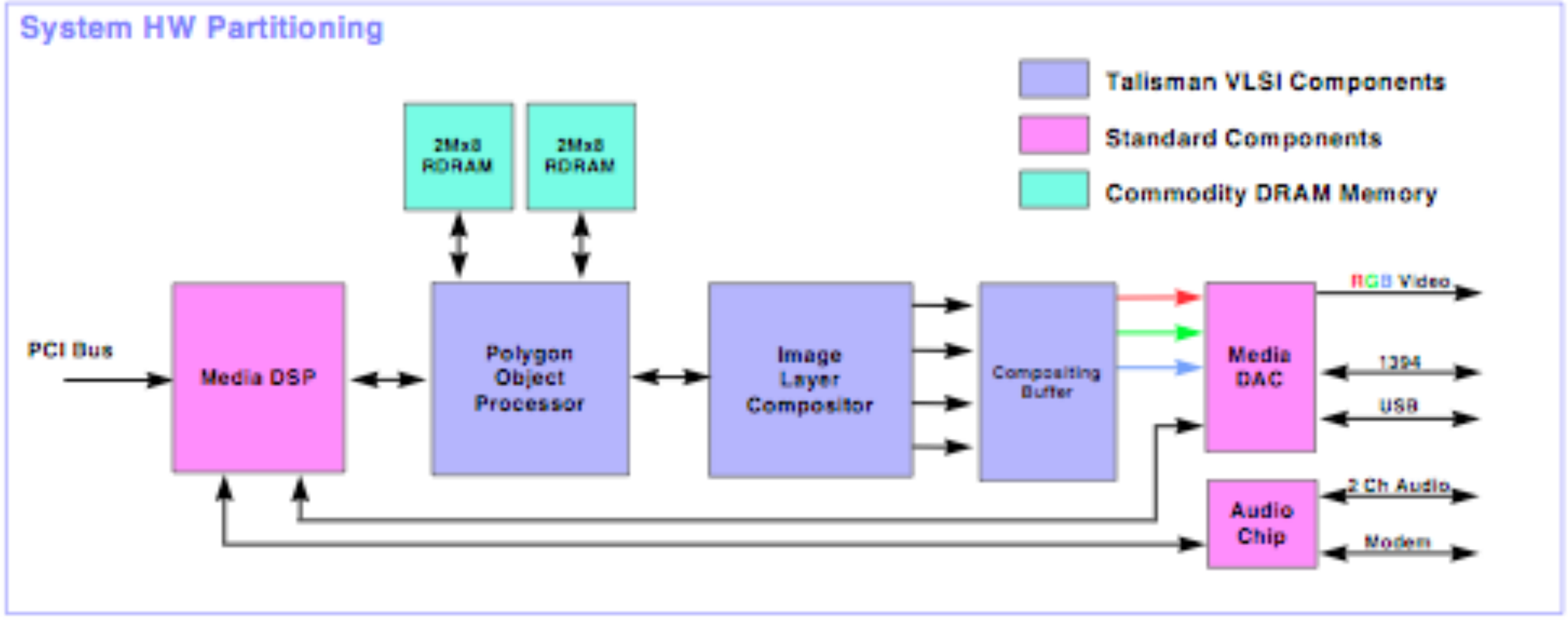
Summary

- Hardware unit for fully programmable culling
  - Simple and flexible implementation and usage
  - Significant performance improvement for relevant applications

			
1.4x / 1.5x	2.1x	1.6x	2.0x

Microsoft Talisman

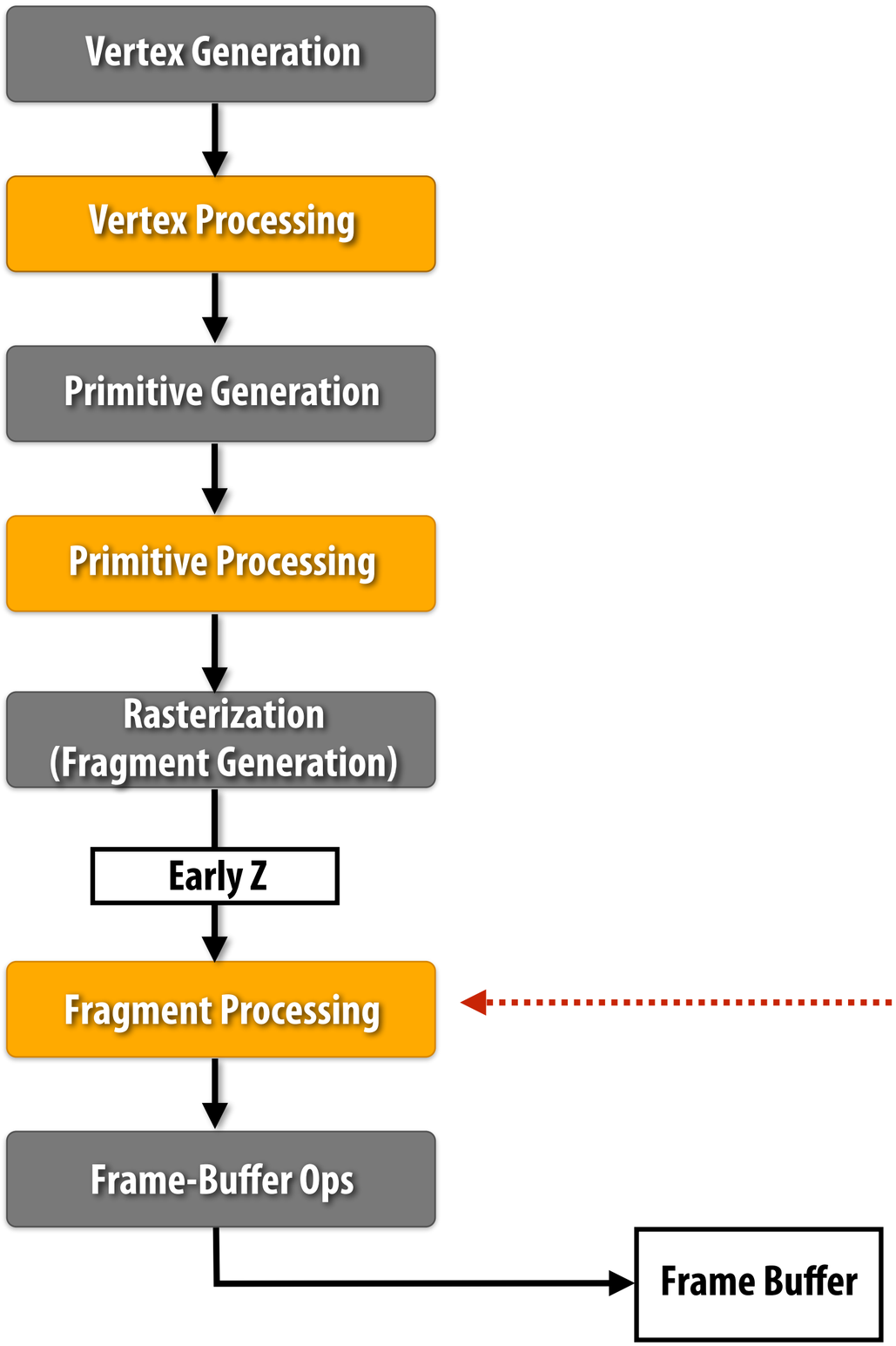
- Composited image layers with full affine transform
- Image compression
- Chunking
- Multi-pass rendering



Deferred shading

- Popular algorithm used by many modern real-time renderers
- Idea: restructure the rendering pipeline to perform shading after all occlusions have been resolved
- Not a new idea: present in several classic graphics systems, but not directly implemented by modern GPU-accelerated graphics pipeline
  - However, modern graphics pipeline provides mechanisms to allow application to implement deferred shading efficiently
  - Natively implemented by PowerVR mobile GPUs
  - Classic hardware-supported implementations:
    - [Deering et al. 88]
    - UNC PixelFlow [Molnar et al. 92]

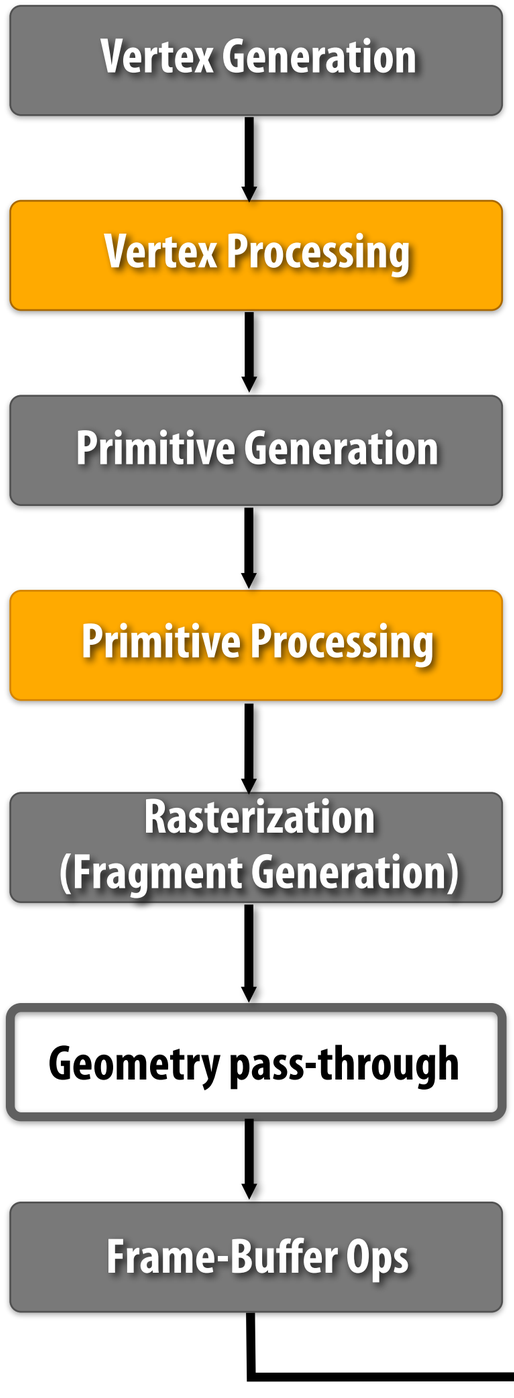
The graphics pipeline



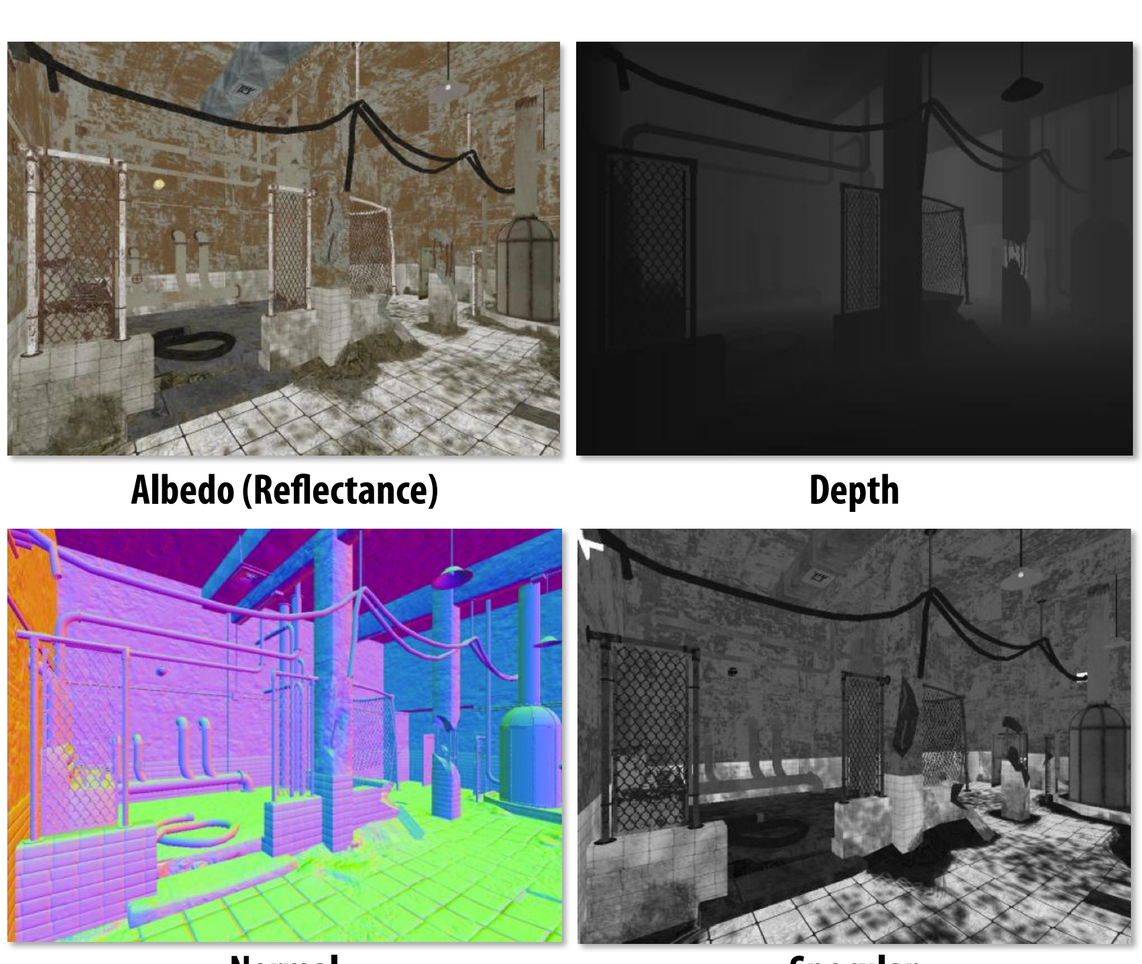
"Feed-forward" rendering

Typical use of fragment processing stage: evaluate application-defined function from surface inputs to surface color (reflectance)

Deferred shading pipeline



- Two pass approach:
- Do not use traditional pipeline to generate RGB image.
- Fragment shader outputs surface properties (shader inputs)
  - (e.g., position, normal, material diffuse color, specular color)
- Rendering output is a screen-size 2D buffer representing information about the surface geometry visible at each pixel (called a "g-buffer", for geometry buffer)



Albedo (Reflectance)

Depth

Normal

Specular

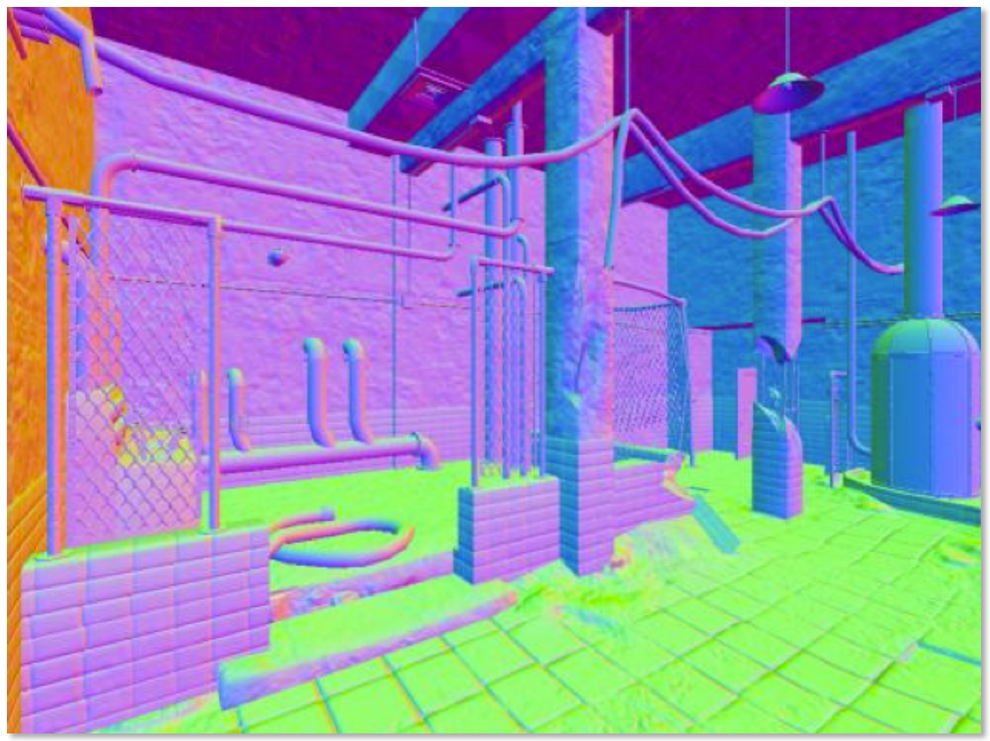
Deferred rendering is a refactoring

- *From*:
  - ```
    for object in scene
      for light in scene
        render(object, light)
    ```
- *To*:
  - ```
    for object in scene
      renderData(object)
    for light in scene
      renderLighting(light)
    ```

G-buffer = "geometry" buffer

Albedo (Reflectance)

Depth



Normal

Specular

Image Credit: J. Klint, "Deferred Rendering in Leadworks Engine"

Example G-buffer layout

Graphics pipeline configured to render to four RGBA output buffers (32-bits per pixel, per buffer)

R8	G8	B8	A8	
Depth 24bpp			Stencil	DS
Lighting Accumulation RGB			Intensity	RT0
Normal X (FP16)		Normal Y (FP16)		RT1
Motion Vectors XY		Spec-Power	Spec-Intensity	RT2
Diffuse Albedo RGB			Sun-Occlusion	RT3

Source: W. Engel, "Light-Prepass Renderer Mark III" SIGGRAPH 2009 Talks

- Implementation on modern GPUs:
  - Application binds "multiple render targets" (RT0, RT1, RT2, RT3 in figure) to pipeline
  - Rendering geometry outputs to depth buffer + multiple color buffers

- More intuitive to consider G-buffer as one big buffer with "fat" pixels
- In the example above: 32 x 5 =160 bits = 20 bytes per pixel

- 96-160 bits per pixel is common in games

Compressed G-buffer layout

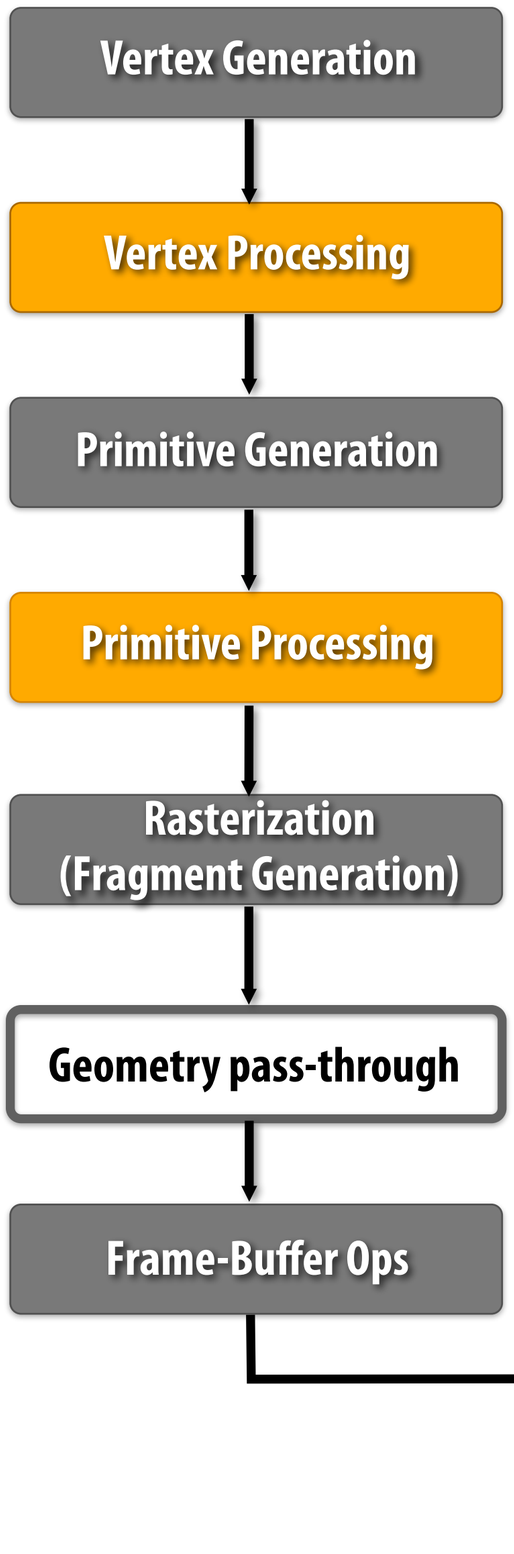
G-buffer layout in Bungie's Destiny (2014)

8	8	8	8	
Albedo Color RGB			Ambient Occlusion	RT0
Normal XYZ * (Biased Specular Smoothness)			Material ID	RT1
Depth			Stencil	DS

- **Material information compressed using indirection**
  - **Store material ID in G-buffer**
  - **Material parameters other than albedo (specular shape/ roughness/color) stored in table indexed by material ID**

Source: N Tatarchuk: SIGGRAPH 2014 Courses

“Two pass” deferred shading



Two pass approach:

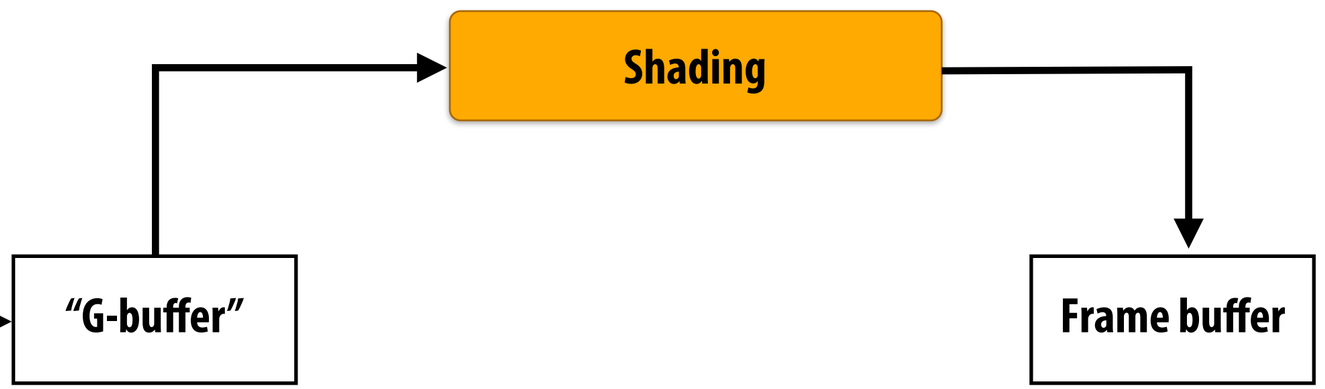
Do not use traditional pipeline to generate RGB image.

Fragment shader outputs surface properties (shader inputs)
(e.g., position, normal, material diffuse color, specular color)

Rendering output is a screen-size 2D buffer representing information about the surface geometry visible at each pixel (called a “g-buffer”, for geometry buffer)

After all geometry has been rendered, execute shader for each sample in the G-buffer: shader reads geometry information for sample, computes RGB output

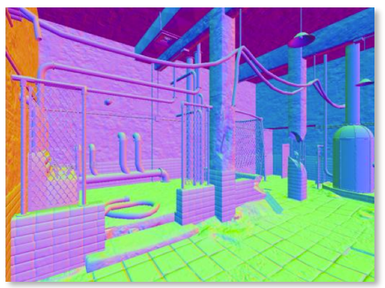
(shading is deferred until all geometry processing -- including all occlusion computations -- is complete)



Two-pass deferred shading algorithm

- **Pass 1: G-buffer generation pass**
  - Render scene geometry using traditional pipeline
  - Write visible geometry information to G-buffer






- **Pass 2: shading/lighting pass**

  For each G-buffer sample, compute shading
  - Read G-buffer data for current sample
  - Accumulate contribution of all lights
  - Output final surface color for sample

Final Image

Image Credit: J. Klint, "Deferred Rendering in Leadworks Engine"

Motivation: why deferred shading?

- Shading is expensive: shade only visible fragments
  - Deferred shading amounts to perfect early occlusion culling
  - Deferred shading is triangle order invariant (will only shade visible fragments, regardless of application's triangle submission order)
  - Also has nice property that the number of shaded fragments is independent of scene complexity (predictable shading performance)

- Recall: forward rendering shades small triangles inefficiently
  - Recall shading granularity is quad fragments: multiple fragments generated for pixels along triangle edges