GPU ARCHITECTURE & GEOMETRY SHADER

Luca Del Bolgia

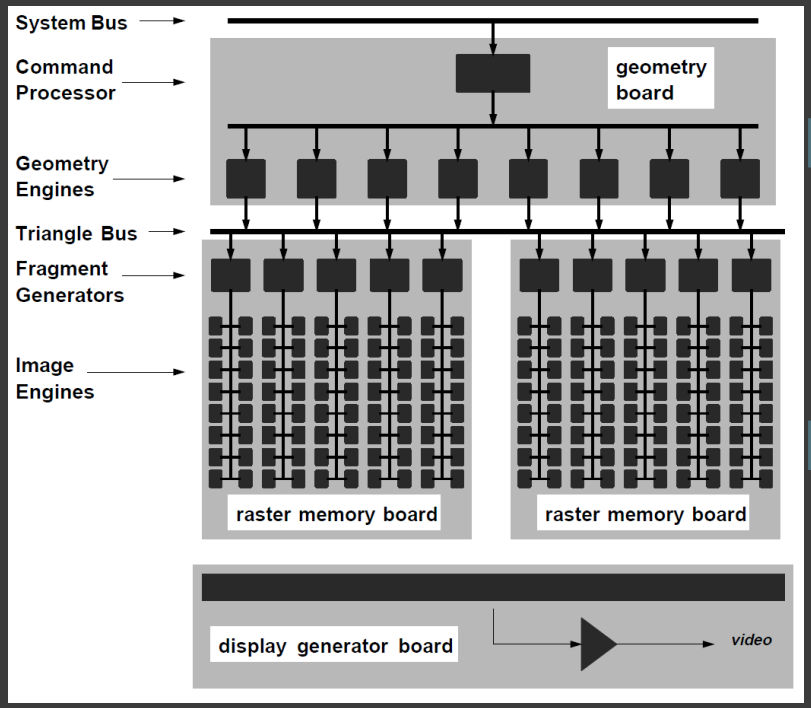
Summary

- What we know - CG course
- Changes to the pipeline
- Geometry Shader
- DX11

WHAT WE KNOW SUMMARY

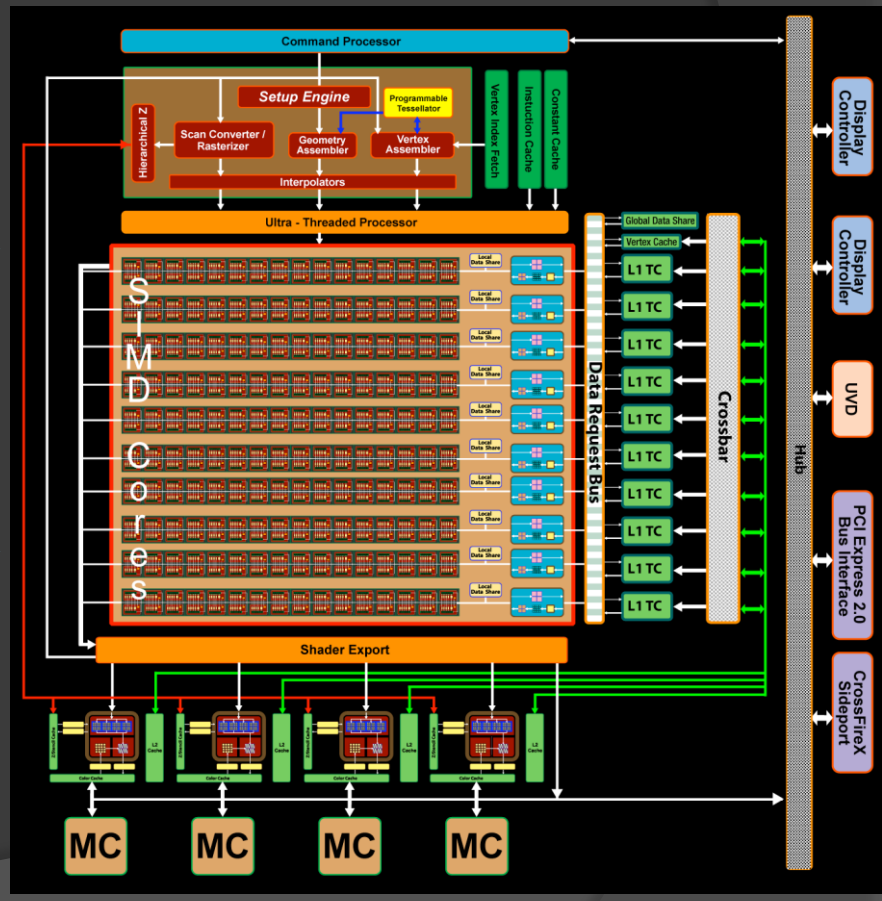
SUPERCOMPUTERS

SGI Reality Engine

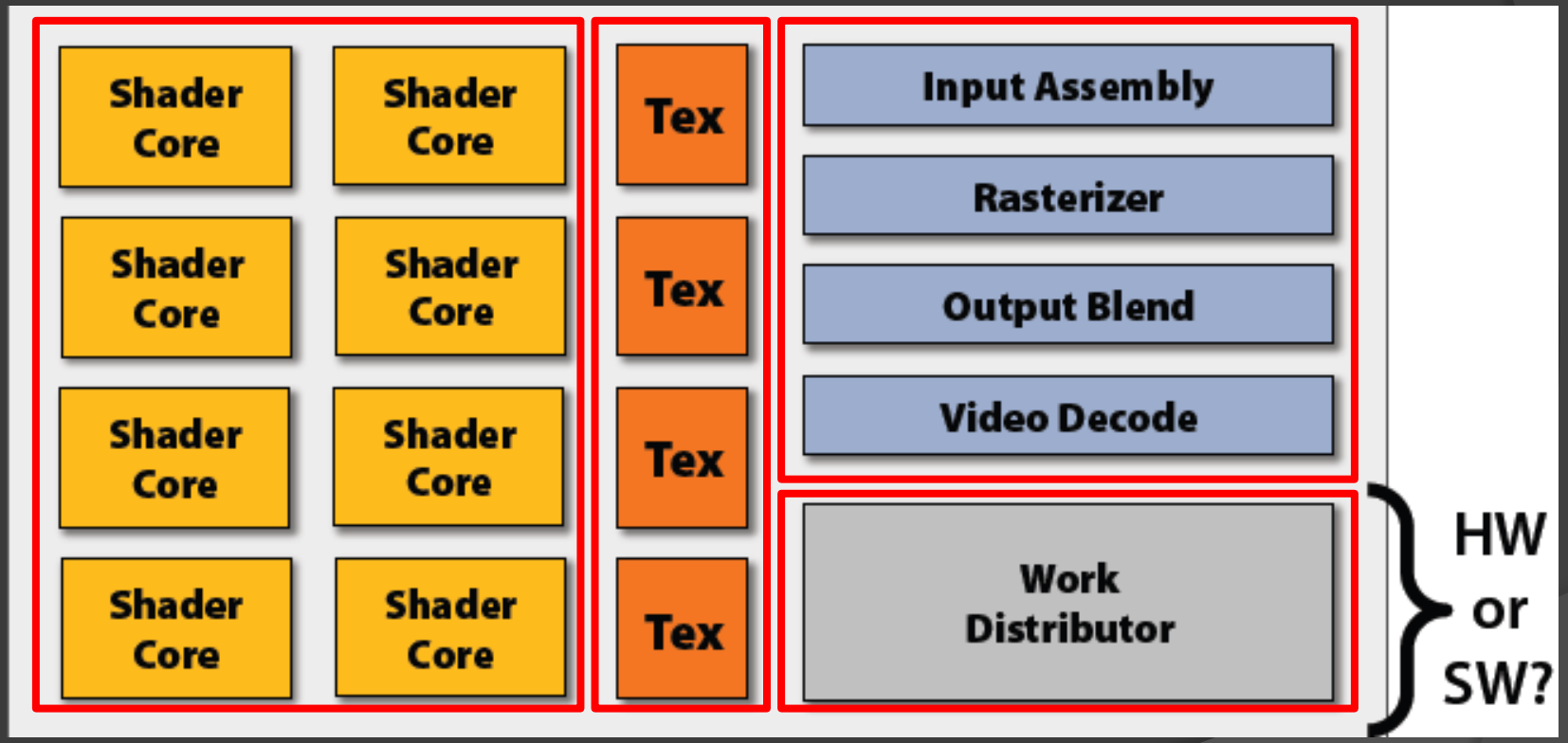


GPU

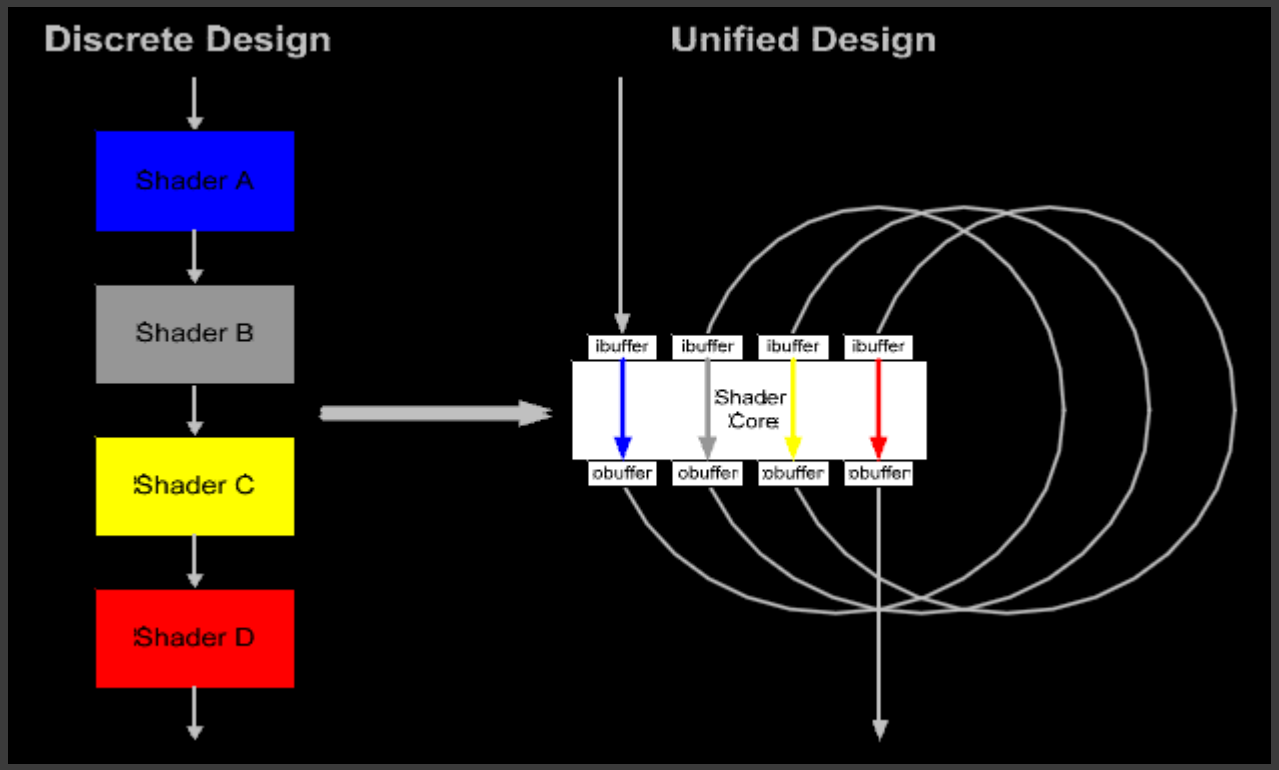
ATI RV770



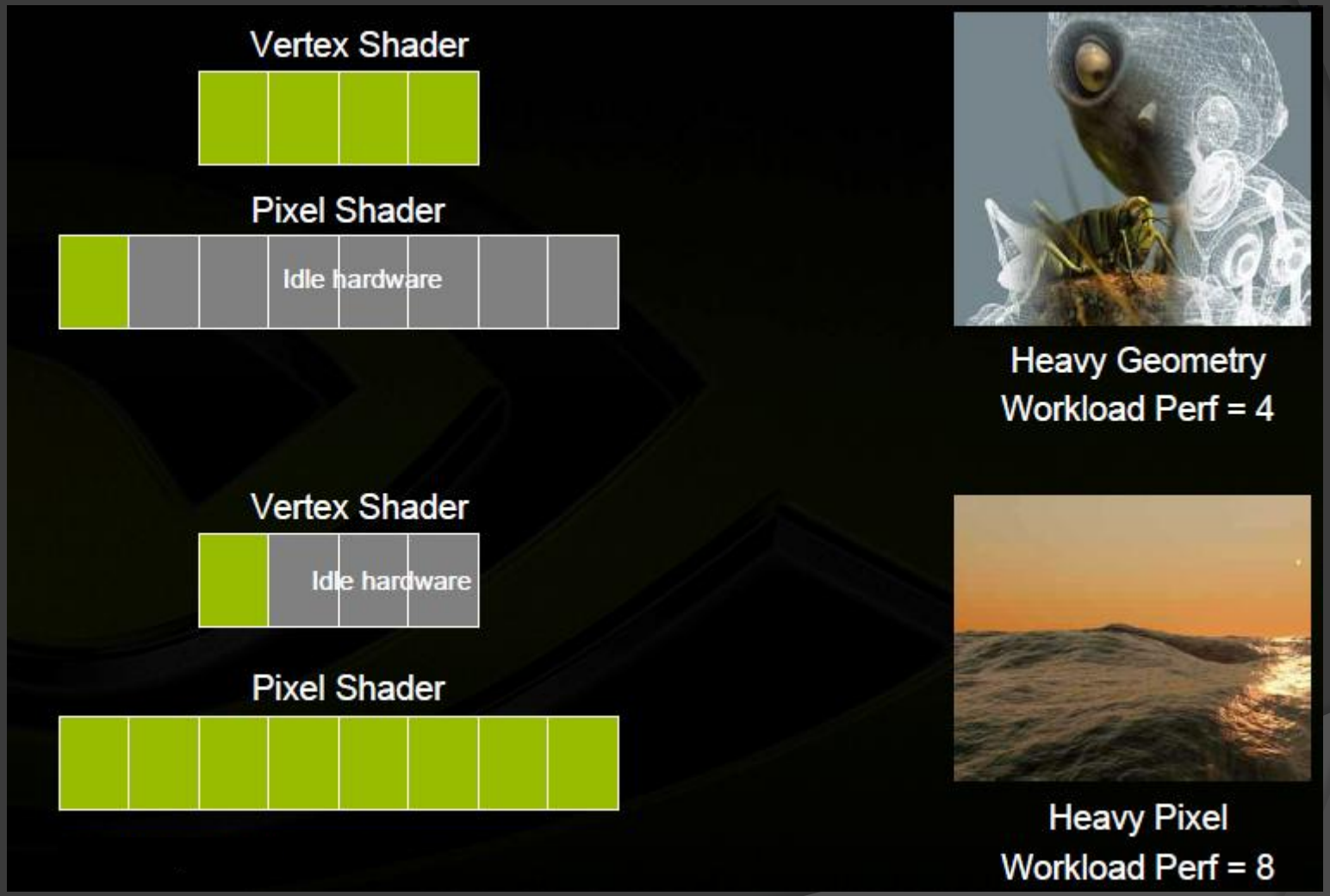
Inside the GPU



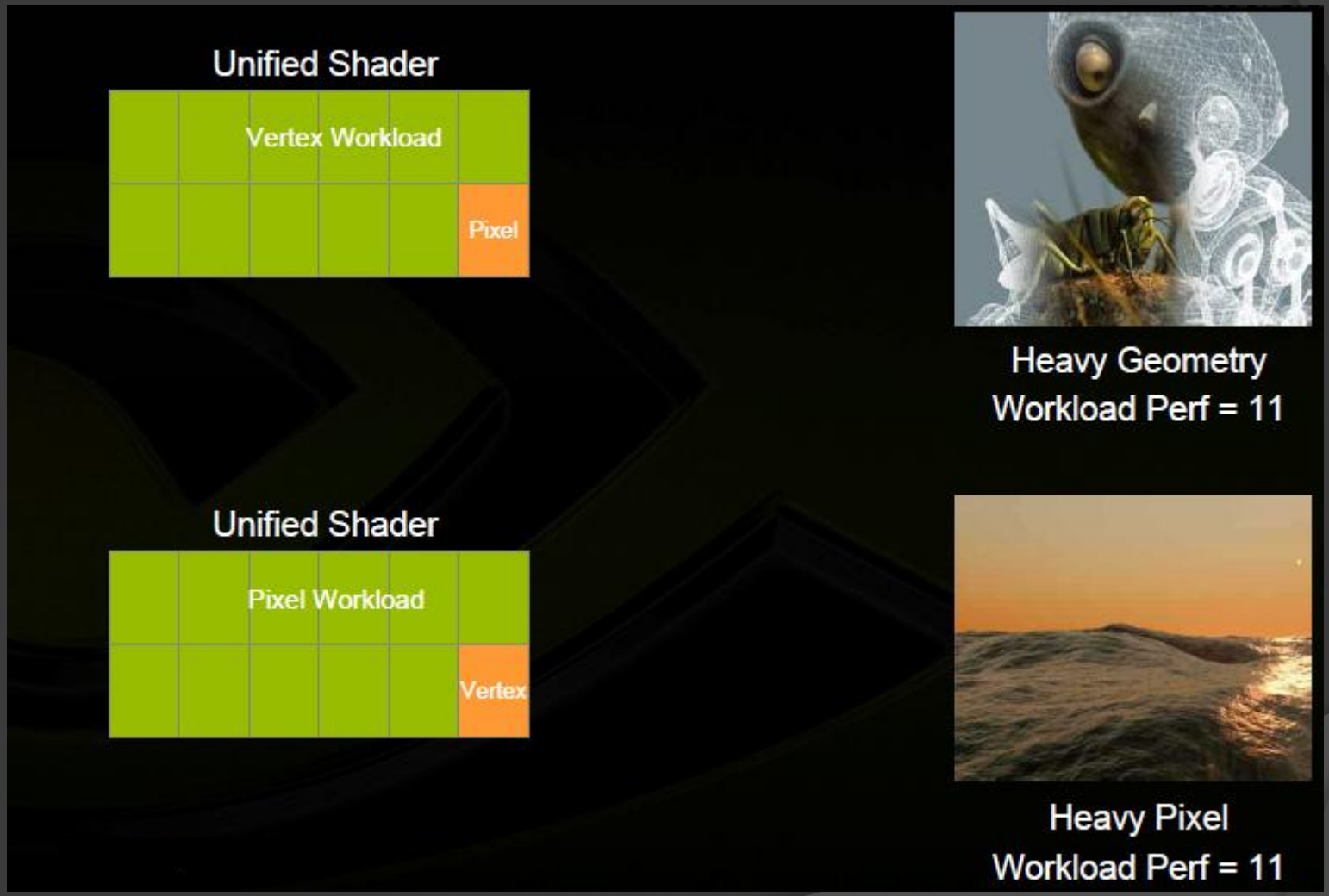
Unified Shader



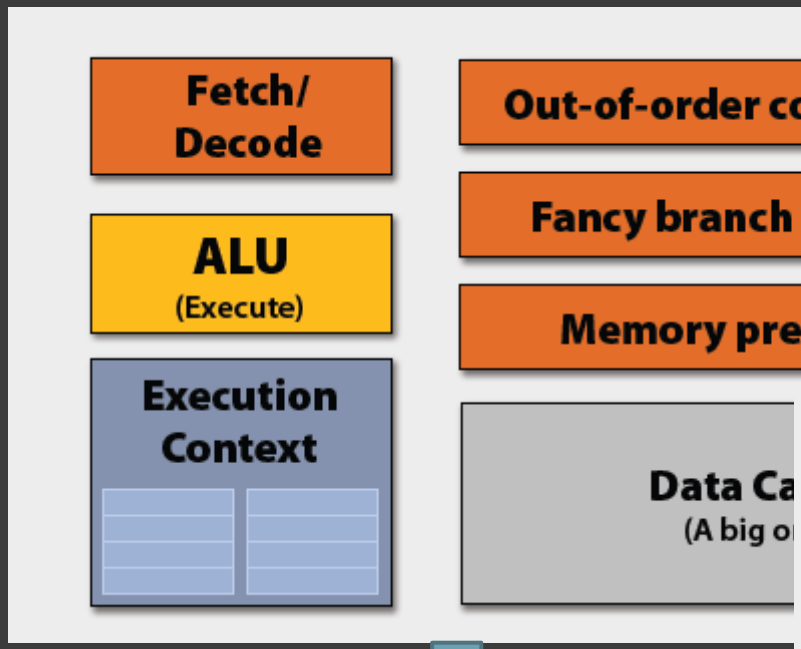
Unified Shader



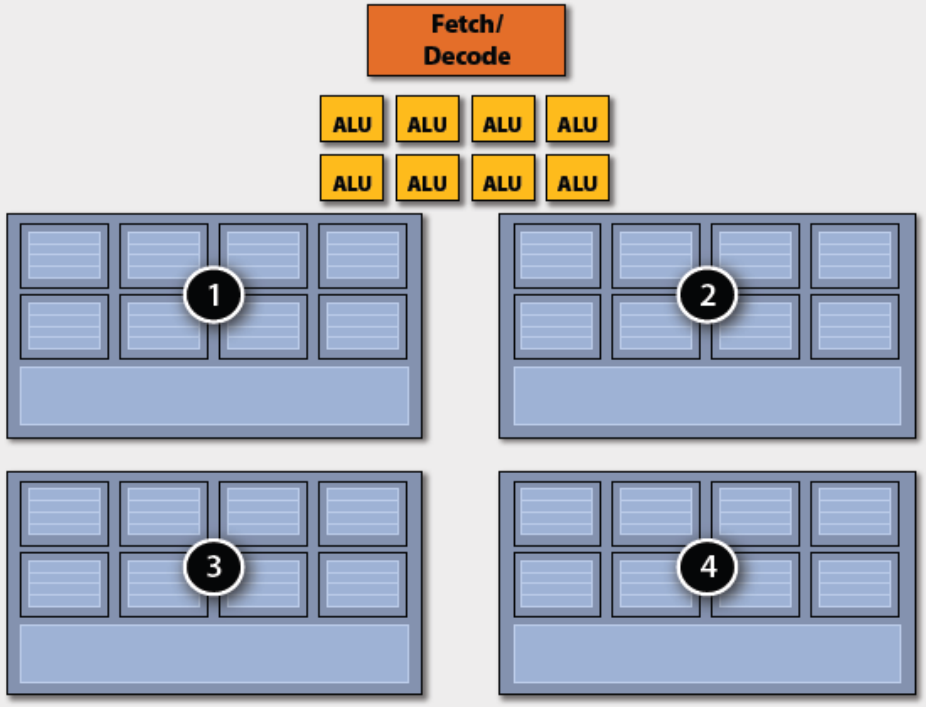
Unified Shader



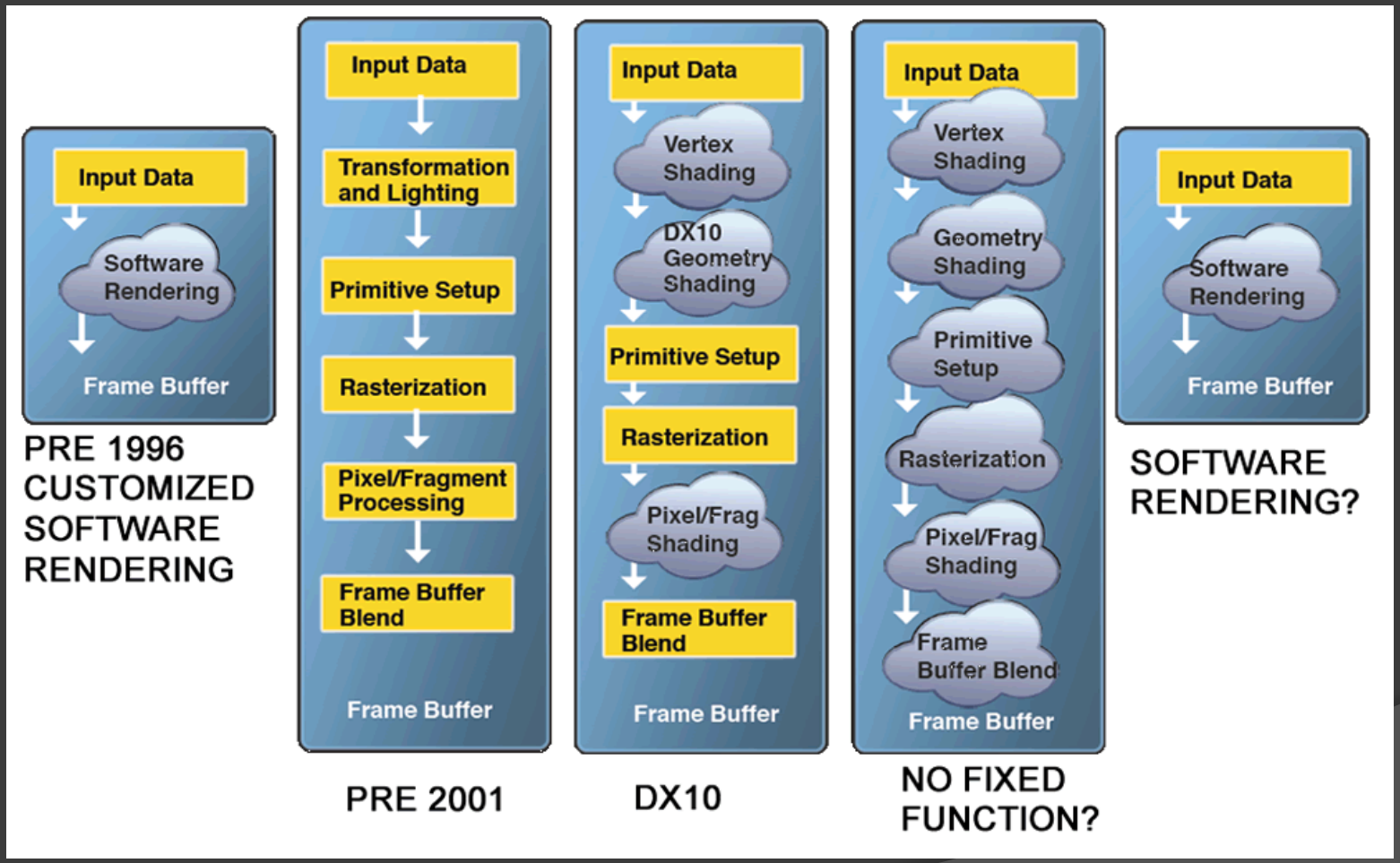
From CPU to GPU



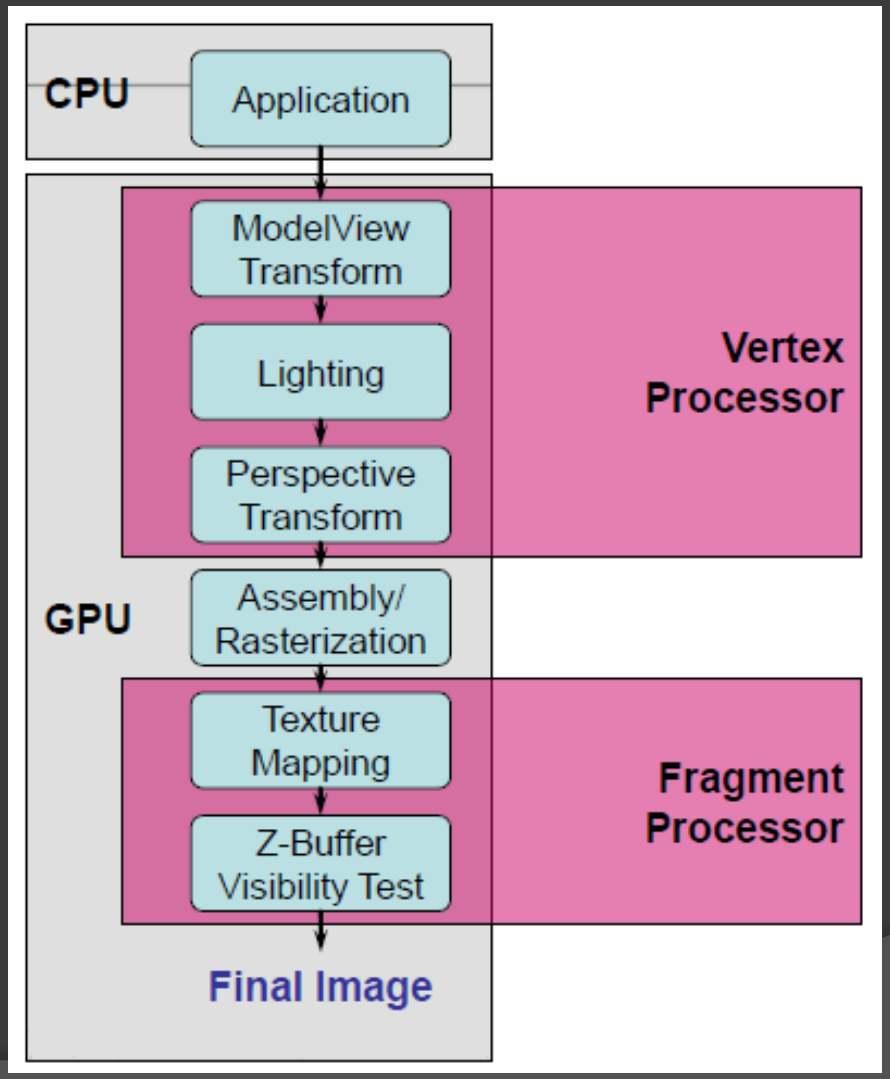
Three Key Ideas



Pipeline


Input Data
Software Rendering
Frame Buffer
PRE 1996 CUSTOMIZED SOFTWARE RENDERING
Input Data
Transformation and Lighting
Primitive Setup
Rasterization
Pixel/Fragment Processing
Frame Buffer Blend
Frame Buffer
PRE 2001
Input Data
Vertex Shading
DX10 Geometry Shading
Primitive Setup
Rasterization
Pixel/Frag Shading
Frame Buffer Blend
Frame Buffer
DX10
Input Data
Vertex Shading
Geometry Shading
Primitive Setup
Rasterization
Pixel/Frag Shading
Frame Buffer Blend
Frame Buffer
NO FIXED FUNCTION?
Input Data
Software Rendering
Frame Buffer
SOFTWARE RENDERING?

Vertex and Fragment

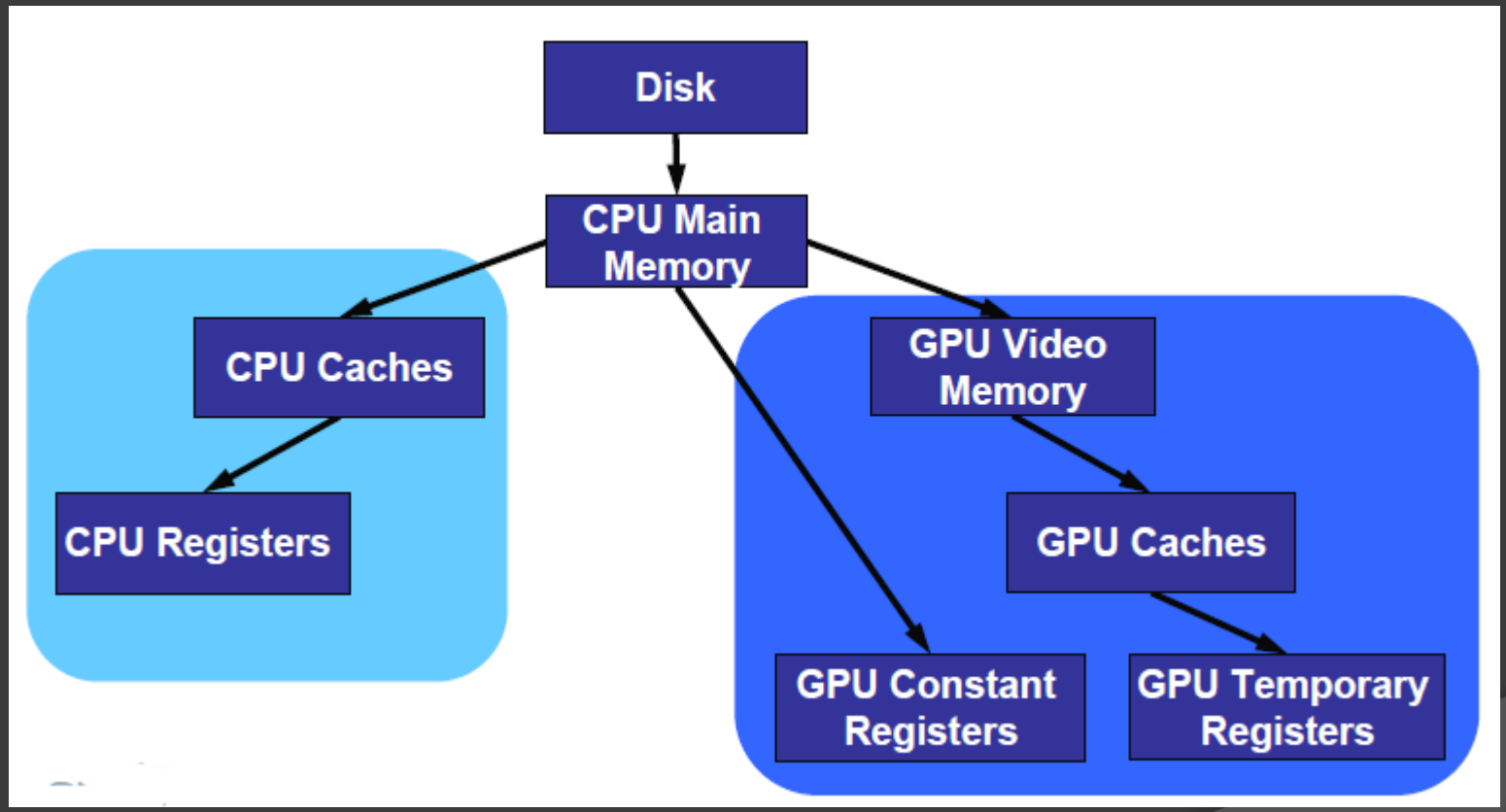


Where?

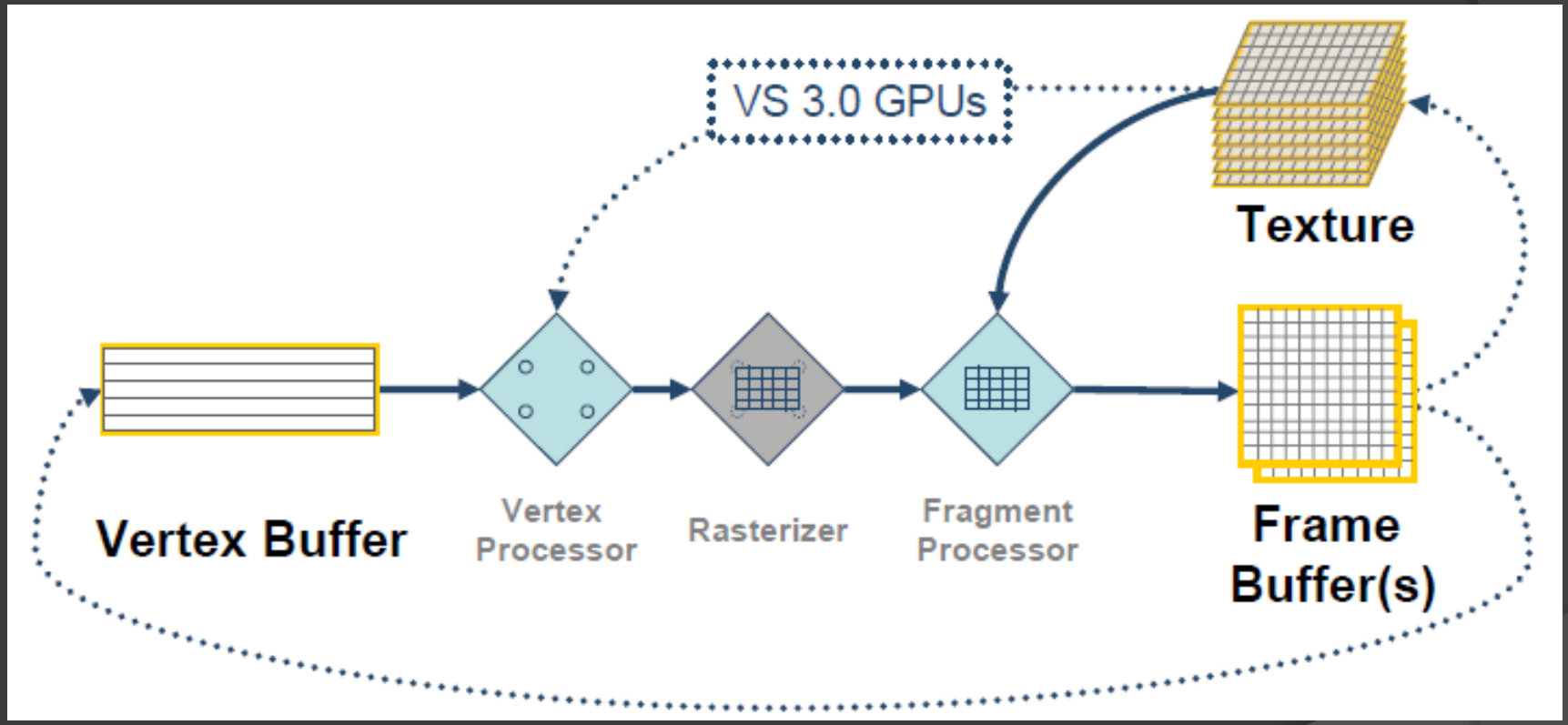
Geometry Processor

Why?

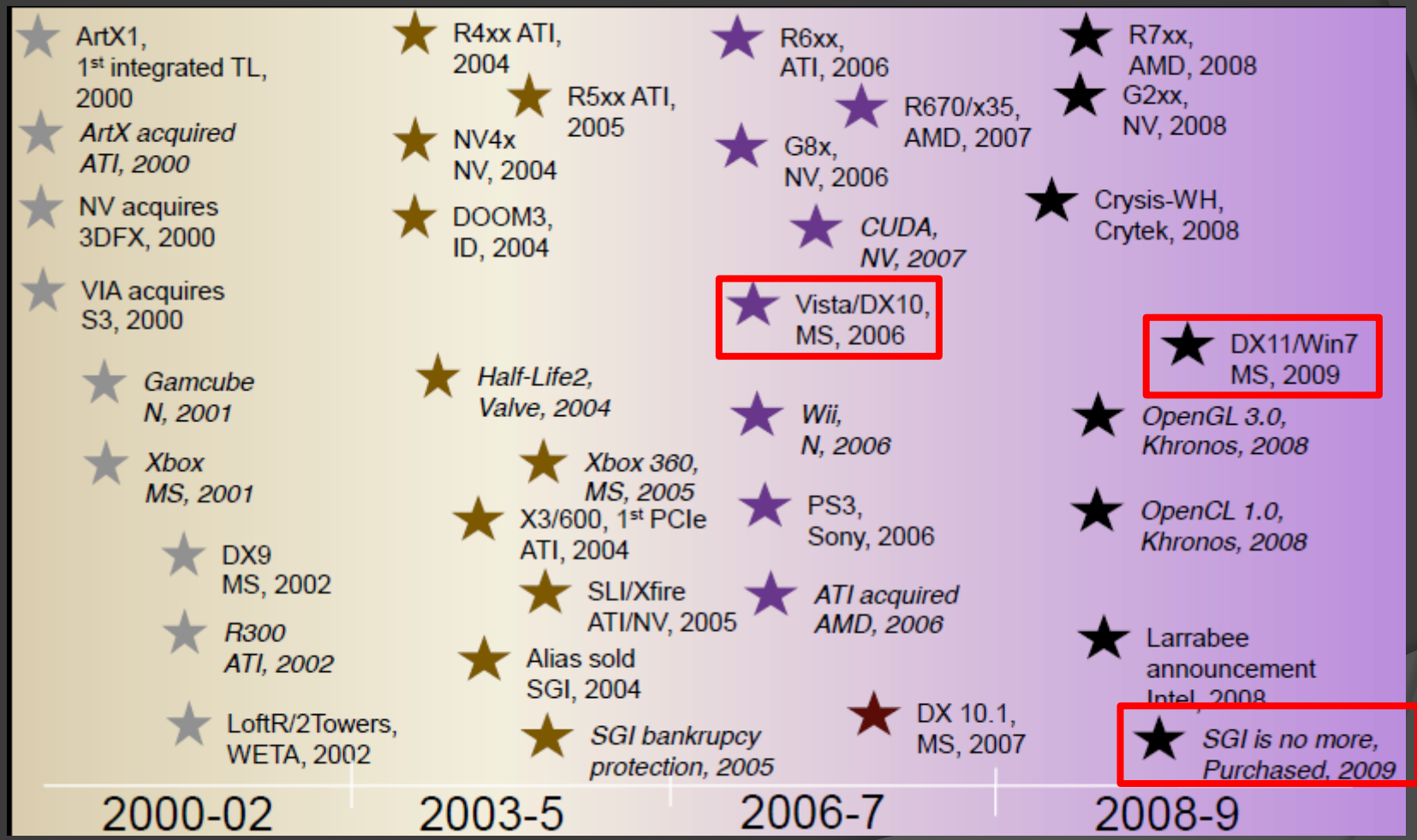
Memory Model



Memory Model

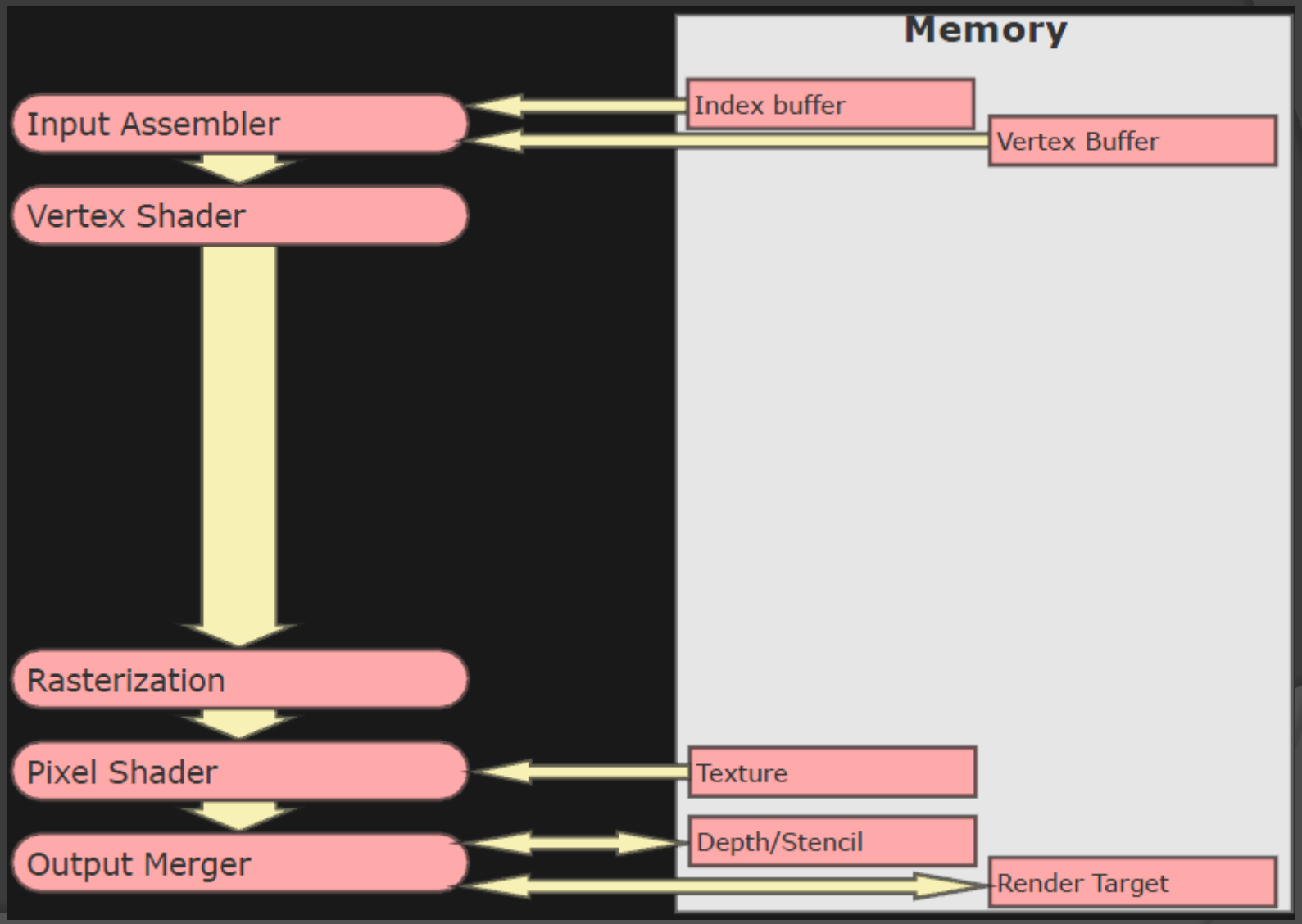


A quick history since 2000

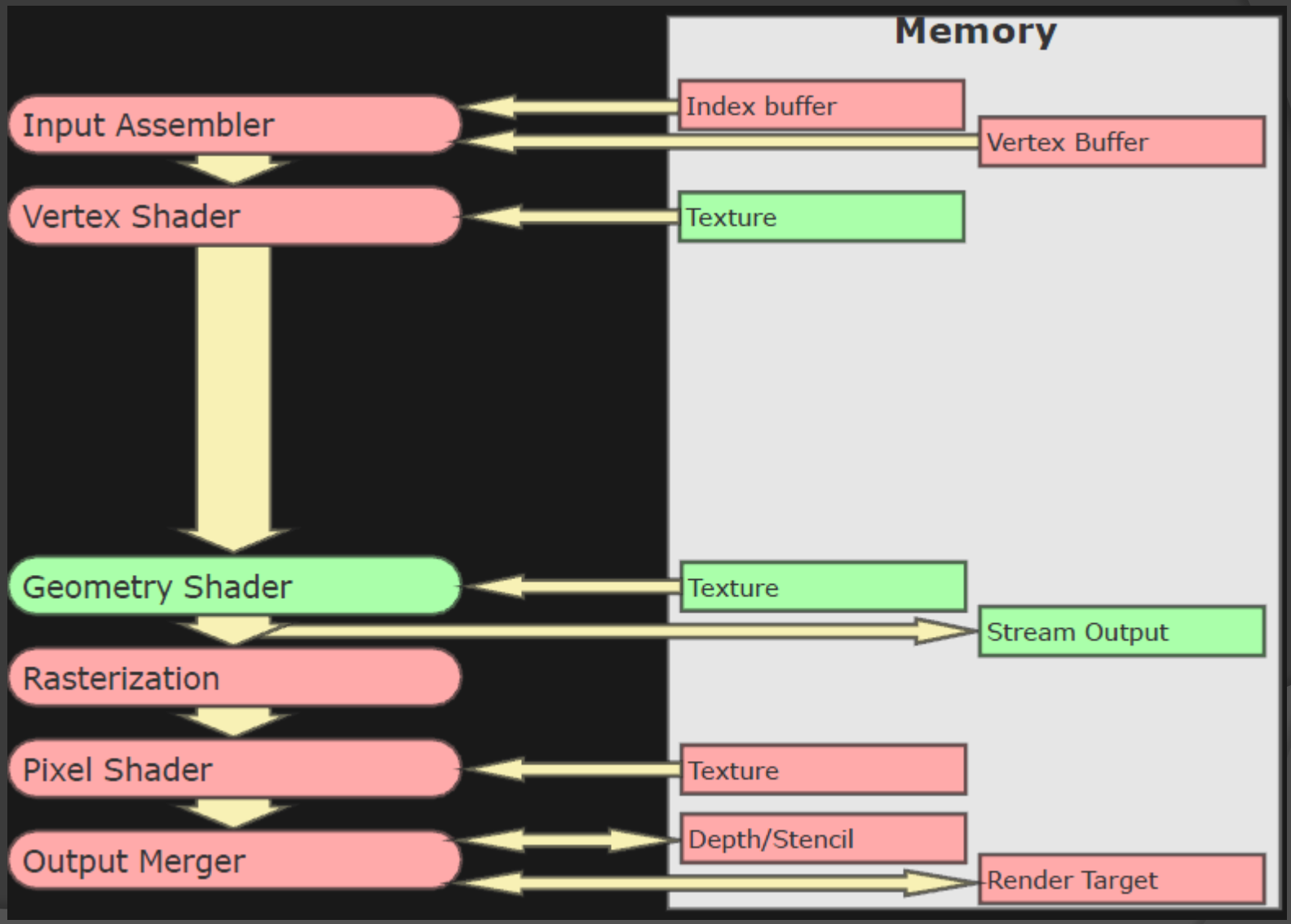


CHANGES TO THE PIPELINE

DX 9

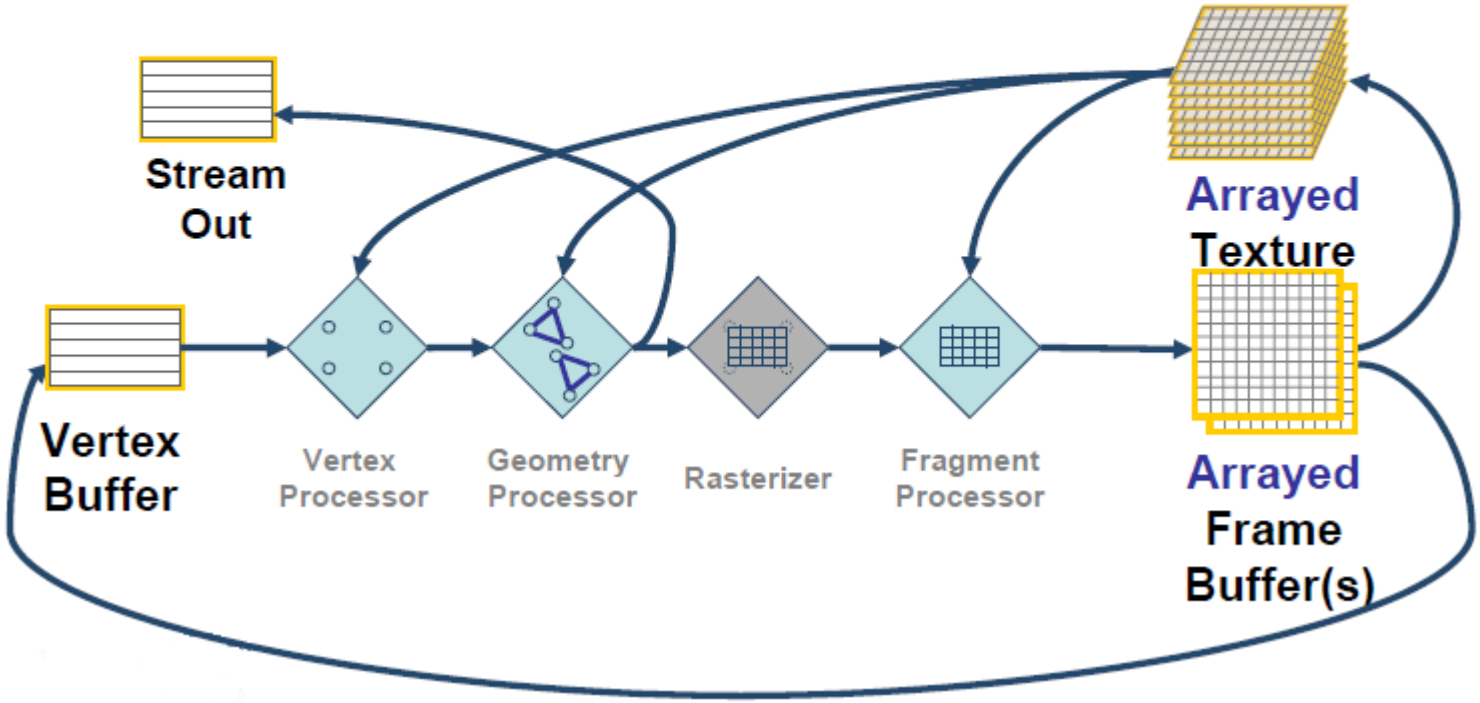

Memory
Index buffer
Vertex Buffer
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Texture
Depth/Stencil
Output Merger
Render Target

DX 10


Memory
Input Assembler
Vertex Shader
Geometry Shader
Rasterization
Pixel Shader
Output Merger
Index buffer
Vertex Buffer
Texture
Texture
Stream Output
Texture
Depth/Stencil
Render Target

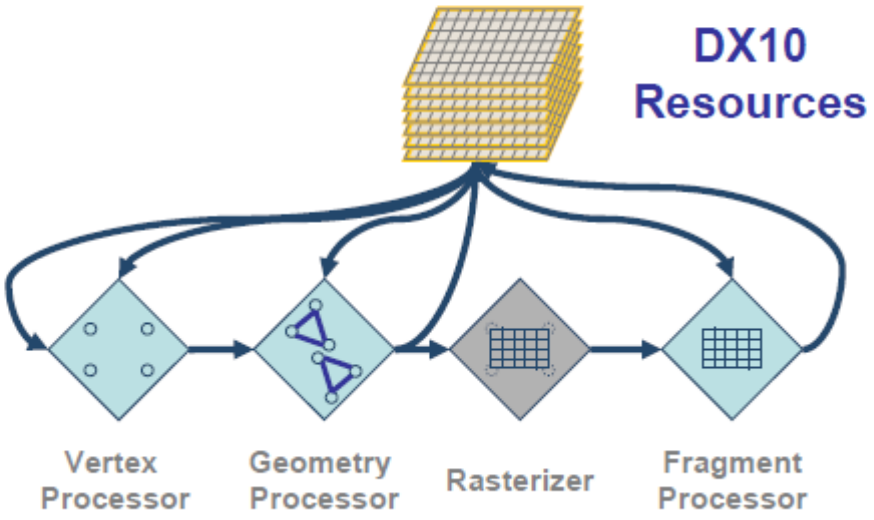
DX10 – Memory model

- **More flexible memory handling**
  - All programmable units can read texture
  - "Stream out" after geometry processor



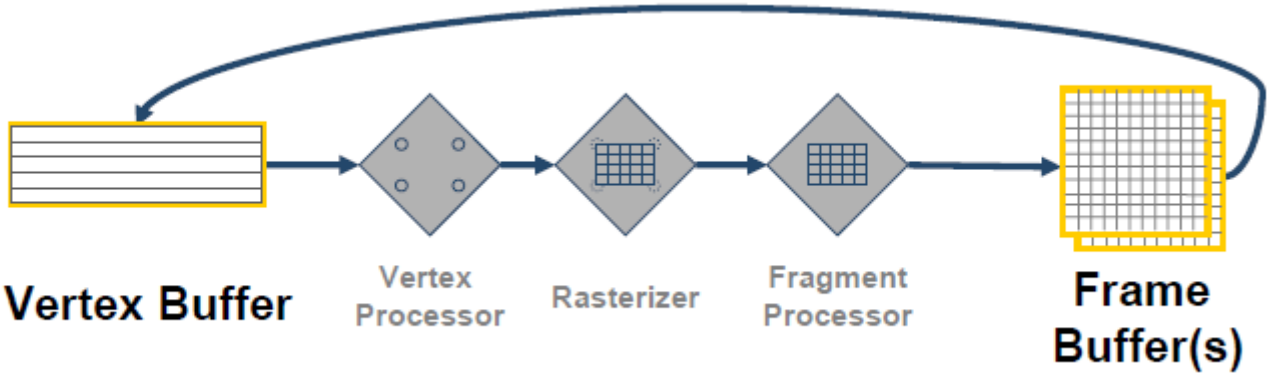
DX10 – Memory model

- DX10 provides “resources”
- Resources are flexible!



DX10 – Memory model

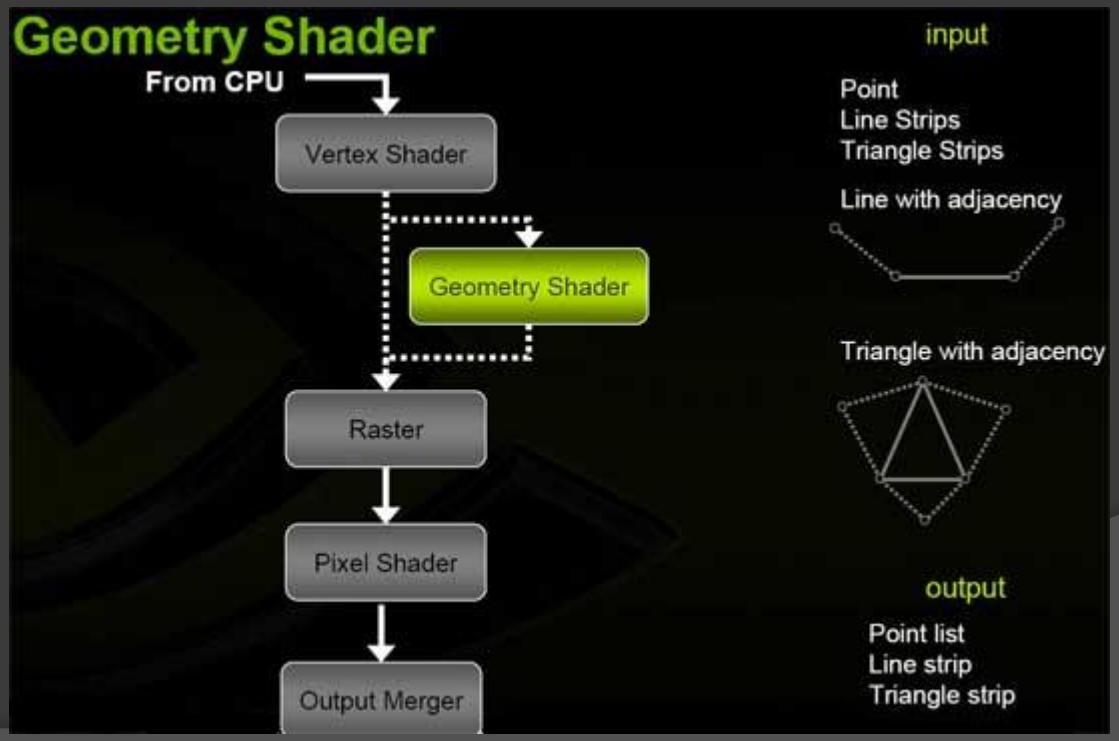
- Enables top-of-pipe feedback loop
- Enables dynamic creation of geometry on GPU



GEOMETRY SHADER

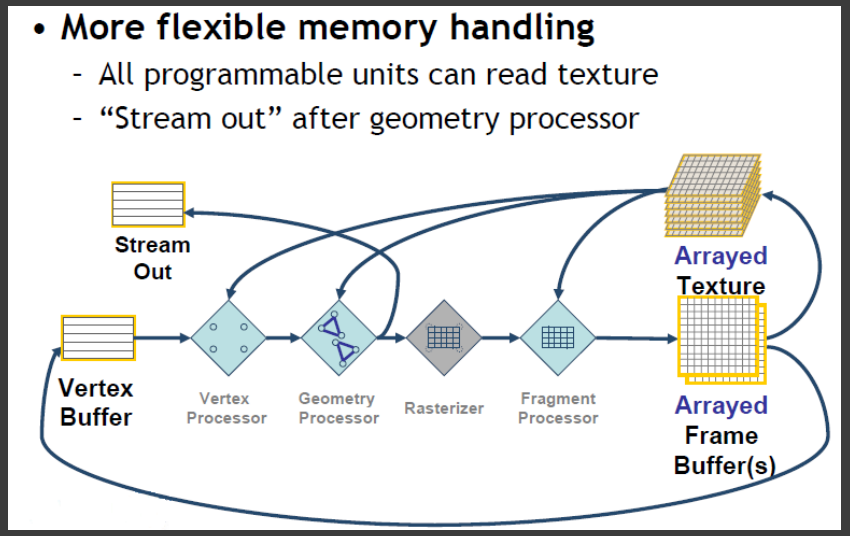
Geometry Shader

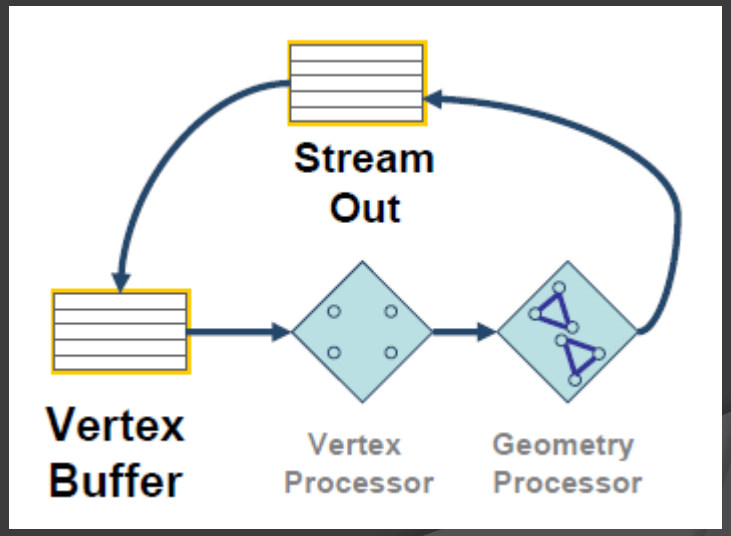
- •Input
  - Vertices of a *fullprimitive (1 for point, 2 for line, 3 for triangle)*
  - Vertices of edge-*adjacentprimitives*
- •Output
  - Point-, line- or triangle-strip
  - Limited output size



Stream Out to Vertex Buffer

- **• Enabled by DX10 StreamOut capability**
- **• Expected to be used for dynamic geometry**
  - Geometry processor(shader) produces **0-n** outputs per input



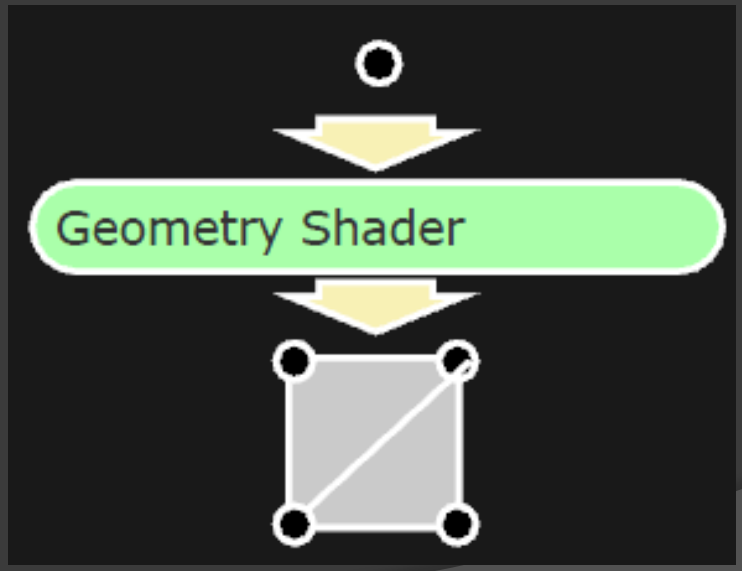


Usage examples

- Tessellation;
- Point Sprite Tessellation;
- Wide Line Tessellation;
- High Performance Culling;

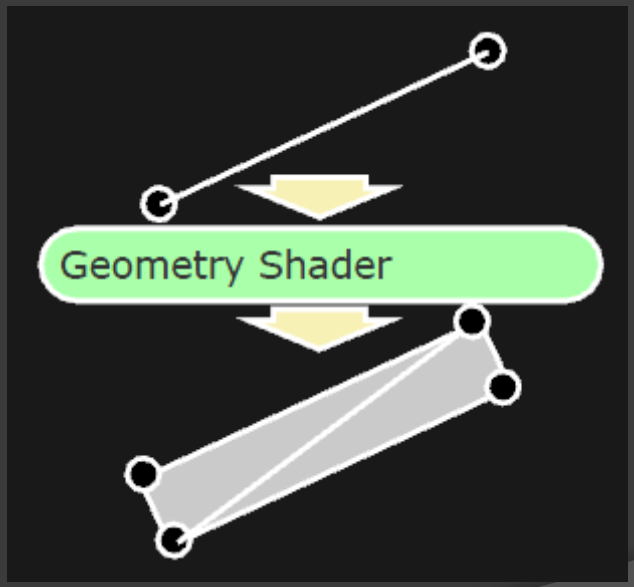
Point Sprite Tessellation

- The shader takes in a single vertex and generates four vertexes (two output triangles) that represent the four corners of a quad



Wide Line Tessellation

- The shader receives two line vertexes and generates four vertexes for a quad that represents a widened line.



Examples

- Smoke represented by point sprites
  - Points converted to quads in geometry shader
- Hair represented by “wide lines”
  - Lines converted to quads in geometry shader

High Performance Culling

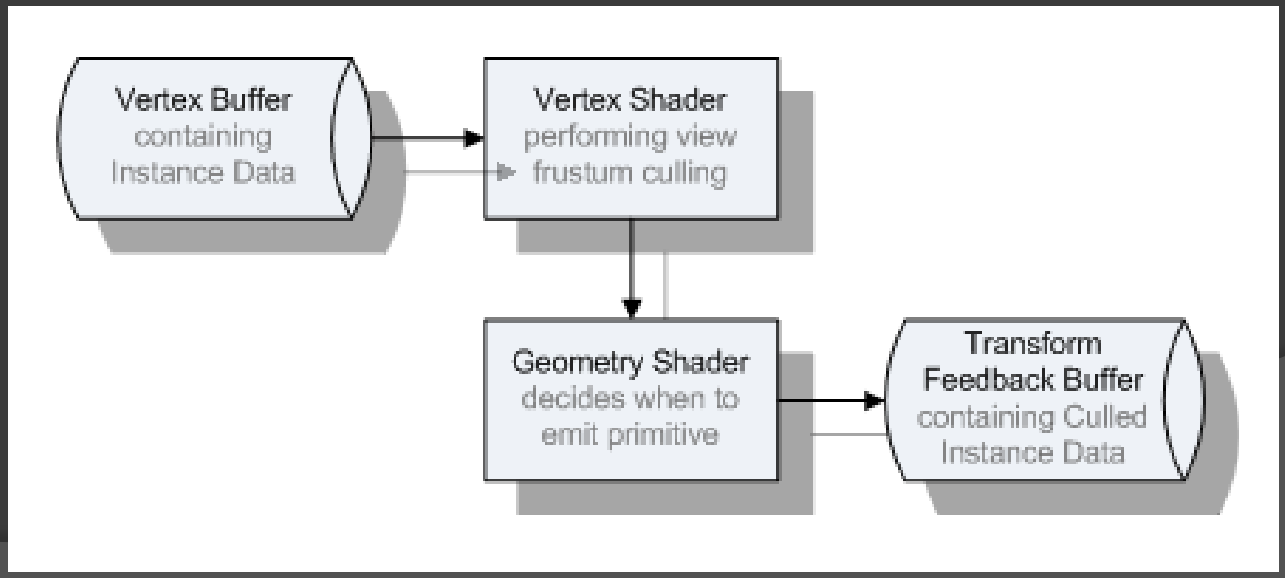
- If we take a closer look at the geometry shader we observe that the most revolutionary in it is not that it can raise the number of emitted primitives but that it can discard them.

Instance Cloud Reduction

- It is a multi-pass technique that in the first pass culls the object instances against the view frustum using the GPU and in the second pass renders only those instances that are likely to be visible in the final scene.
- it does not produce any fragments in the first pass, instead the first pass does the view frustum culling.

Pass 1: Culling

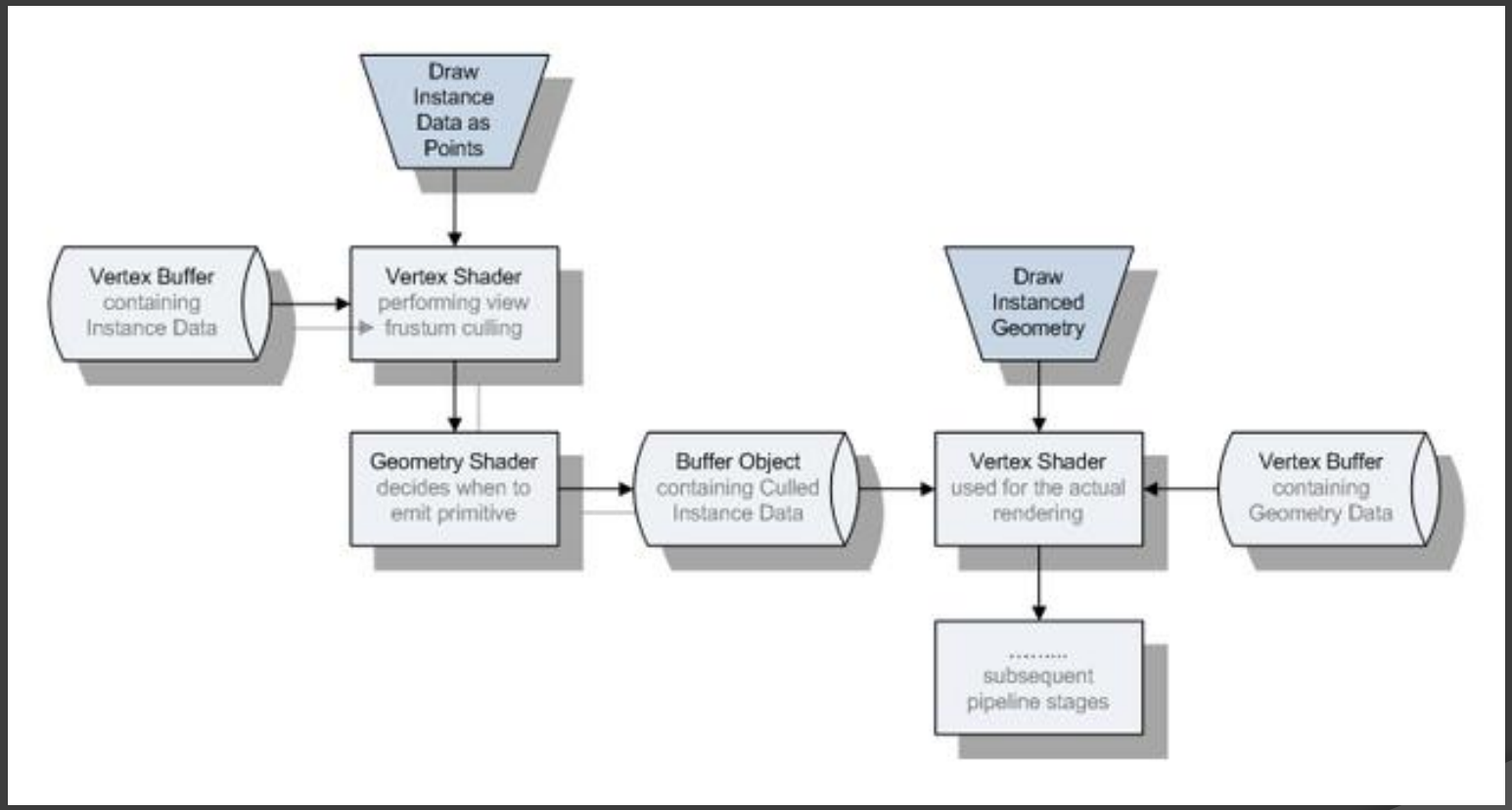
- In the first pass the vertex shader determines whether the actual object instance's bounding volume is inside the view frustum and sends a flag about the culling to the geometry shader.
- The geometry shader will emit the instance data to the destination buffer if the flag says that the instance is likely to be visible or does not emit anything if it is determined that the object instance is out of view.



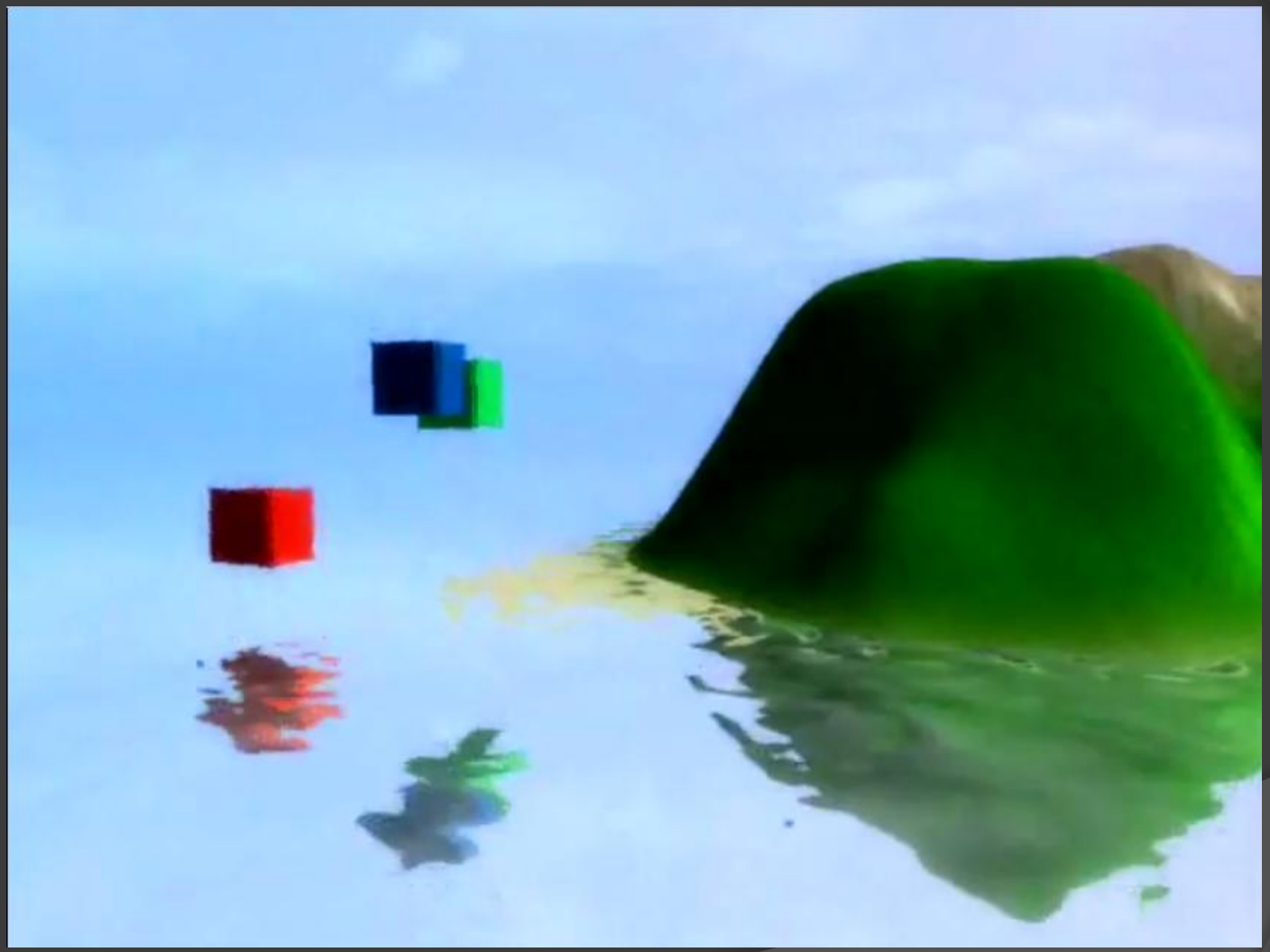
Pass 2: Rendering

- In the second pass there is nothing special to do. The only things that need to be changed is that the instance data for the rendering must be sourced from the generated culled instance data buffer.

Pass 1 & 2

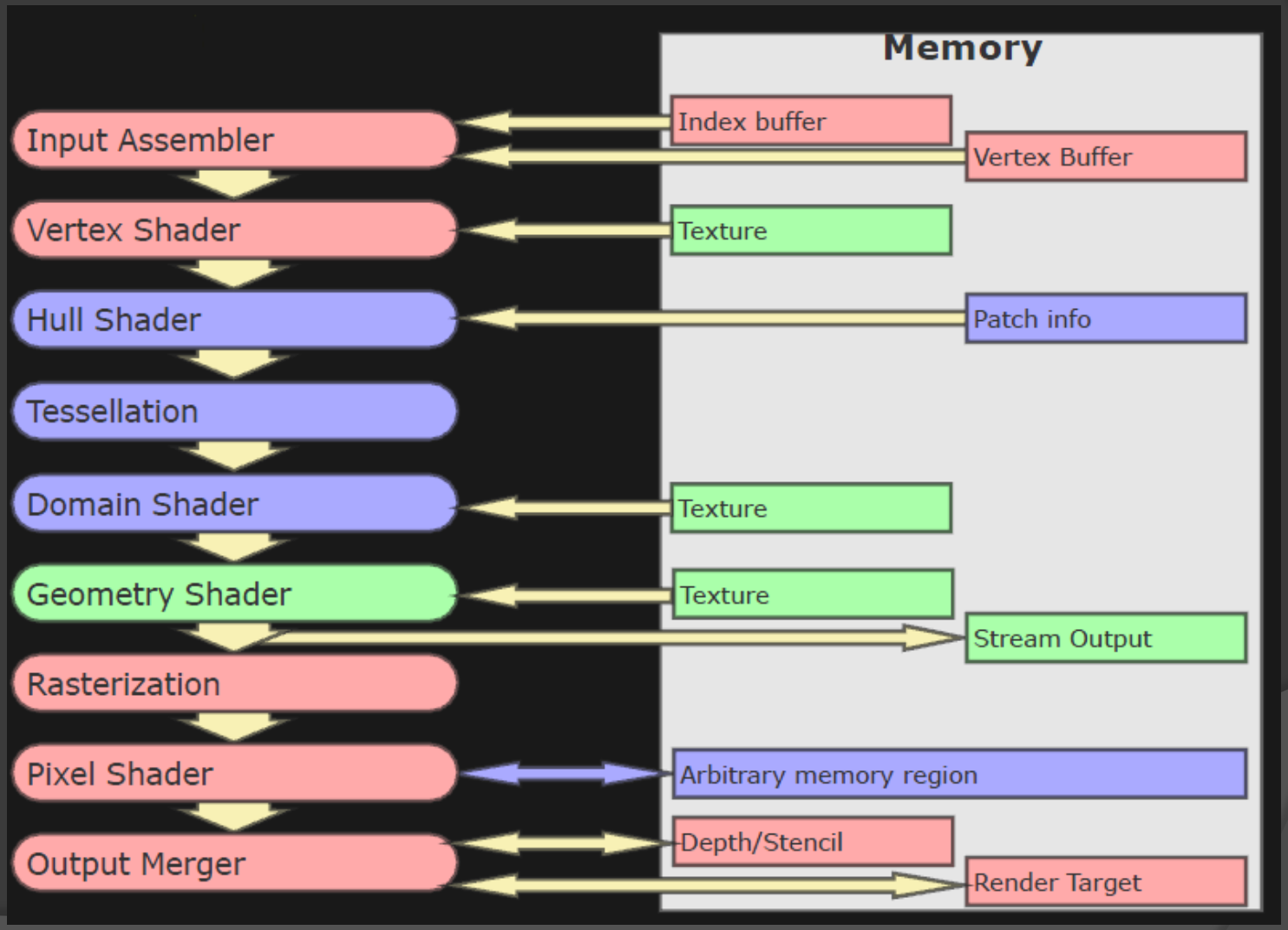

Draw Instance Data as Points
Vertex Buffer containing Instance Data
Vertex Shader performing view frustum culling
Geometry Shader decides when to emit primitive
Buffer Object containing Culled Instance Data
Draw Instanced Geometry
Vertex Shader used for the actual rendering
Vertex Buffer containing Geometry Data
..... subsequent pipeline stages

TRIANGLE SHRINK

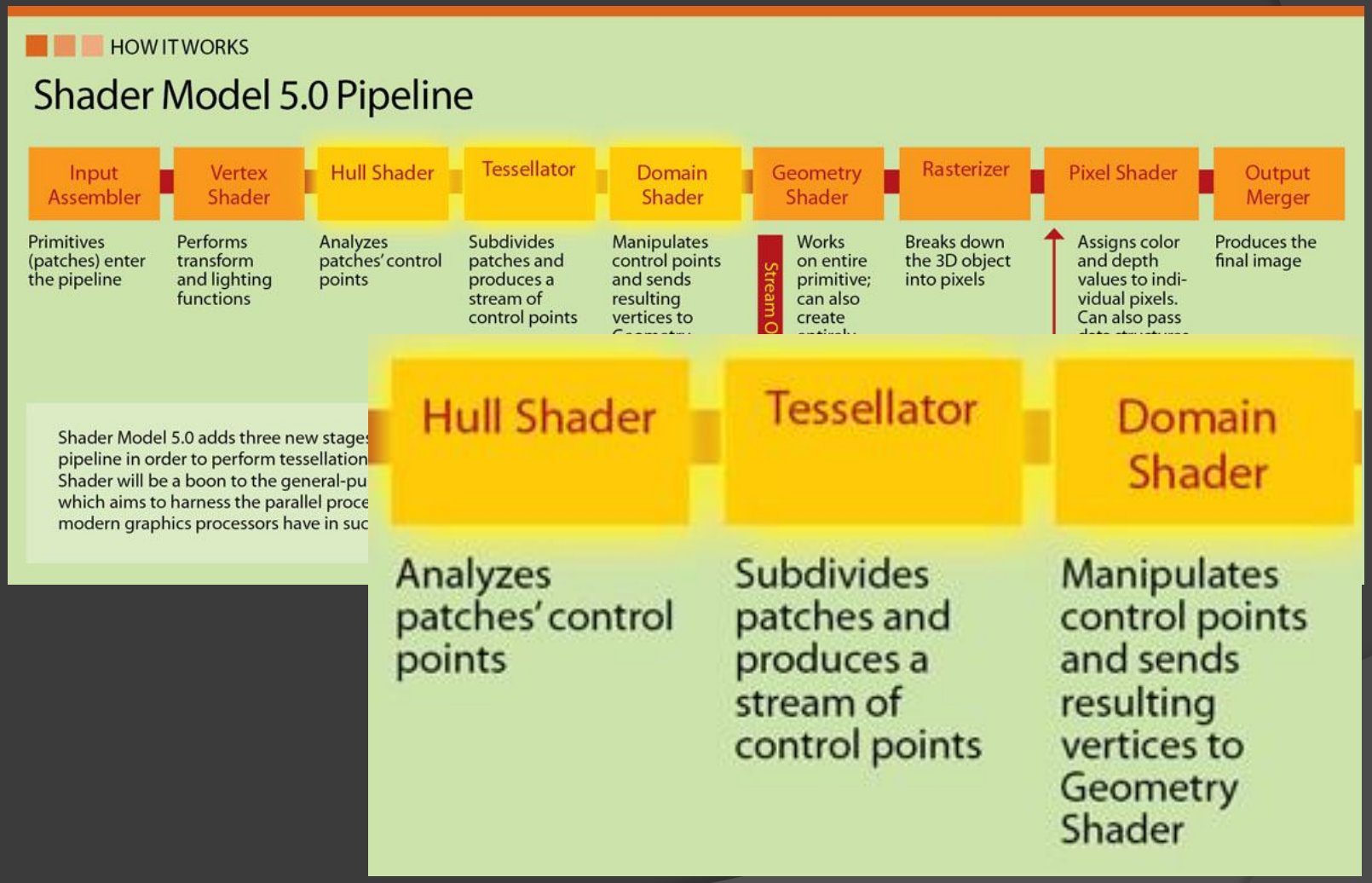


DX11

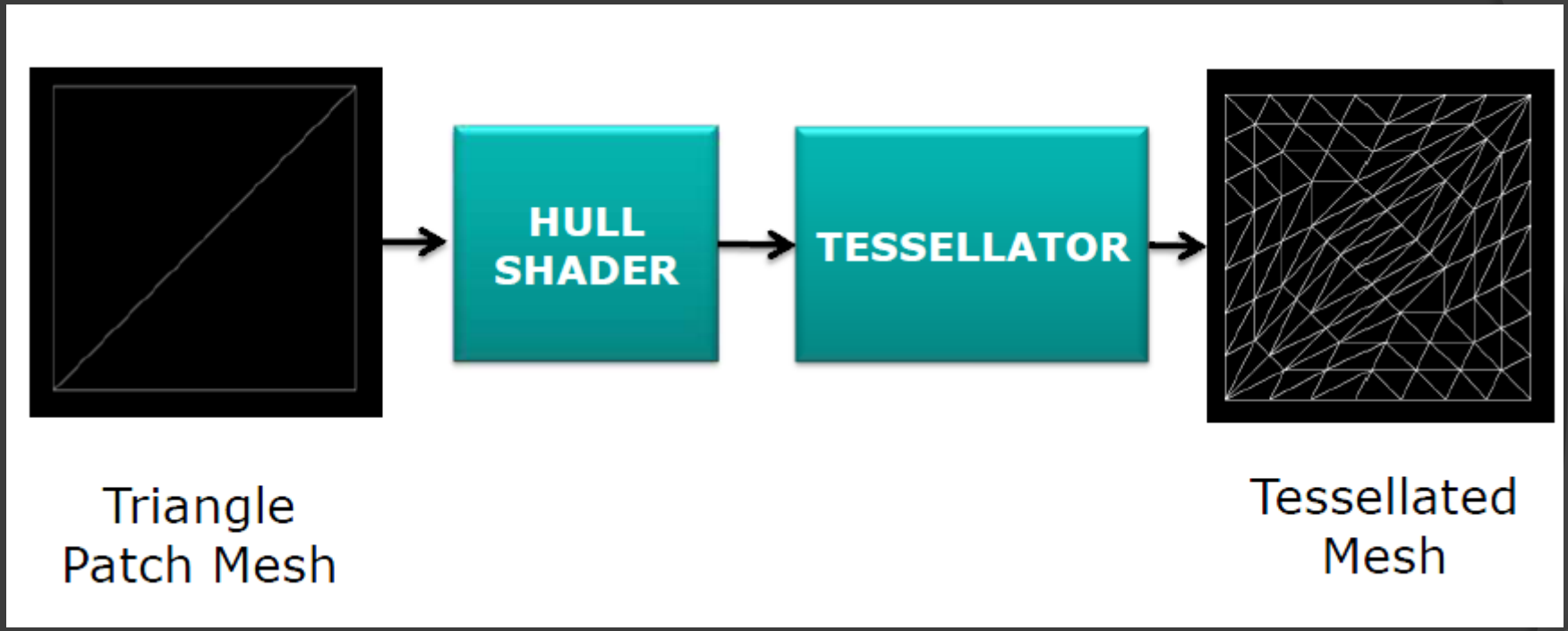
DX11


Memory
Input Assembler
Vertex Shader
Hull Shader
Tessellation
Domain Shader
Geometry Shader
Rasterization
Pixel Shader
Output Merger
Index buffer
Vertex Buffer
Texture
Patch info
Texture
Texture
Stream Output
Arbitrary memory region
Depth/Stencil
Render Target

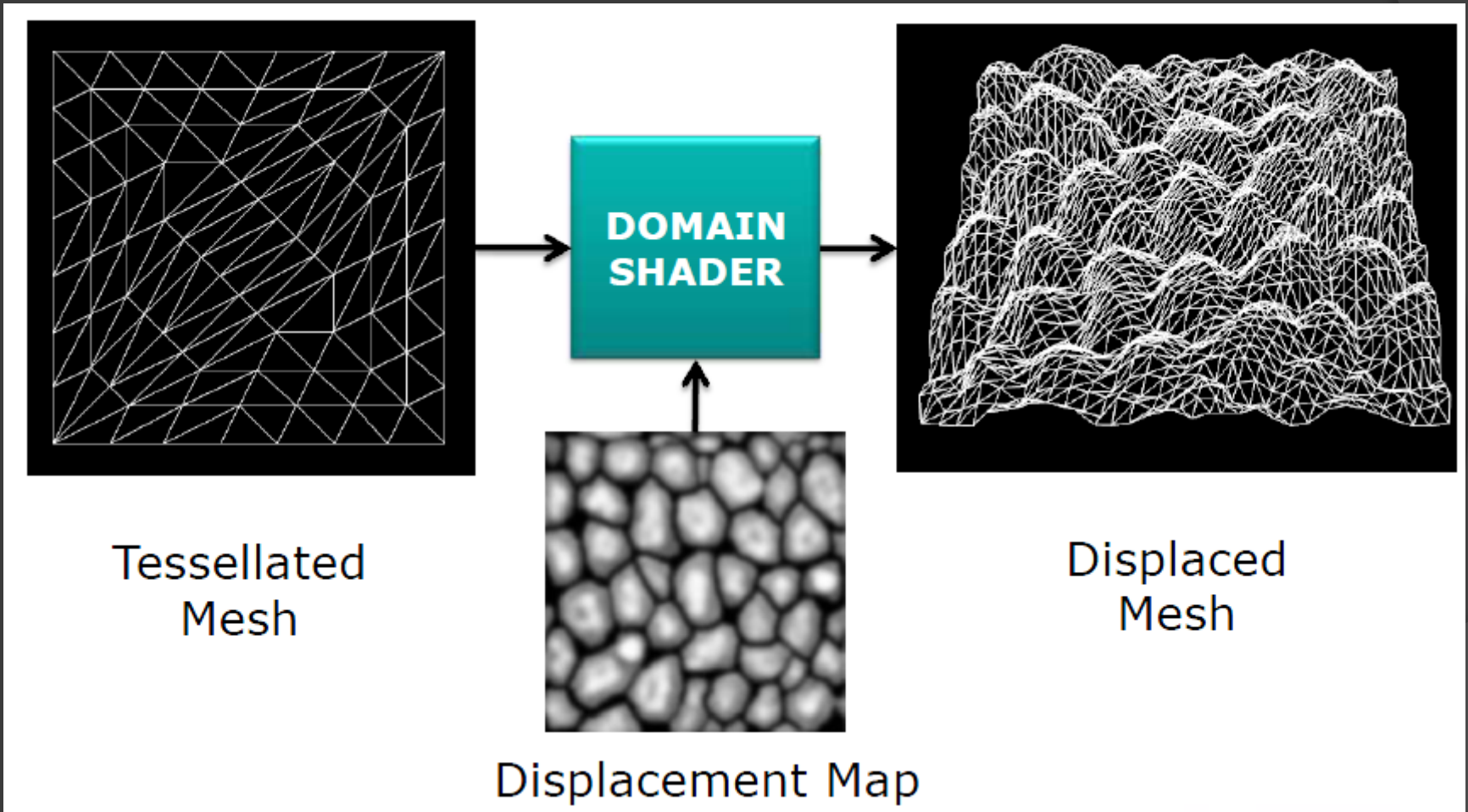
DX11


HOW IT WORKS
Shader Model 5.0 Pipeline
Input Assembler
Vertex Shader
Hull Shader
Tessellator
Domain Shader
Geometry Shader
Rasterizer
Pixel Shader
Output Merger
Primitives (patches) enter the pipeline
Performs transform and lighting functions
Analyzes patches' control points
Subdivides patches and produces a stream of control points
Manipulates control points and sends resulting vertices to
Works on entire primitive; can also create
Breaks down the 3D object into pixels
Assigns color and depth values to individual pixels. Can also pass
Produces the final image
Shader Model 5.0 adds three new stages
pipeline in order to perform tessellation
Shader will be a boon to the general-pu
which aims to harness the parallel proce
modern graphics processors have in suc
Hull Shader
Tessellator
Domain Shader
Analyzes patches' control points
Subdivides patches and produces a stream of control points
Manipulates control points and sends resulting vertices to Geometry Shader

Tessellation

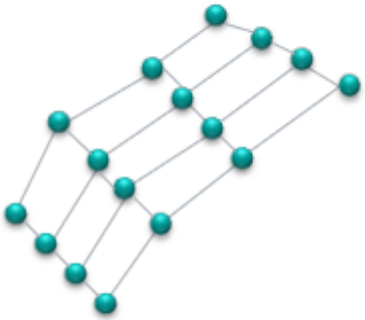


Tessellation

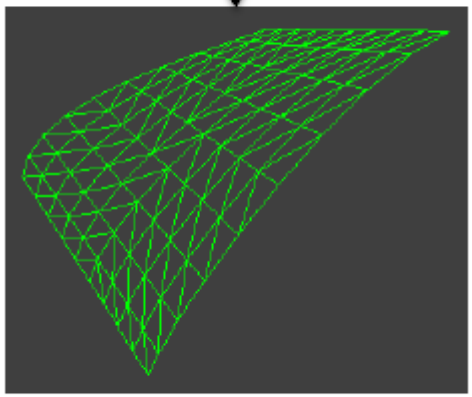


Bezier Patch

Bicubic Bezier Patch Example



Bezier Control Points



Tessellated Bezier Patch

$$Pos(u,v) = \sum_{m=0}^{3} \sum_{n=0}^{3} B_m(u)\, B_n(v)\, G$$

Where: $B_0 = (1-t)^3$
$B_1 = 3t(1-t)^2$
$B_2 = 3t^2(1-t)$
$B_3 = t^3$

and $G = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$

References

- Beyond Programmable Shading SIGGRAPH 2009
- John Owens, University of California ,GPU Memory Model Overview
- DirectX10 Tutorial 9: The Geometry Shader: http://takinginitiative.net/2011/01/12/directx10-tutorial-9-the-geometry-shader/
- Geometry Shader: http://www.notjustcode.it/dblog/articolo.asp?articolo=282
- GLSL Geometry Shader "Triangle Shrink“: http://www.youtube.com/watch?v=IRzAxtBWtV8
- Instance culling using geometry shaders: http://rastergrid.com/blog/2010/02/instance-culling-using-geometry-shaders/
- Bill Bilodeau, 2010, Direct3D 11 Tutorial: Tessellation
- Michael Brown, White Paper DirectX 11: http://www.maximumpc.com/article/features/white_paper_directx_11