7-GPU Architecture

Dr. Hamish Carr & Dr. Rafael Kuffner dos Anjos

Agenda

HOW GPU’S WORK

MODERN PIPELINE

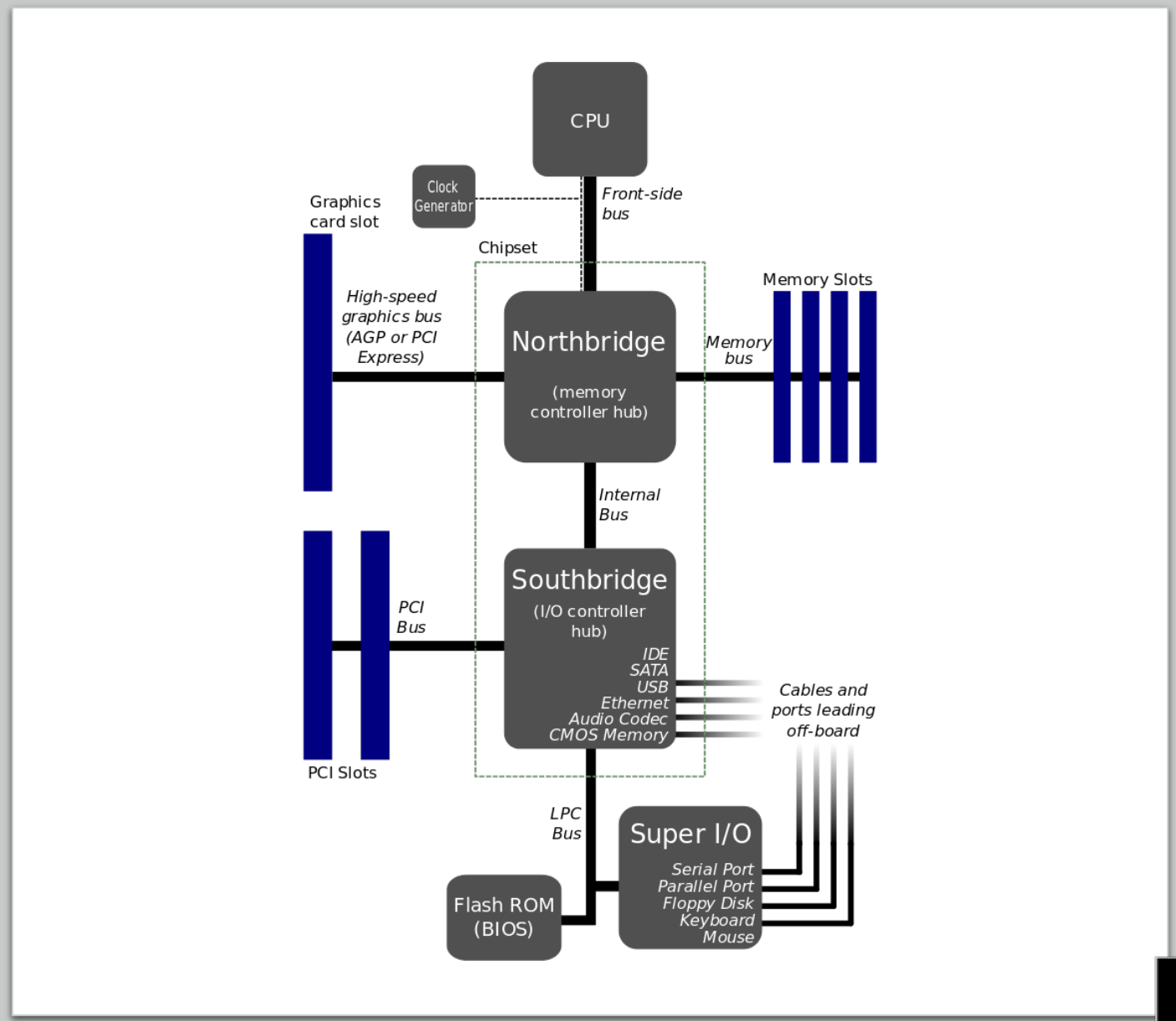
SHADERS

GPUs

- Can be *discrete* or *integrated*
- Many levels, but all have custom silicon
  - Hardware-software co-design
  - For 35 years

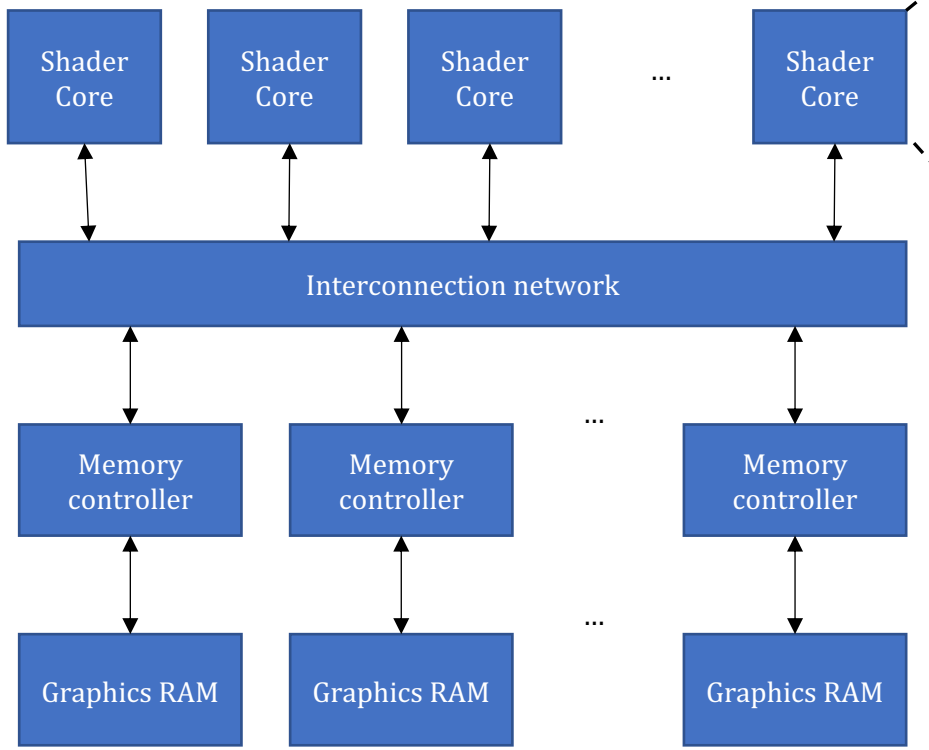
GPU-Centric PC Architecture

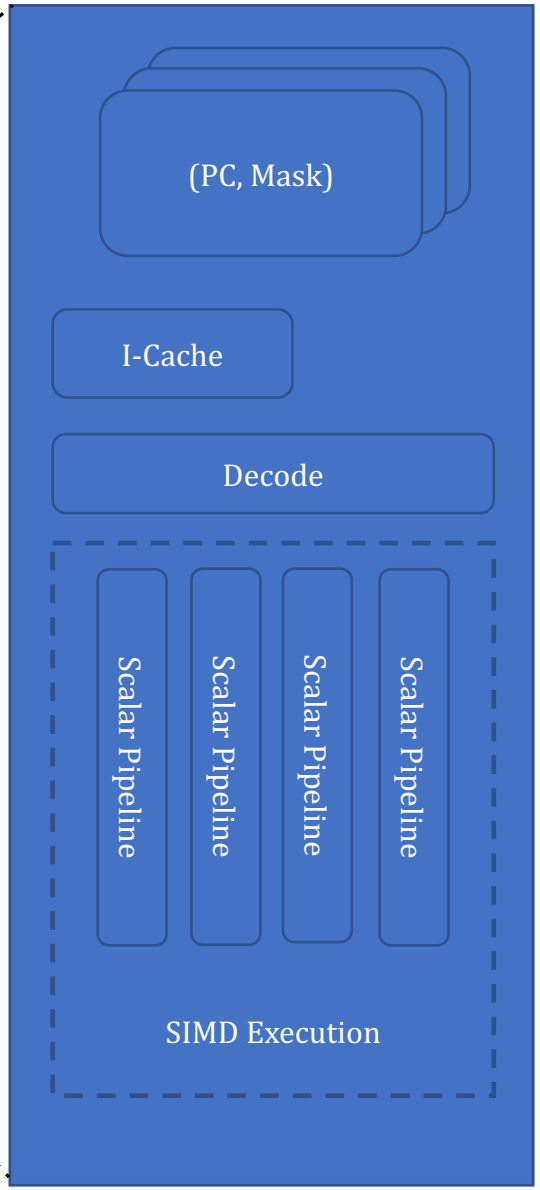
- Core logic on motherboard
- Everything fed to CPU
- GPU connected over bus
  - data transfer limited
  - Batch transfer preferred
- GPU often more powerful
  - CPU is *just* the I/O processor



High-Level View of GPU

- Massively parallel by design
- Many processors interacting





GPU Characteristics

- Many shader cores (1000s of processors)
- Stream processing
- SIMD using warps (NV) / wavefronts (AMD)
- Small amounts of per-core memory
- High latency to main VRAM
- Stalling common when accessing data
- But many 1000s of data to be processed

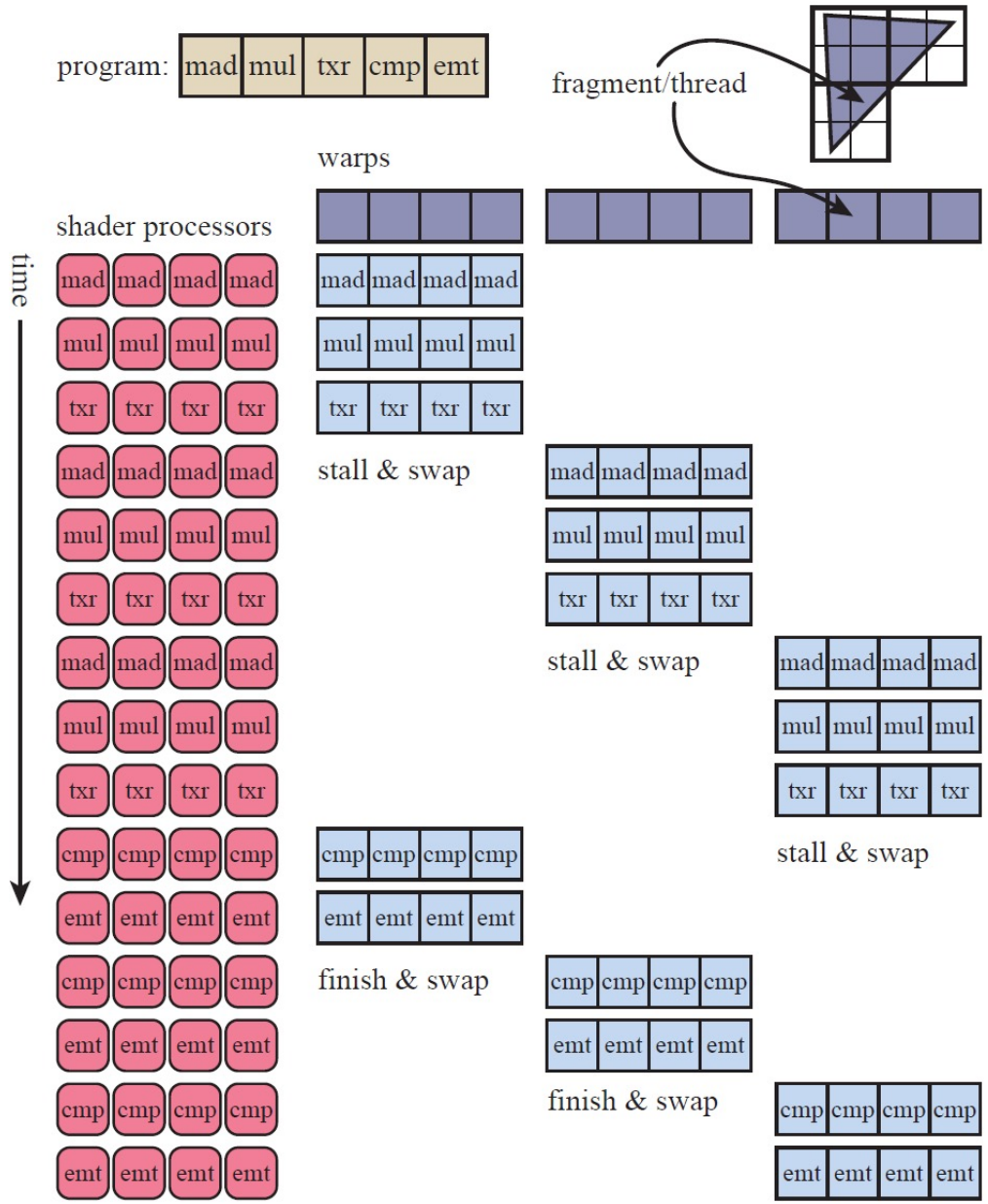
Context Switching

- A core computes something in register
- Then it accesses a texture and stalls
- So suspend the thread and start another
- And use *hardware* to perform the switch
- Return to first thread after data load done
- Minimises downtime, maximises throughput
- Because we have so many operations to do

SIMD Parallelism

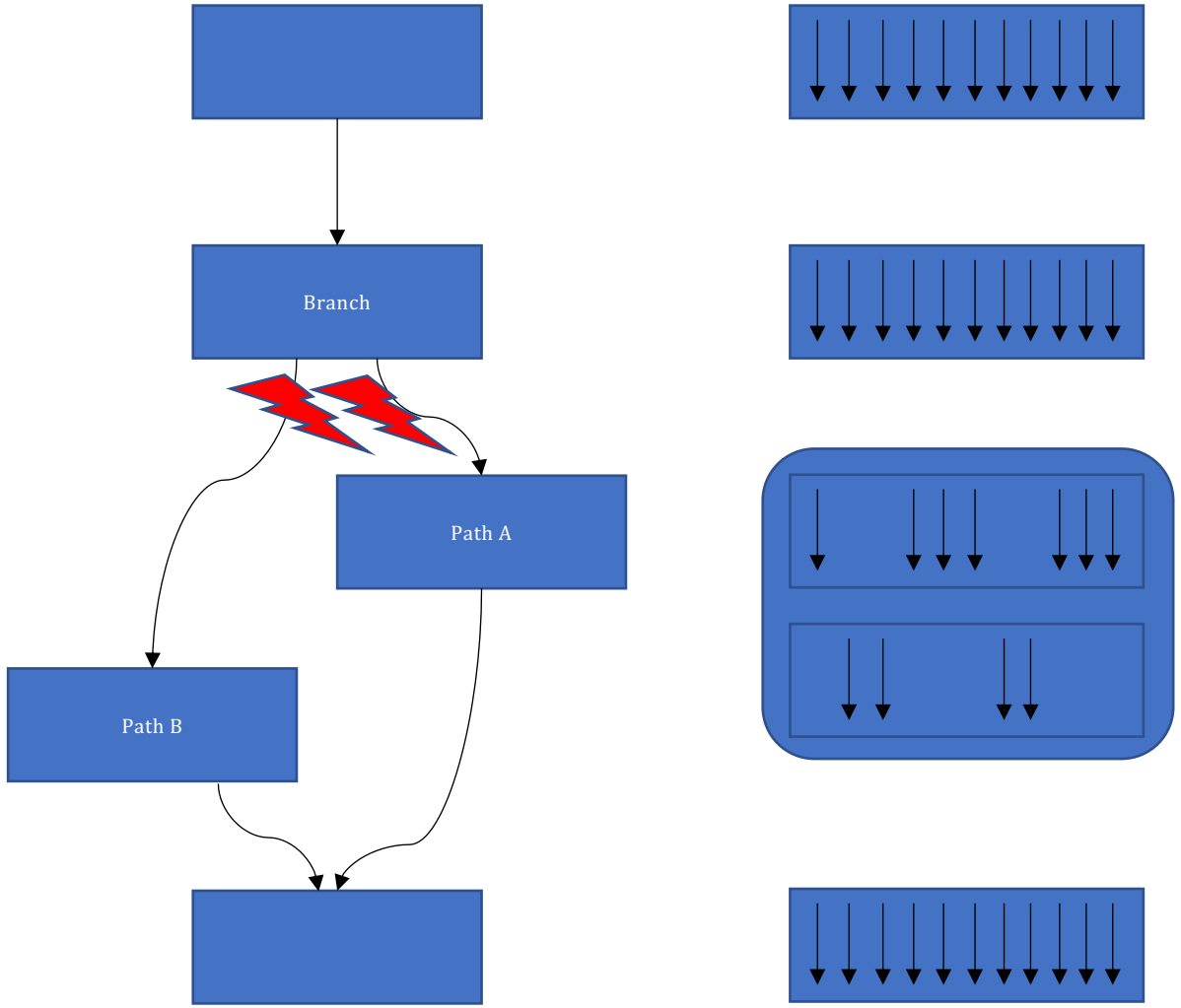
- Instructions *shared* between groups of cores
  - Called *warps* (NVIDIA) / *wavefronts* (AMD)
  - Usually 8-64 cores mapped to a SIMD lane
- Works best with non-branching code
- And when memory fetches are few
- And when register usage is low
- All this affects the *occupancy rate*

Simplified Example

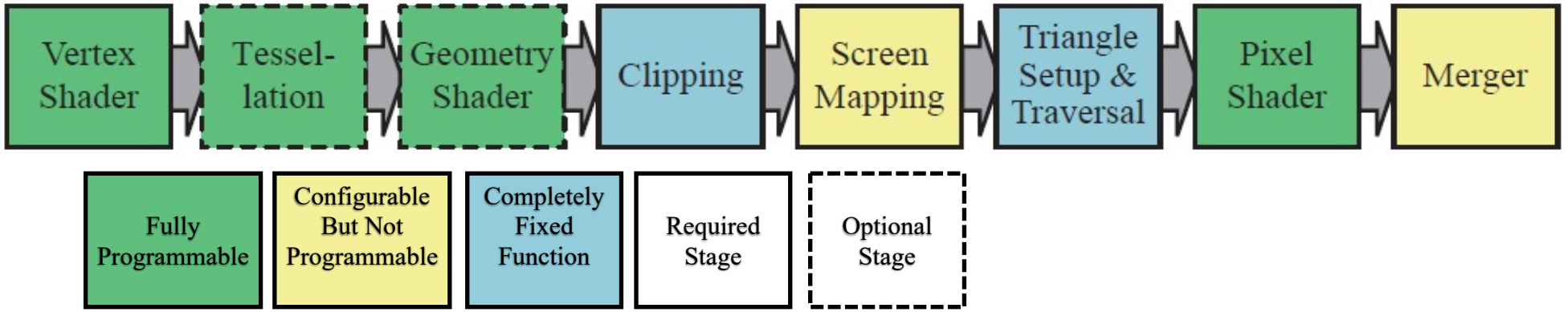


Divergence

- If statements stall pipelines
- Cores take different branches
- Warp threads *diverge*
- Branch to different codepaths
  - Often controlled by semaphores
  - Tends to cause bottlenecks



The Modern Pipeline



- Based on the projective rendering pipeline
- With extra stages (of course!)
- But essentially what we've been implementing

Draw call – invocation by the programmer

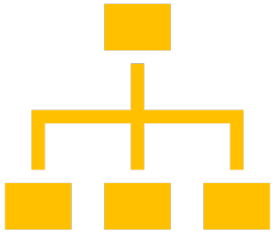
Uniform input – value constant over draw call

Varying input – value varying per element

Causes pipeline to execute & run shaders
May involve 1000s of vertices

Terminology

Uniform vs. Varying



Uniform values are the same for all elements

E.g. lighting (i.e. a lot of OpenGL state)

Lots of *constant registers* available on GPU

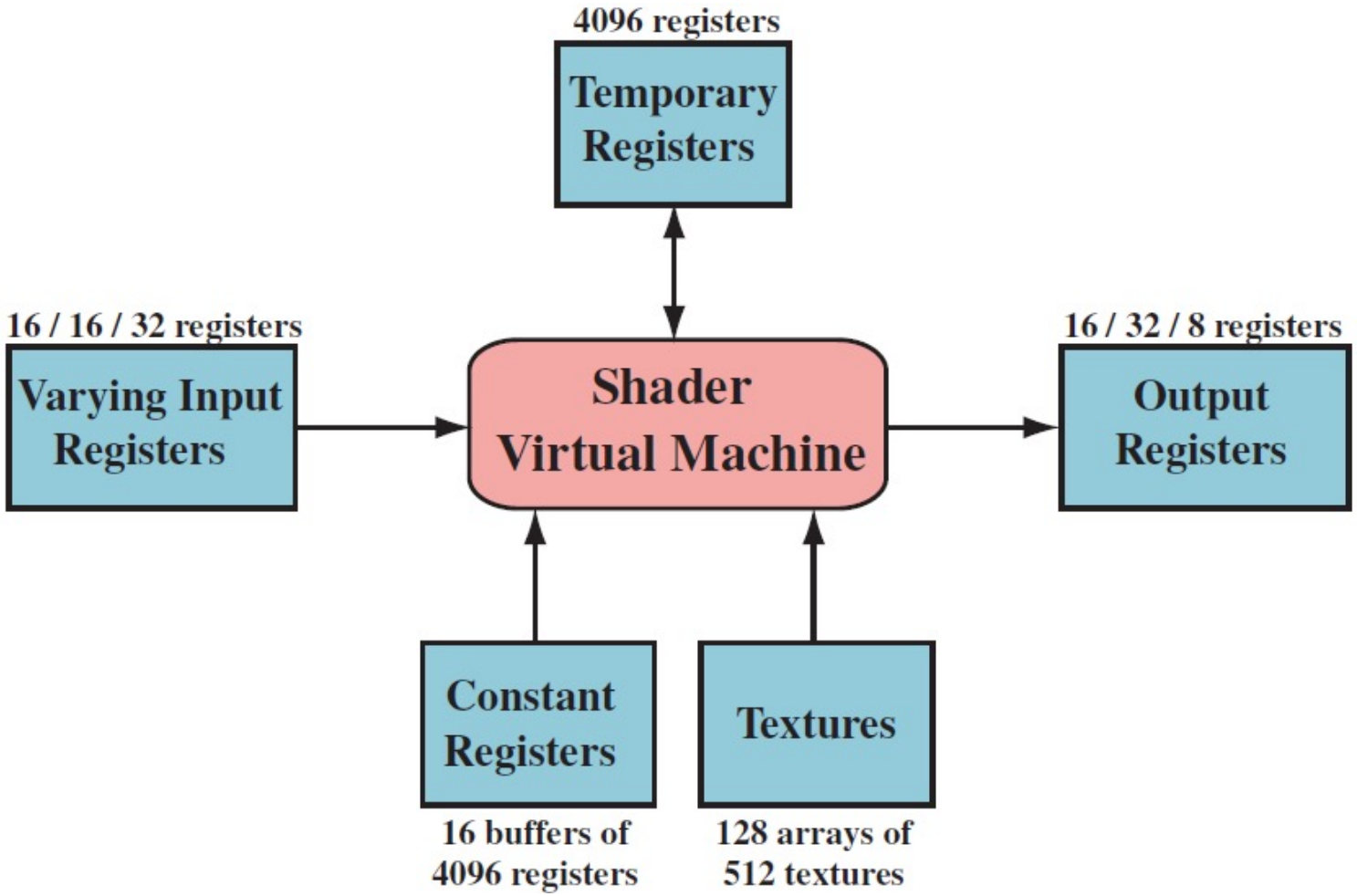
Accessible from all cores

Varying values are different for each element

E.g. uv coordinates

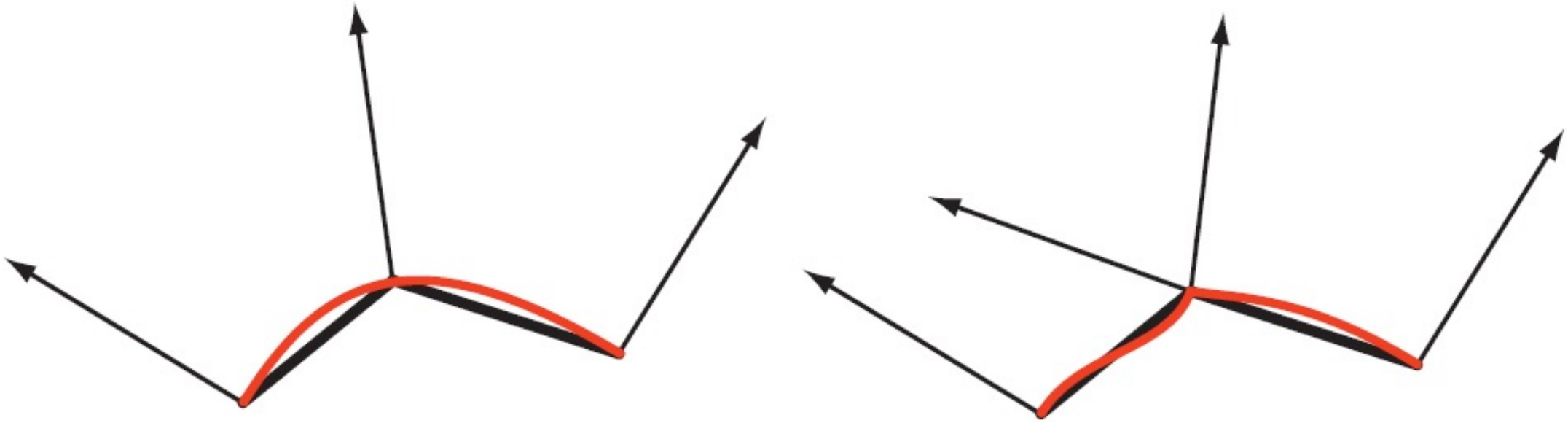
Stored in local registers per core

Unified VM Architecture



Input Assembler

- Fixed stage *before* vertex shader
- Weaves together data streams
  - E.g. mixing vertices & normal
  - Striping data from vertex arrays
- Also allows instancing
  - Reuse of the same object multiple times



Vertex Shader

- First stage under programmer control
- Converts to clip space (required)
- May also tweak vertex position, properties
- Occurs *before* lighting computed
- E.g. adjusting normal to generate a crease

Uses of Vertex Shaders

Deforming an existing mesh

Animation: skinning & morphing

Procedural deformation: flags, cloth, water

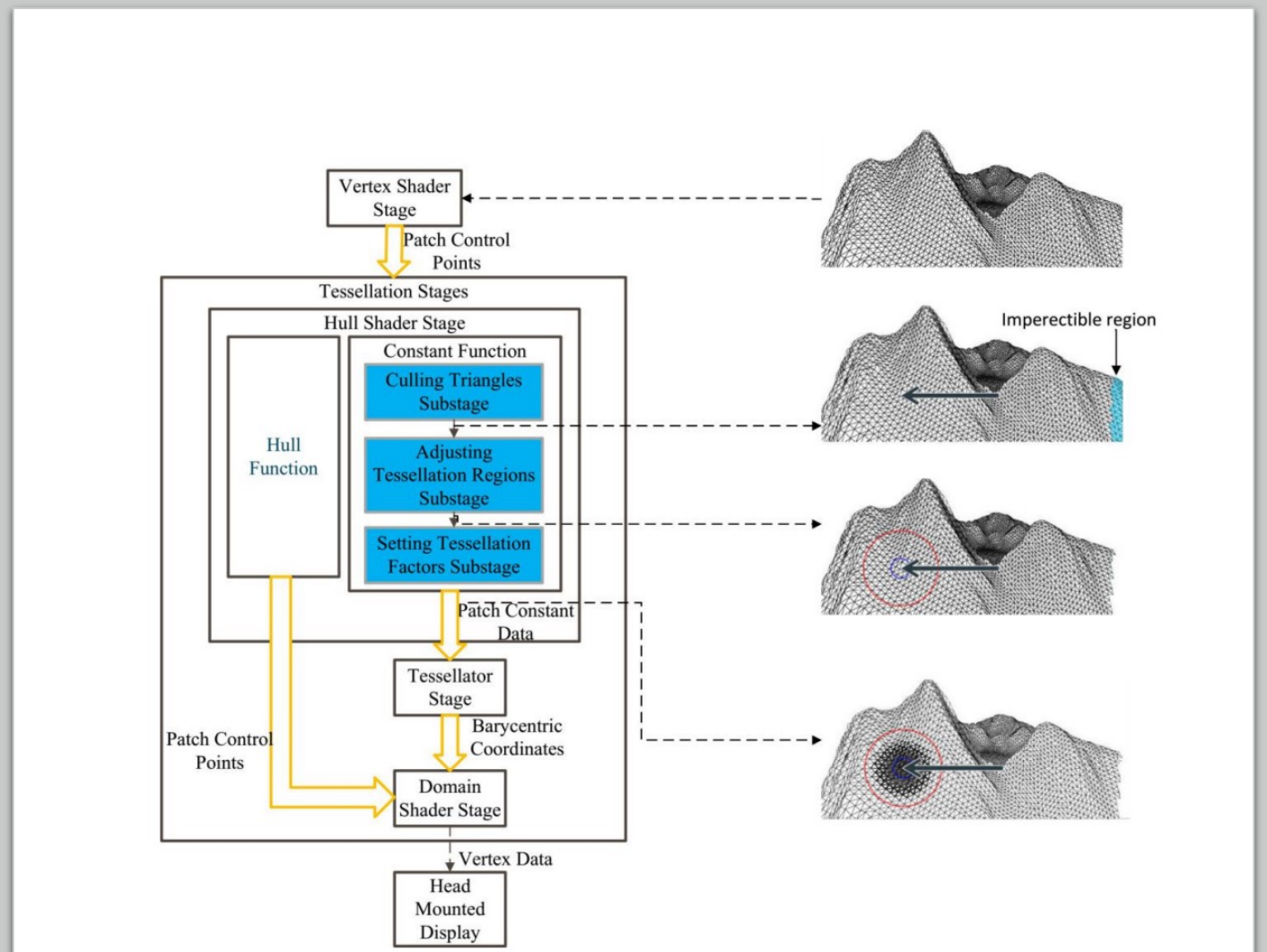
Particle creation: from degenerate meshes

Lens distortion, heat haze, &c.

Applying terrain height fields

Tessellation Stage

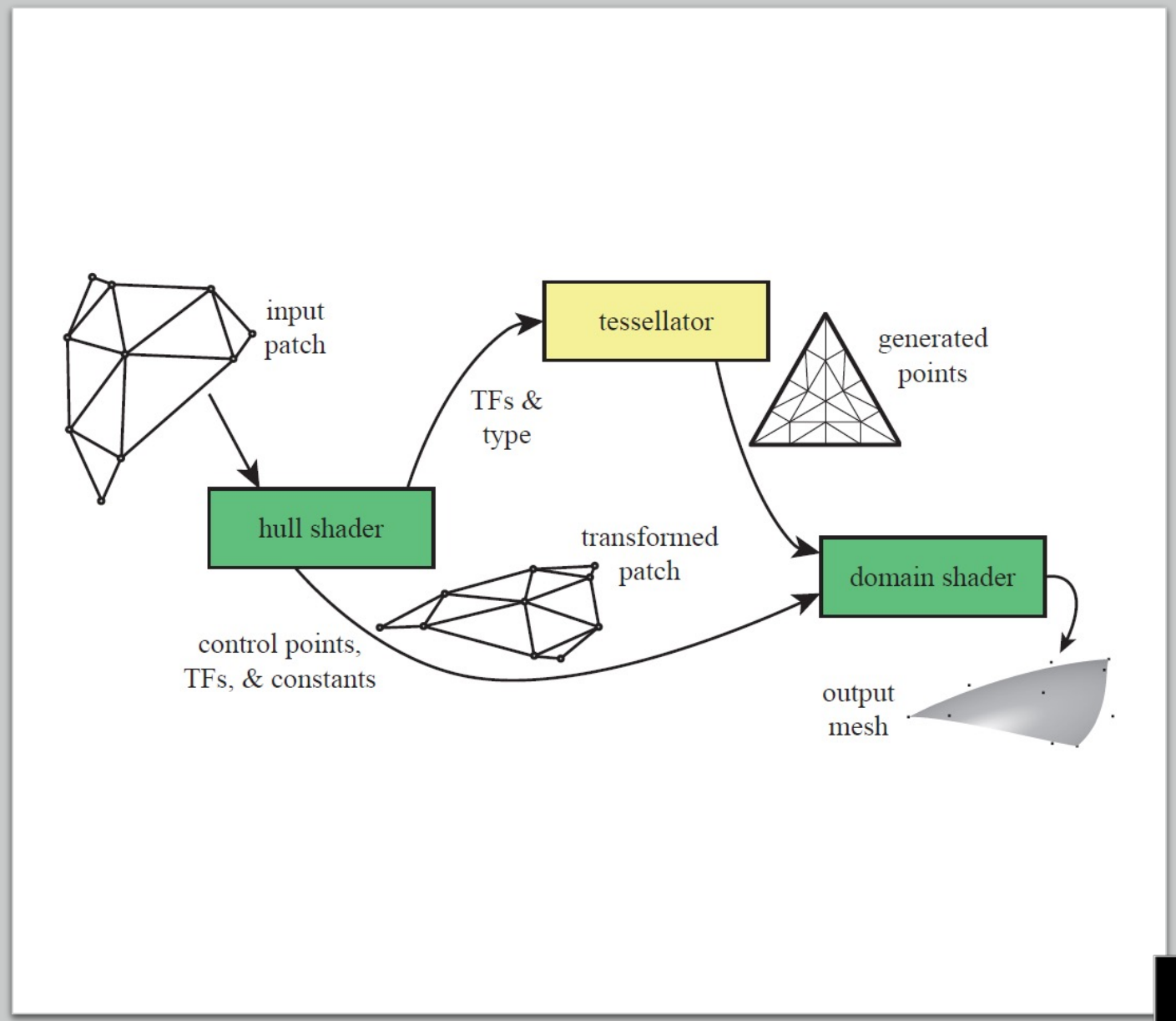
- Allows converting mesh into finer mesh
- Useful for higher-order surfaces
- Saves memory / bandwidth
- Consists of three internal stages
  - *Hull shader (tessellation control shader)*
  - *Tessellator*
  - *Domain shader (tessellation evaluation shader)*

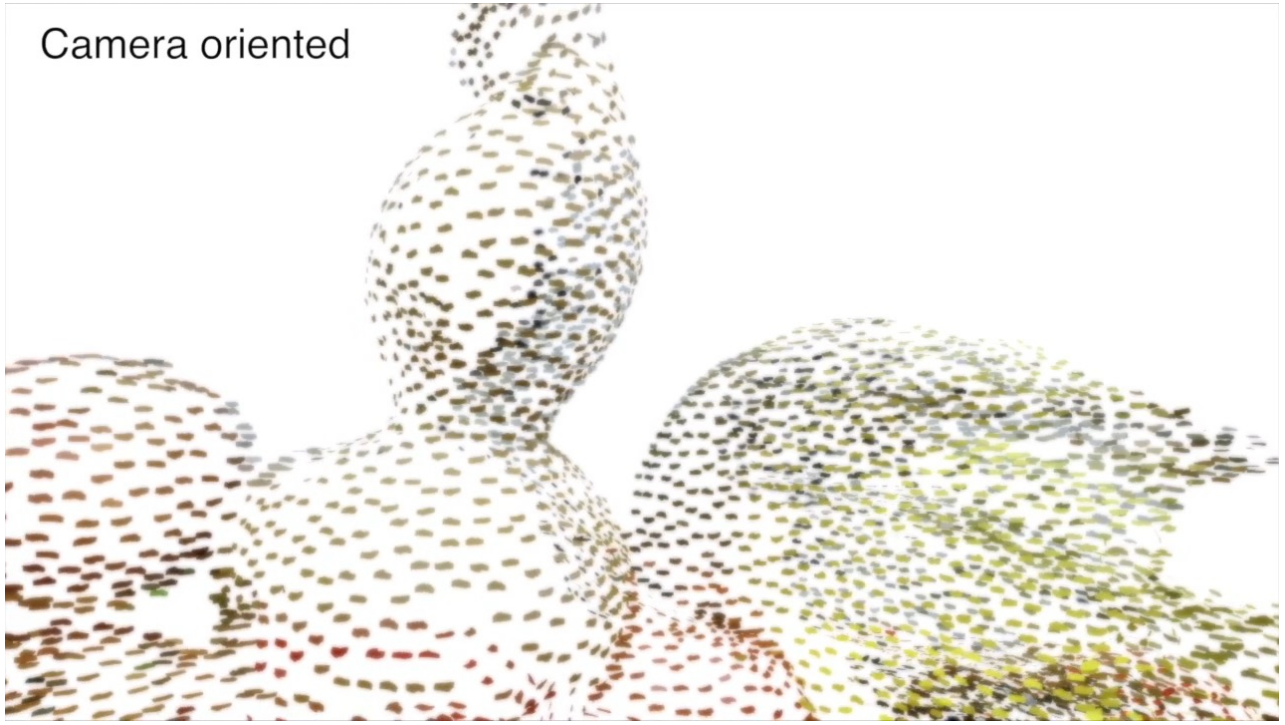


Zheng et al. 2020

Tessellation Substages

- Hull shader modifies input patch
- Tessellator converts to smaller patches
- Domain shader generates vertex attributes



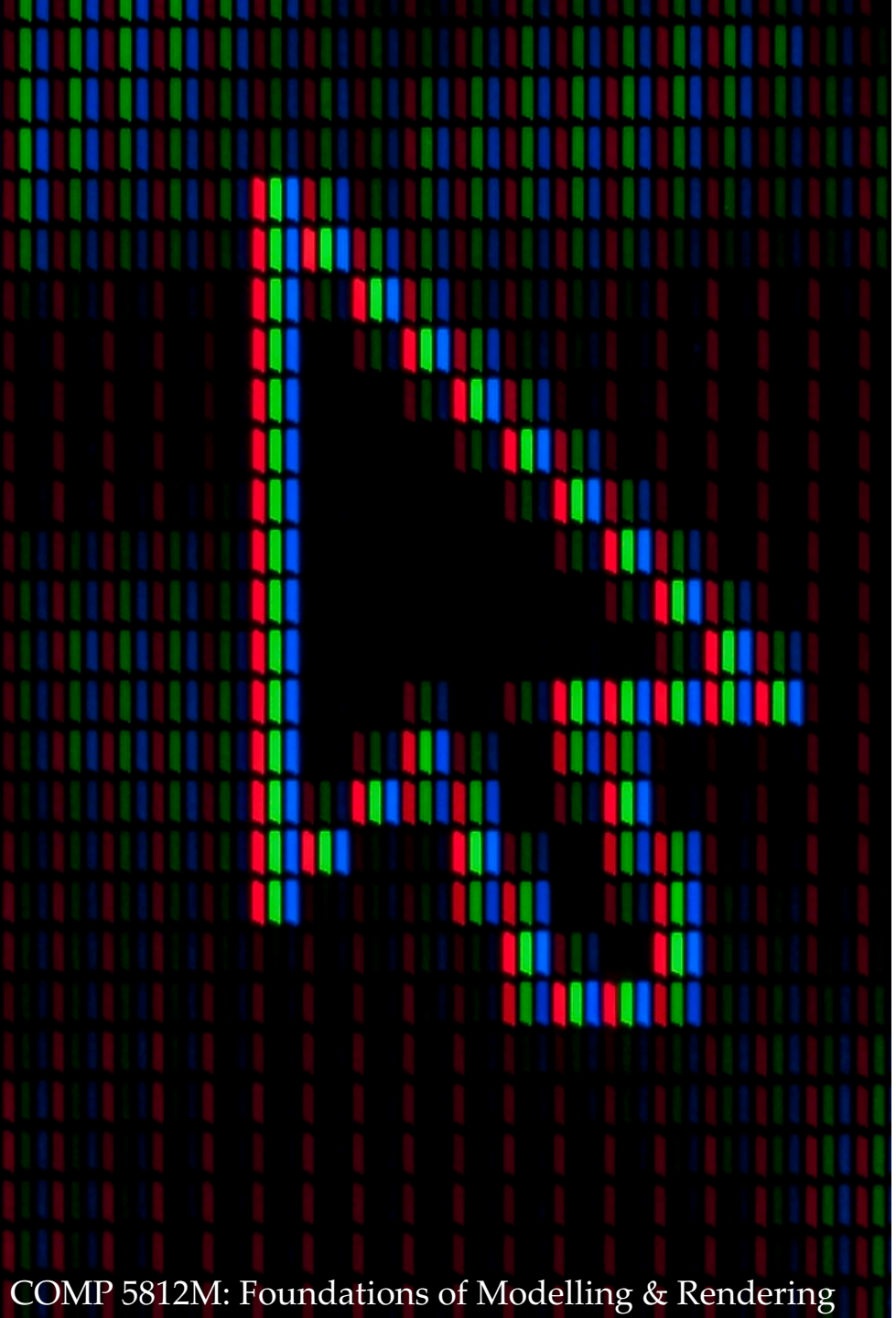


Geometry Shader

- Converts primitives into other primitives
- E.g. turns mesh into wireframe, point into quad
- We will use this for a fur shader next term
- Can be used to generate multiple streams
- Not the most heavily used
  - *Because* it is so general-purpose
  - Means it is harder to optimise

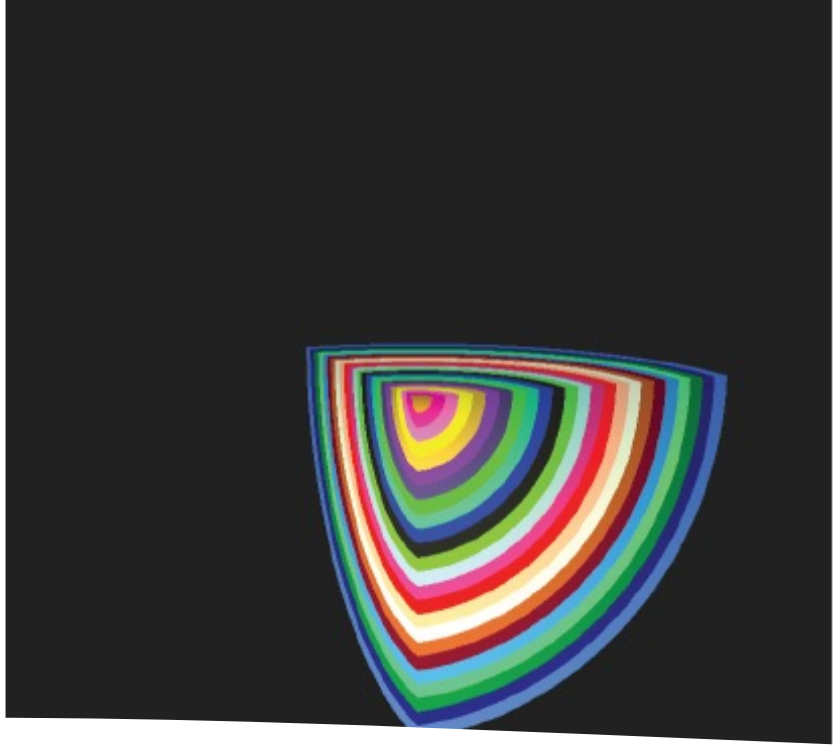
Clipping, Mapping, Raster

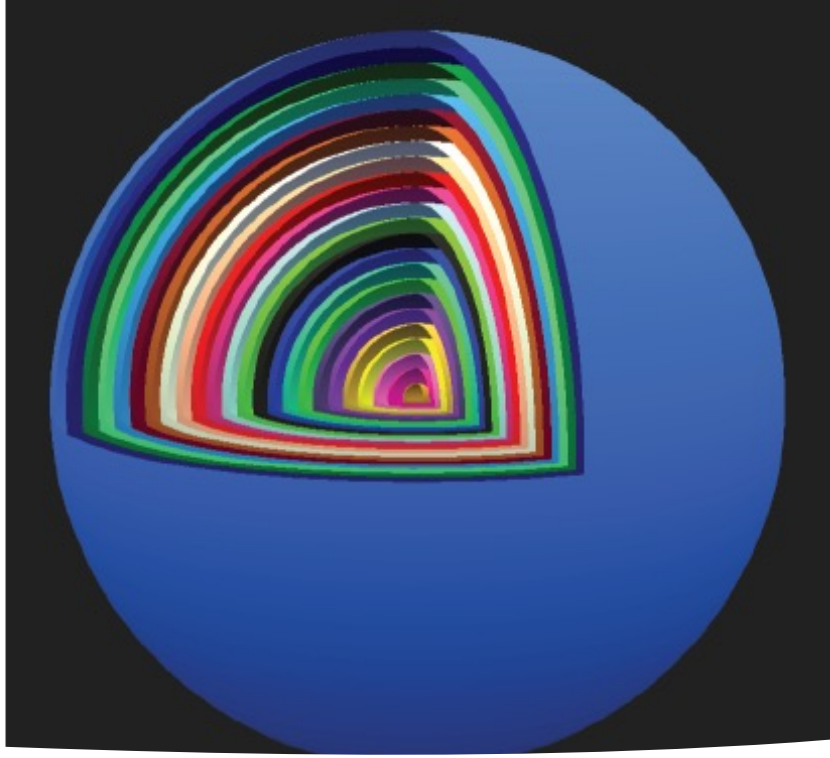
- Clipping stage trims off-screen portions
  - Can convert triangle into quad (2 triangles)
- Mapping stage is the viewport transform
- Rasterisation we know about
  - Generates fragments
  - Computes barycentric coordinates
  - Interpolates attributes from vertices



Pixel Shader

- Also known as the fragment shader (OpenGL)
- Input is fragments with attributes
- Typically computes the fragment's colour
  - Also opacity, can modify z-depth
- Has the ability to *discard* the fragment
- Does *not* perform the depth test
- Can render to multiple targets





Clipping in Pixel Shader

- Input: nested spheres in different colours
  - Already rasterised into fragments
  - Discard any fragments in a known quadrant
  - Simple geometric test on x,y,z

Multiple Render Targets

- We have used a *colour* and *depth* buffer
- Why limit ourselves to two?
- Pixel shader can write to *many* buffers
- And can send output to different places
- Eg: colour in one, object IDs in another
- Leads to *deferred* shading
  - Compute visibility, save attributes for later

Pixel Shader Limitation

- Pixel shader can only write to *its own* x,y
- Some exceptions in practice
- However, different cores can have same x,y
- Buffer locking done per-pixel ***in hardware!***
- Atomic operation means pixel shader stalls
- But this is tolerable in practice, due to high number of fragments

Stream Output

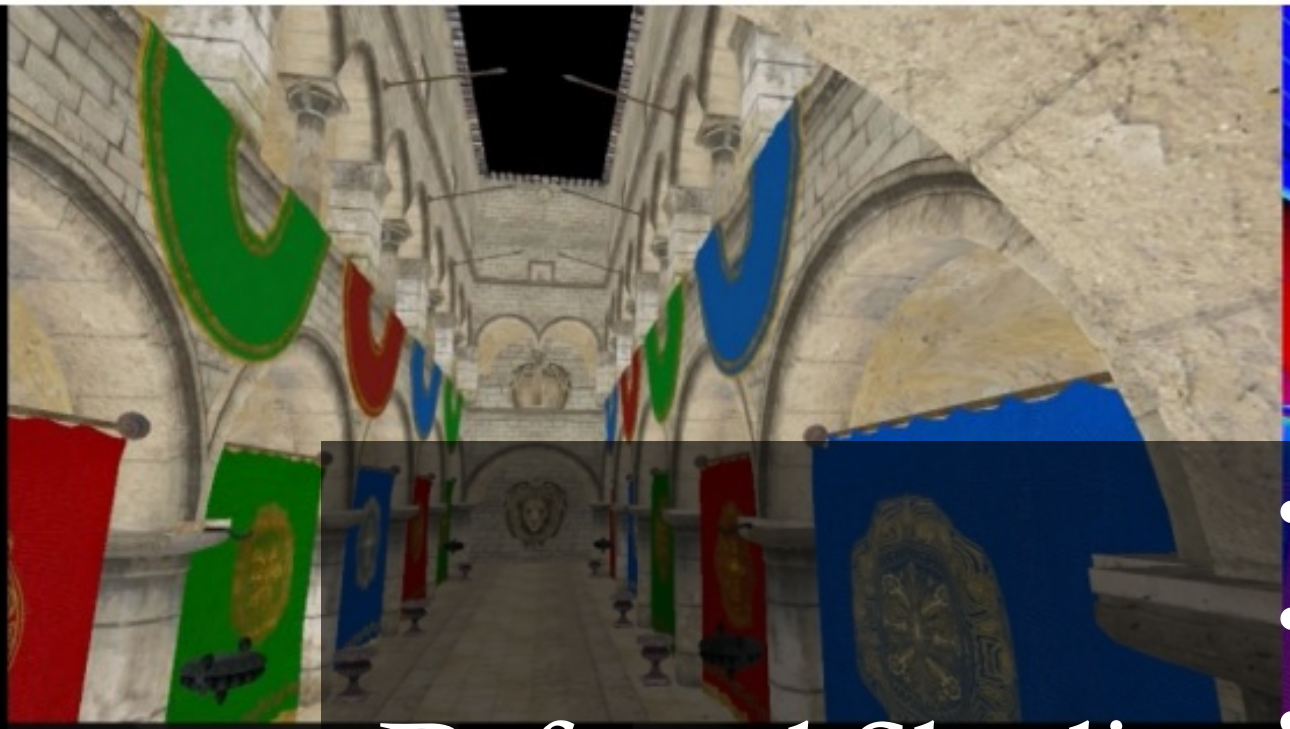
- Relax the constraint on pixel shader location
- Allow it to write *anywhere* in a buffer
- Which then just becomes an arbitrary array
- Use the pixel shaders to compute *anything*
  - Geometry, vertices, lighting, whatever
- Then use the array as input in a second pass
  - *Multi-pass* rendering & compute shaders

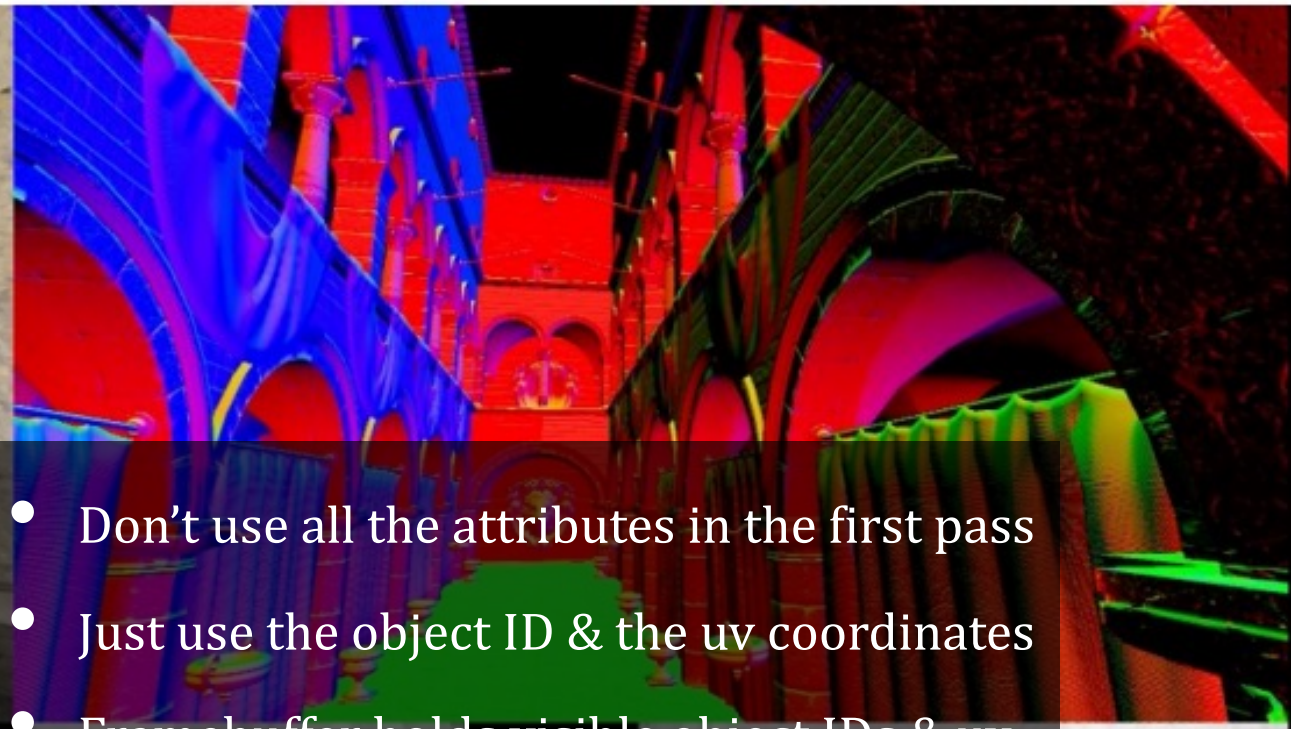
Merging Stage

- Output merger (DX) / Per-sample ops (OpenGL)
- Combines individual fragments
  - Essentially doing image processing
- Includes depth-test, stencil-buffer, blending
- In theory, can save and sort fragments (A-Buffer)
- But in practice not common

Lighting

- The vertex shader doesn't do lighting
- Because we now assume we do Phong shading
- I.e. the pixel shader does it per-fragment
- Before the z-test
- We could test z in pixel shader (early-z: DX 11)
- Or we can leave it to merging (DX 12)
  - Why would we do it that way?





Deferred Shading

- Don’t use all the attributes in the first pass
- Just use the object ID & the uv coordinates
- Framebuffer holds visible object IDs & uv
- Now re-render a single quad
- Only one fragment per pixel
- Lighting computation only done once

Oosten 2015

Net Result

- The pipeline stages are *malleable*
- We move computations around all the time
- Do *NOT* assume that this will remain fixed
- We may end up doing pure compute
  - And leaving *all* of it up to the coder
  - Until someone simplifies it again

Compute Shader

- Throw all of the existing stages out
- Just use the GPU as a compute device
- Now common for neural nets & other task
- Eventually, we may write entire pipelines
  - But for now, the standard stages are useful

Images by

- S3: Nana Dua
- S7: Mae Um
- S22: Umberto
- S27: Lance Grandahl