# Computer Graphics

by Ruen-Rone Lee
Associate Researcher
CS/NTHU

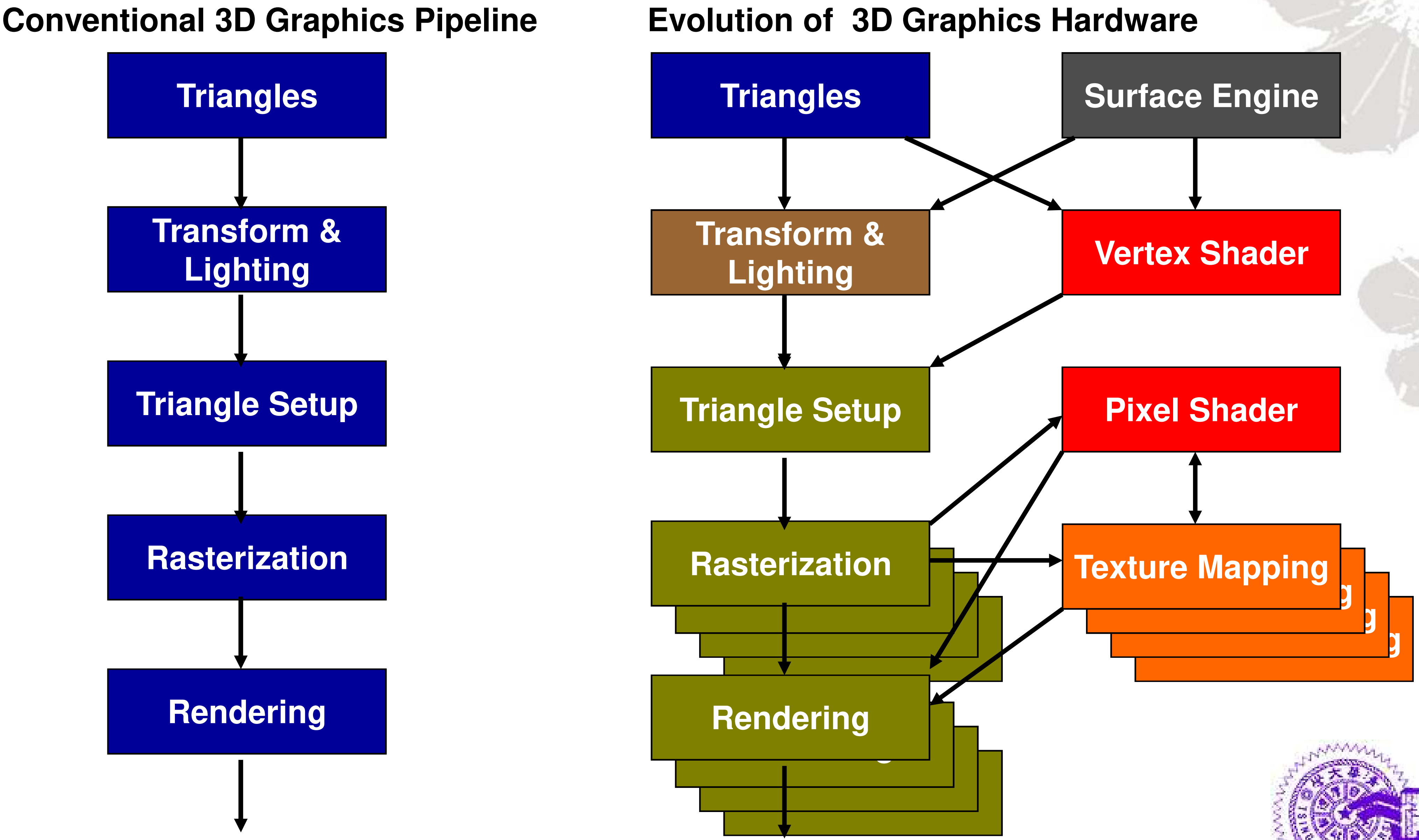
# *Wrap up from last class*

- Fragment Operations
  - Fragment Tests
    - Pixel ownership test
    - Scissor test
    - Alpha test
    - Stencil test
    - Depth test
  - Blending
    - Fog blending
    - Color blending

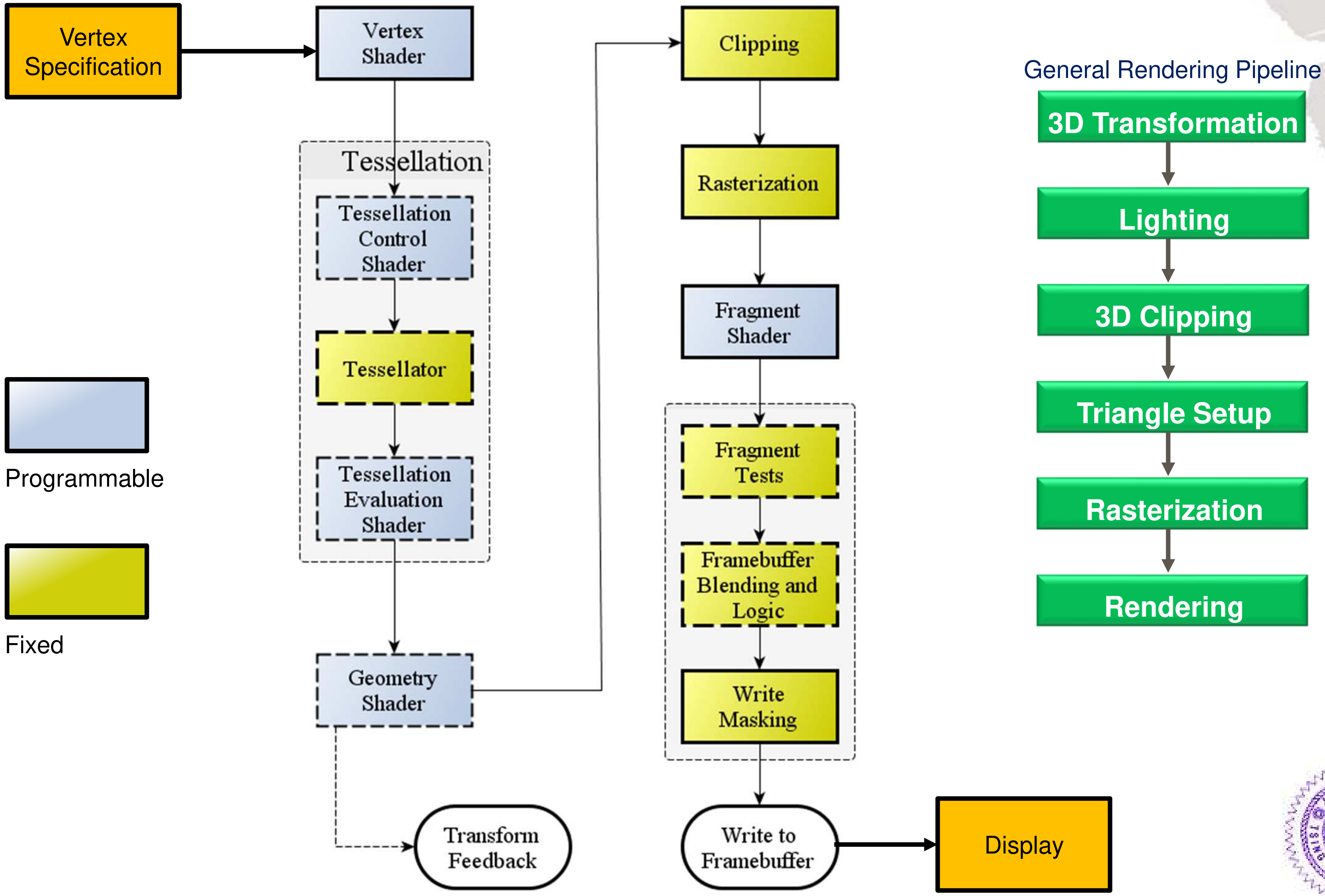
# 3D Graphics Pipeline Revisit

OpenGL Pipeline

# 3D Graphics Hardware Evolution



# Mixed Pipeline (Fixed+Programmable)



Vertex Specification

Vertex Shader

Tessellation

Tessellation Control Shader

Tessellator

Tessellation Evaluation Shader

Geometry Shader

Transform Feedback

Clipping

Rasterization

Fragment Shader

Fragment Tests

Framebuffer Blending and Logic

Write Masking

Write to Framebuffer

Display

Programmable

Fixed

General Rendering Pipeline

3D Transformation

Lighting

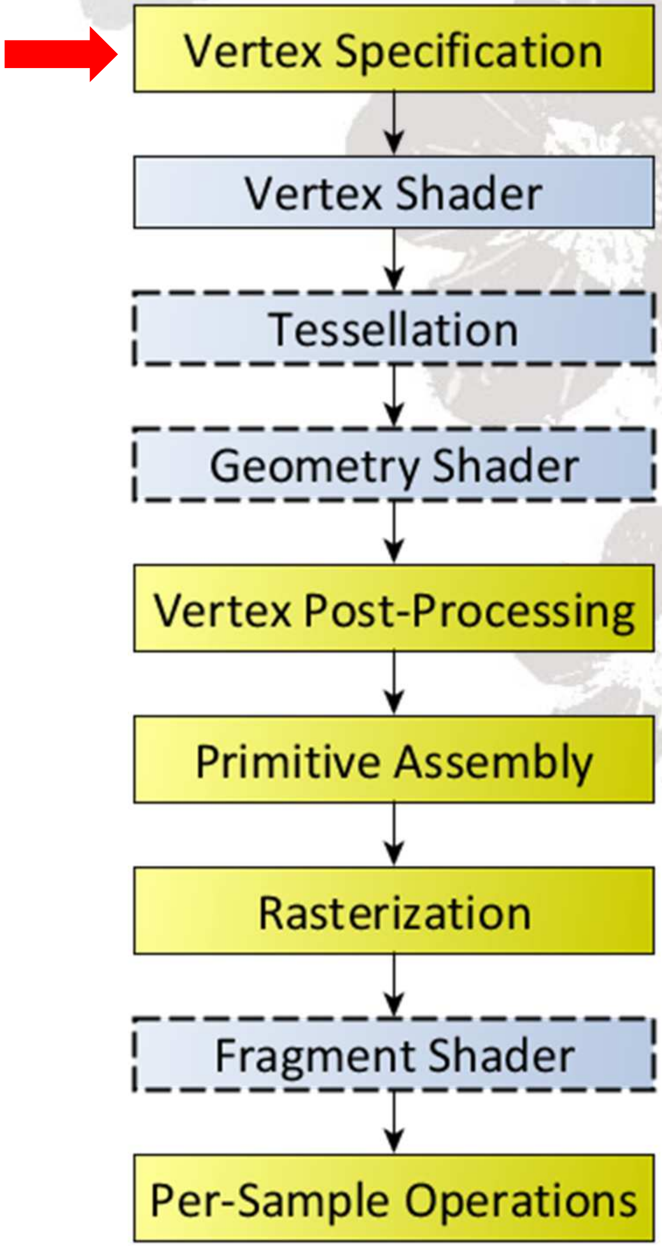
3D Clipping

Triangle Setup

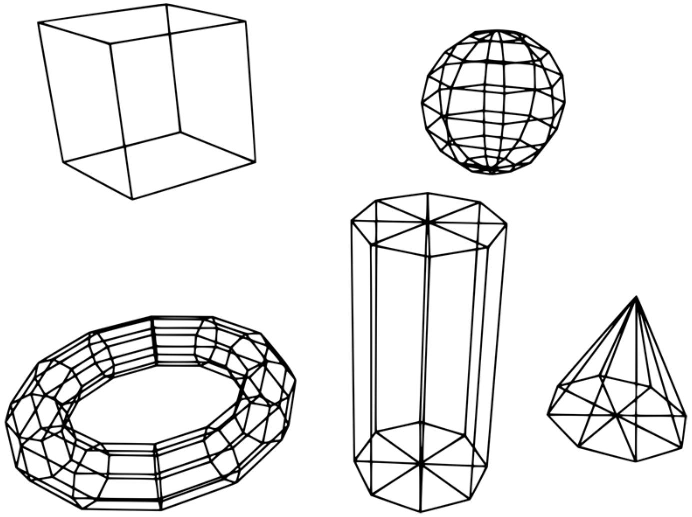
Rasterization

Rendering

# *Vertex Specification*



- **Sets up an ordered list of vertices to send to the pipeline**
  - **Prepared in application level**
  - **Vertex attributes: position, normal, color, texture coordinate,…**

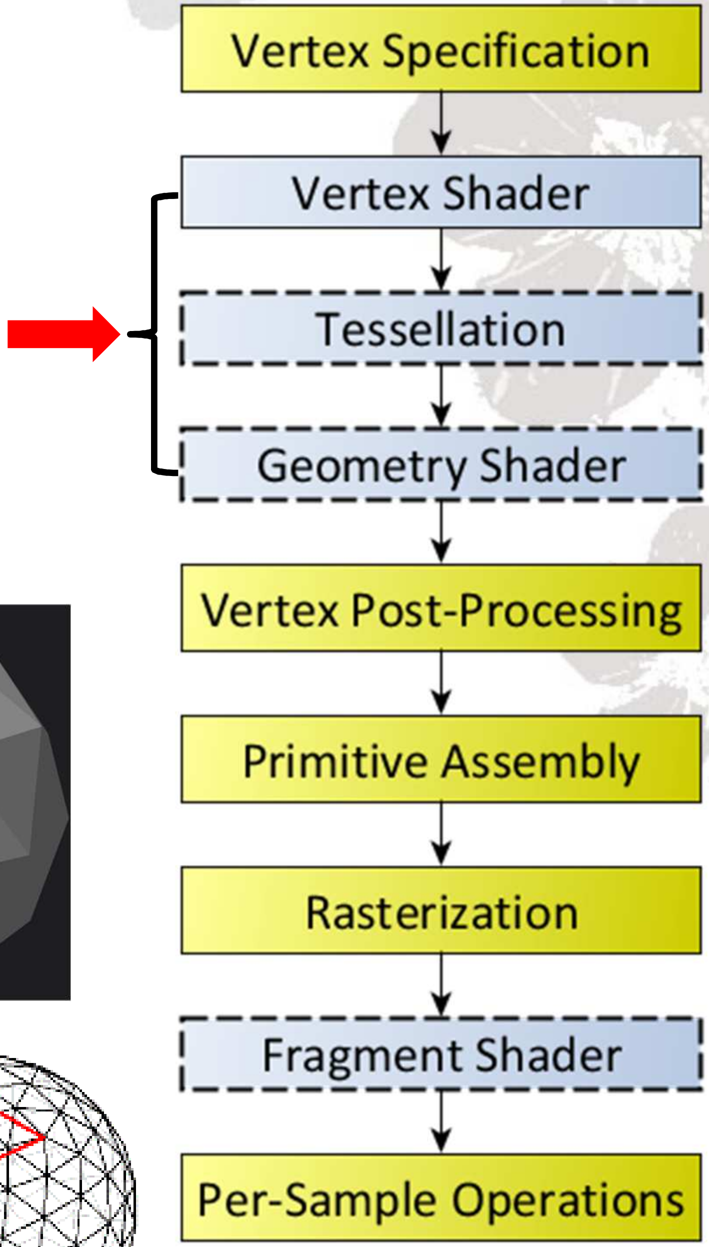


**Vertex attributes**

Position: (x, y, z)
Normal: (nx, ny, nz)
Color: (r, g, b)
Texture: (u, v)
Fog: (f)
…

# Vertex Processing

- Process vertex data according to specific vertex rendering
  - Vertex Transformation
  - Vertex Lighting
  - Primitive Tessellation
  - Vertex Displacement



Vertex Specification

Vertex Shader

Tessellation

Geometry Shader

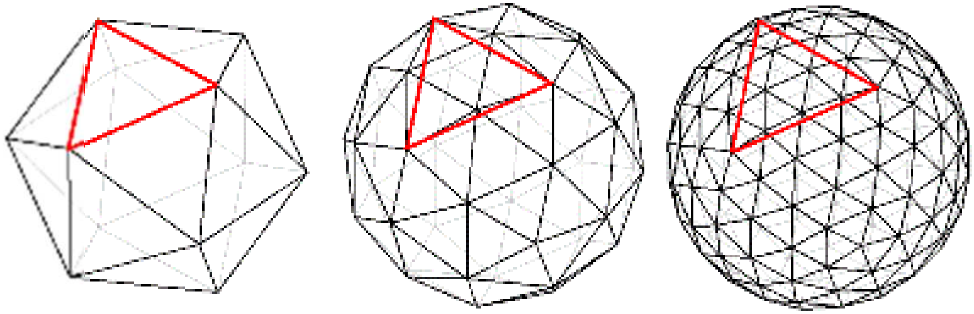
Vertex Post-Processing

Primitive Assembly

Rasterization

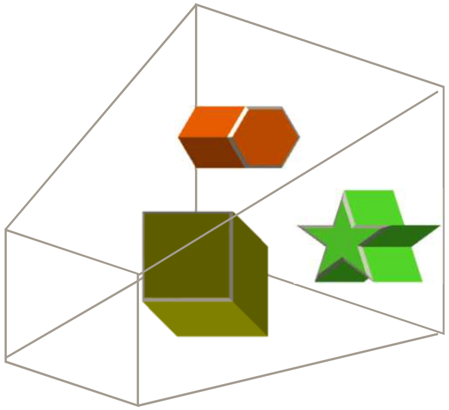
Fragment Shader

Per-Sample Operations

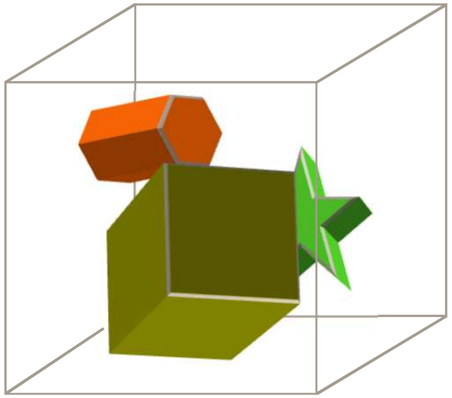


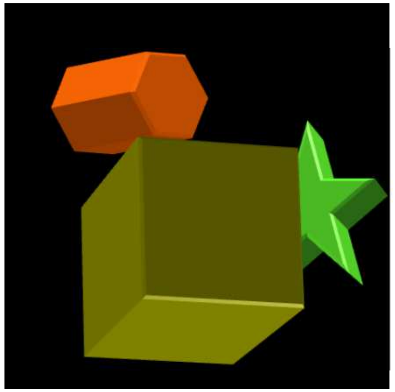
# Vertex Post-Processing

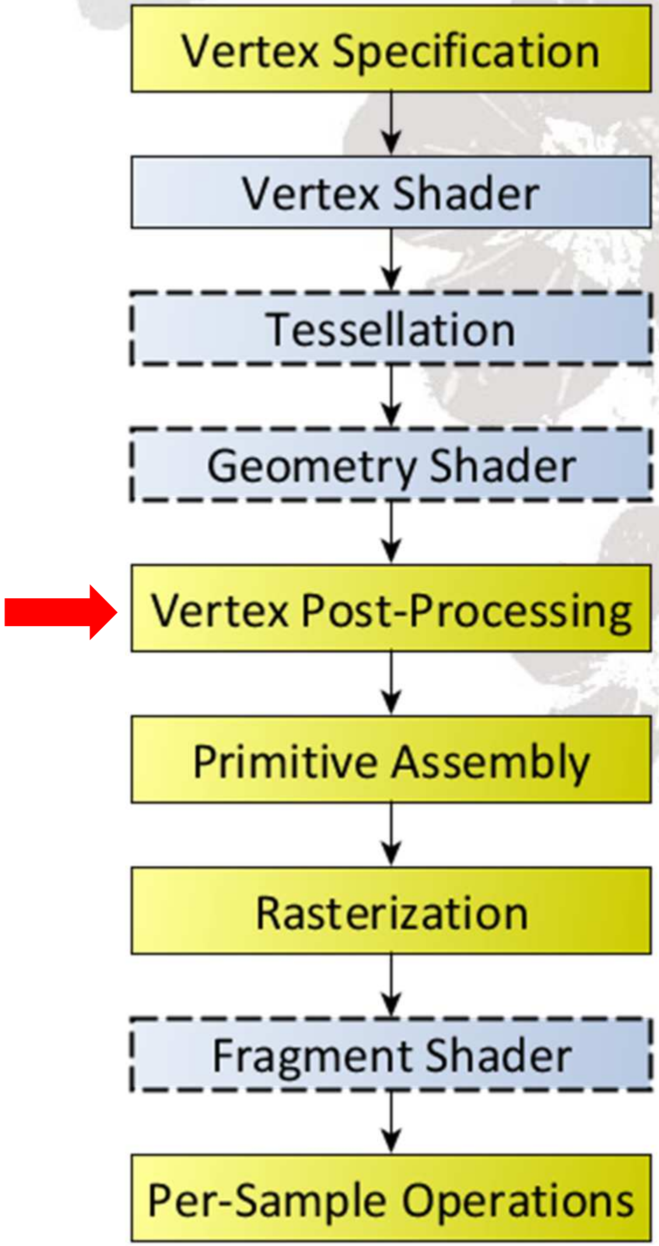
- Process vertex data after vertex processing
  - Transform feedback
  - Perspective division
  - Clipping
  - Viewport mapping



$$\begin{bmatrix}x\\y\\z\\w\end{bmatrix} = \begin{bmatrix}x/w\\y/w\\z/w\\1\end{bmatrix}$$







Vertex Specification

Vertex Shader

Tessellation

Geometry Shader

Vertex Post-Processing

Primitive Assembly

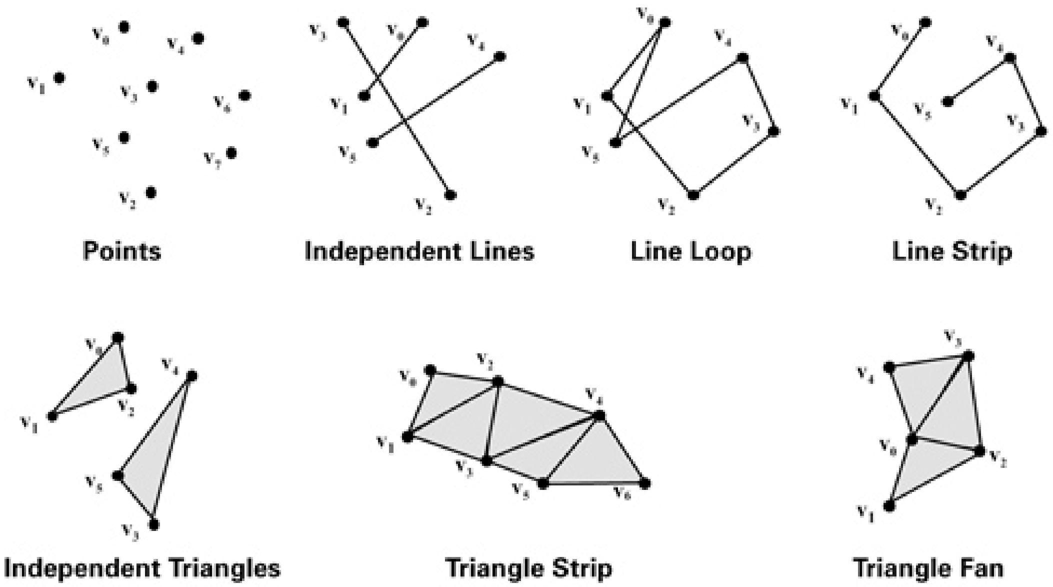
Rasterization

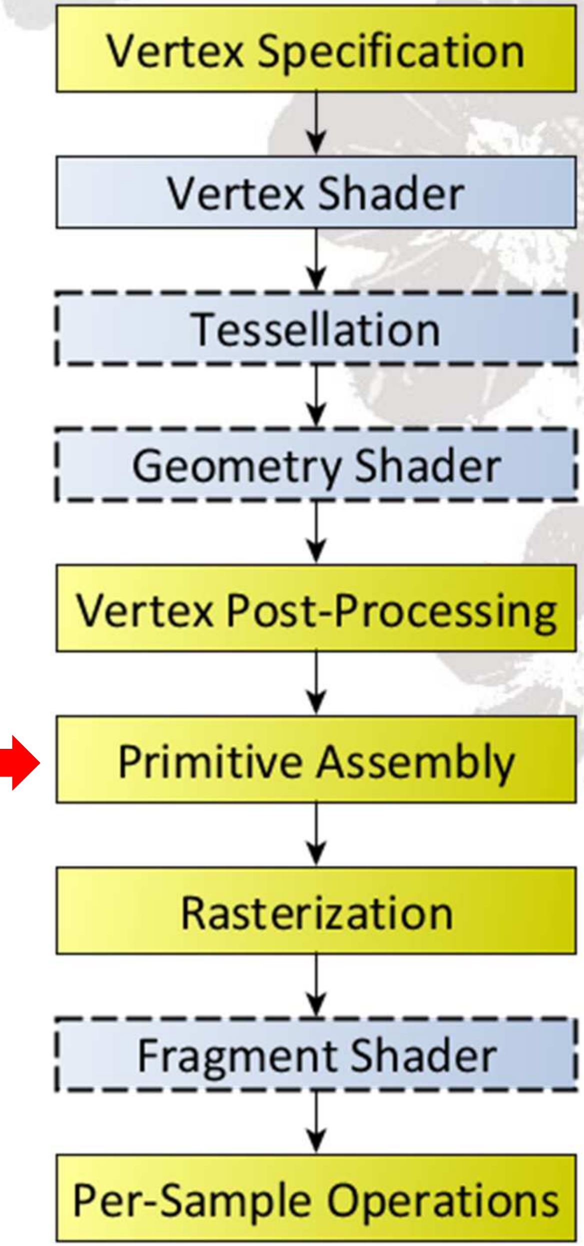
Fragment Shader

Per-Sample Operations

# *Primitive Assembly*

- **Collecting a run of vertex data output from the prior stages and composing it into a sequence of primitives**
  - **Back face culling**





Vertex Specification

Vertex Shader

Tessellation

Geometry Shader

Vertex Post-Processing

Primitive Assembly

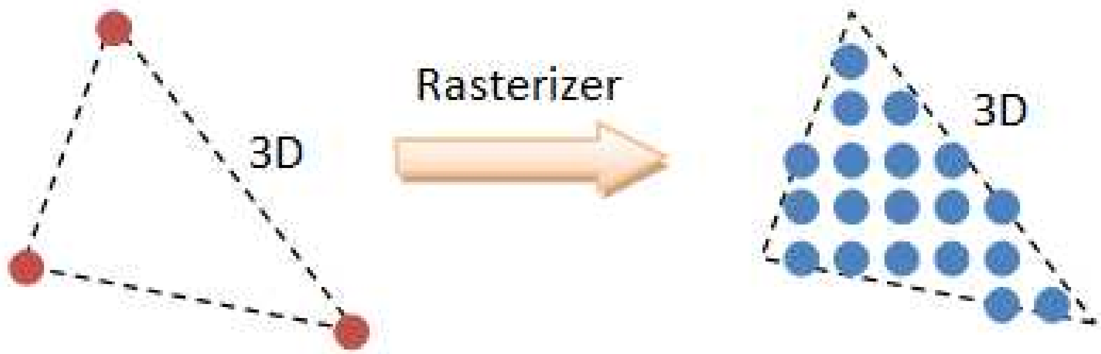
Rasterization

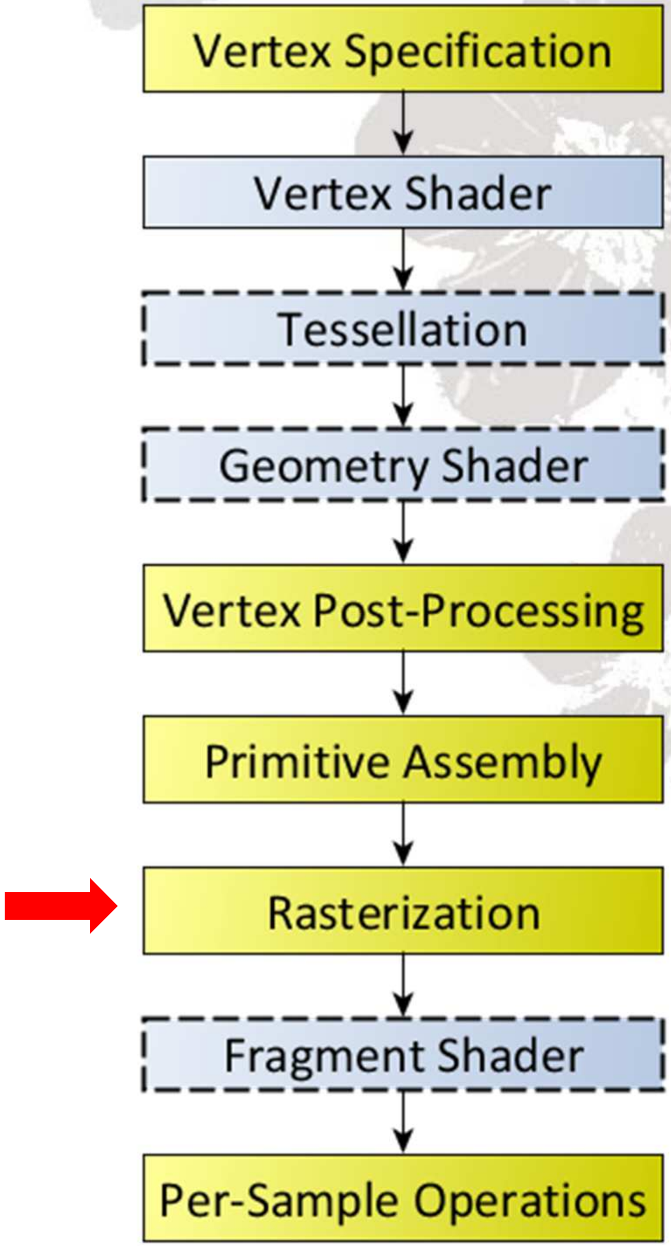
Fragment Shader

Per-Sample Operations

# *Rasterization*

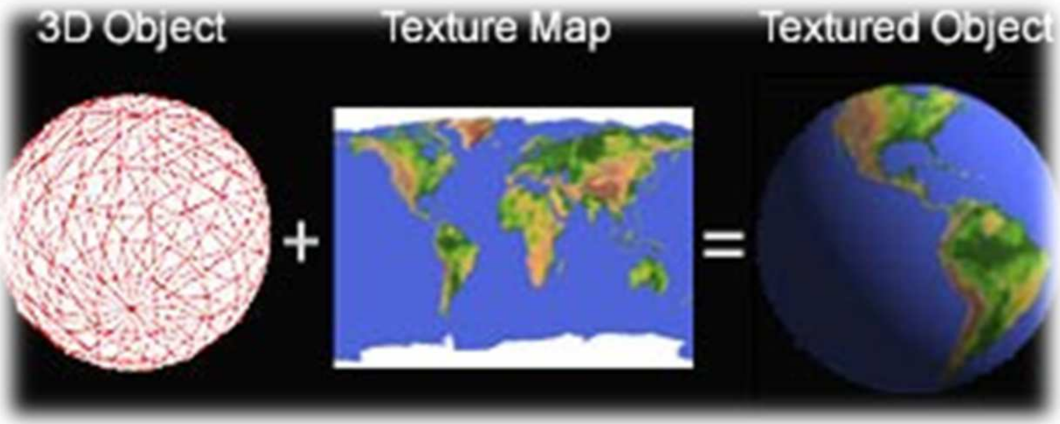
- Rasterizing a primitive into a sequence of fragments
  - Position
  - Color
  - Normal
  - Texture coordinates

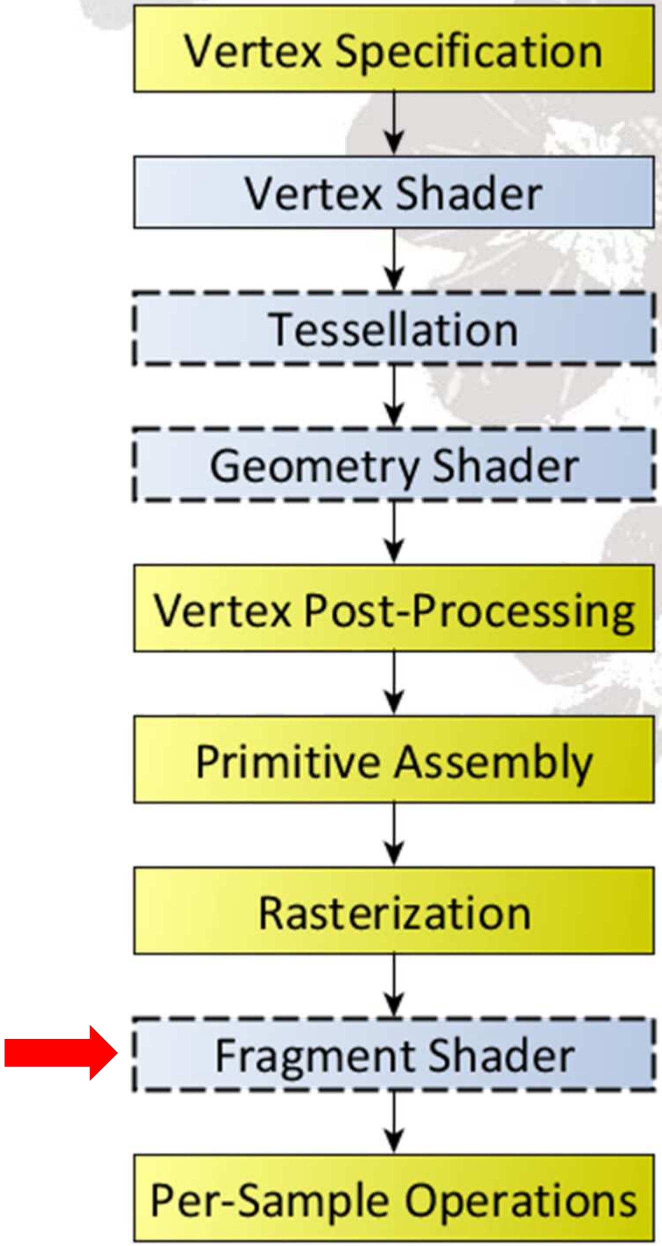




# *Fragment Processing*

- Process the color of each fragment
  - Texture mapping
  - Color combine (with texture)
  - Per-pixel lighting
  - Fog blending
  - Alpha test




Vertex Specification

Vertex Shader

Tessellation

Geometry Shader

Vertex Post-Processing

Primitive Assembly

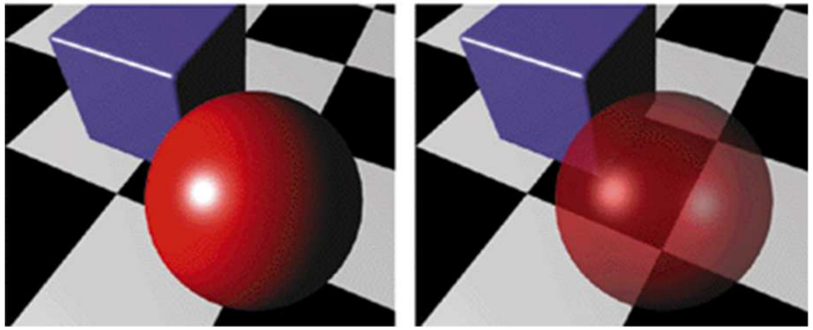
Rasterization

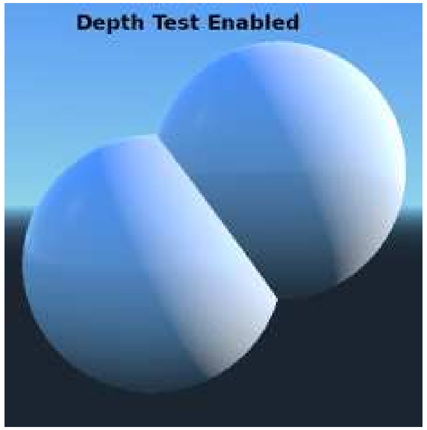
Fragment Shader

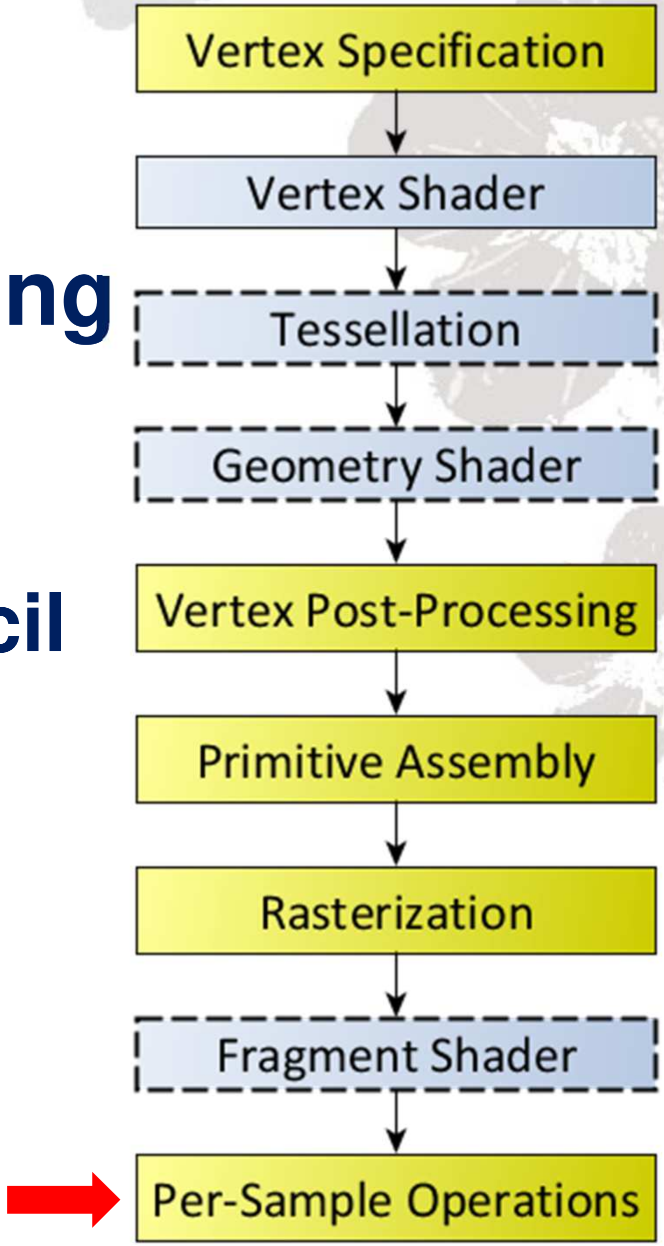
Per-Sample Operations

# *Per-Sample Operations*

- **Per-fragment OPs before updating the depth/stencil/color buffers**
    - **Fragment tests such as pixel ownership test, scissor test, stencil test, depth test**
    - **Color blending (with frame buffer)**
    - **Dithering**
    - **Color masking**







# Programmable Shaders

*Shader Architecture*

*Vertex Shader*

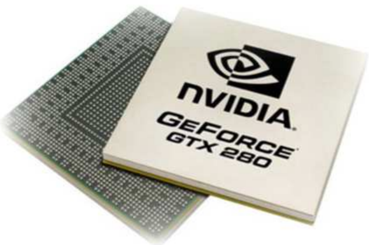
*Tessellation Shaders*

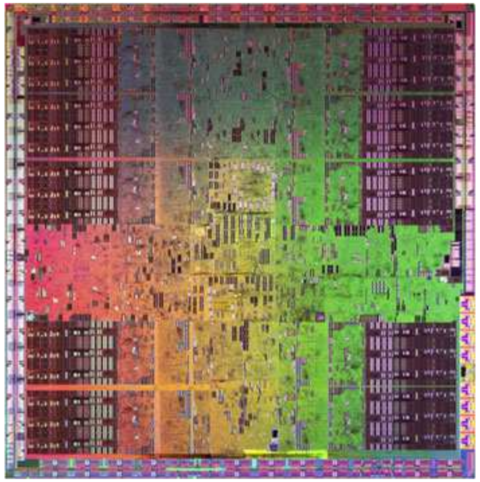
*Geometry Shader*

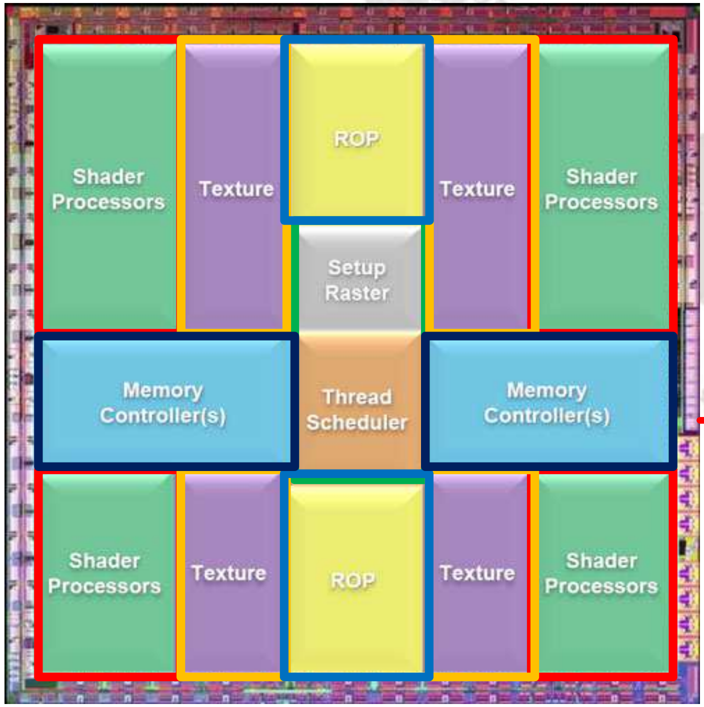
*Fragment Shader*

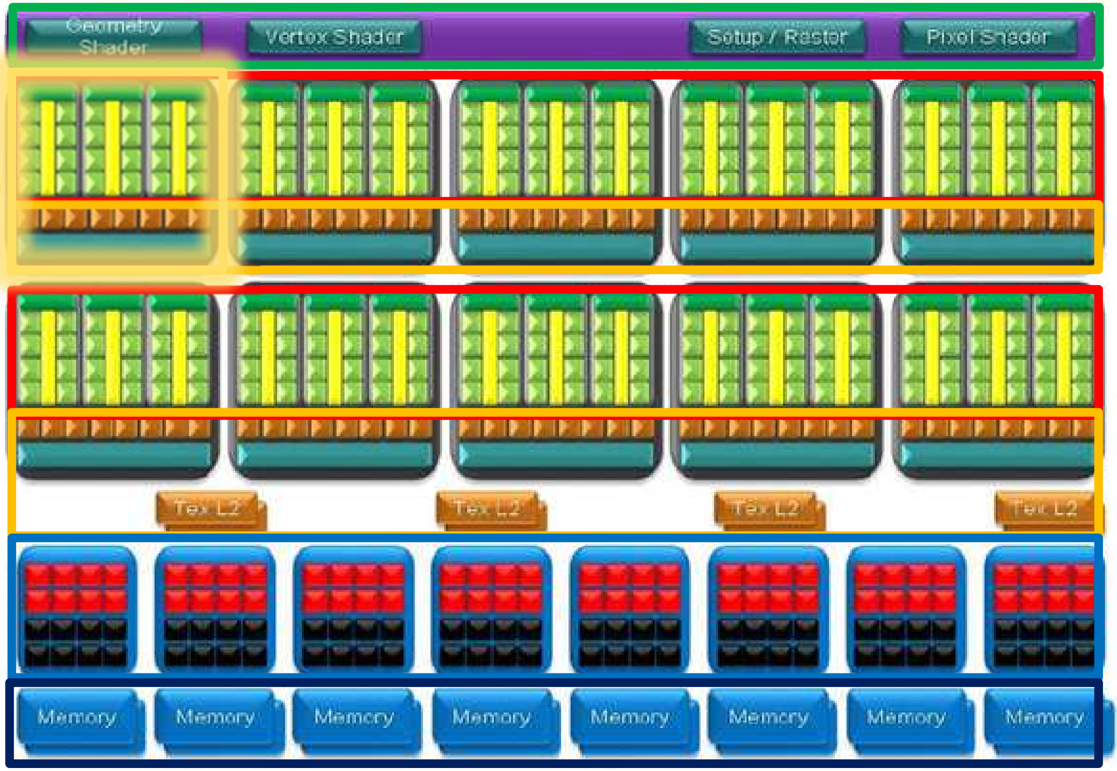
*Compute Shader*

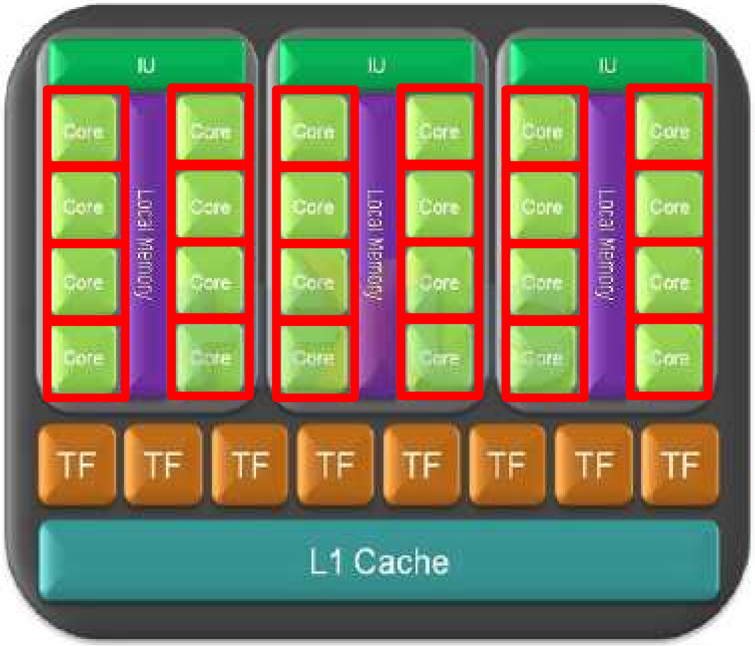
# Shader Architecture



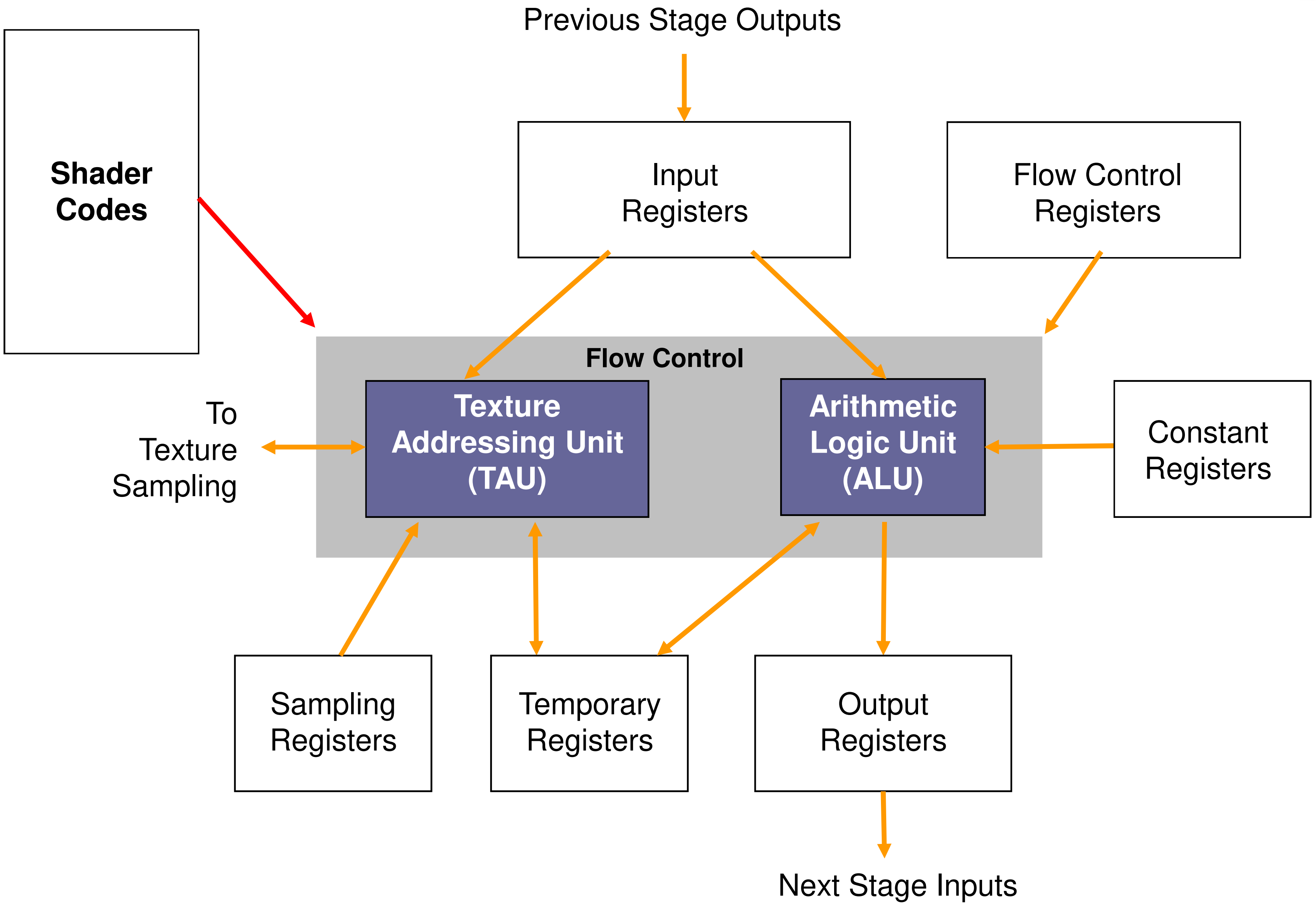






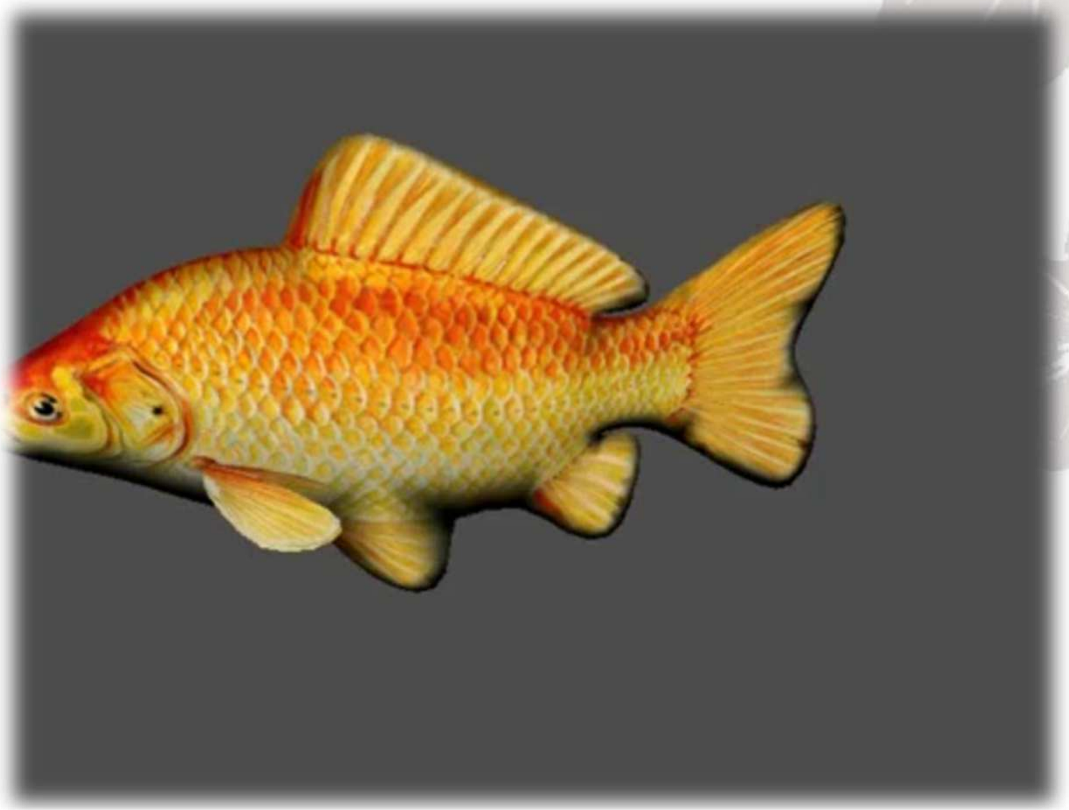


# Shader Architecture



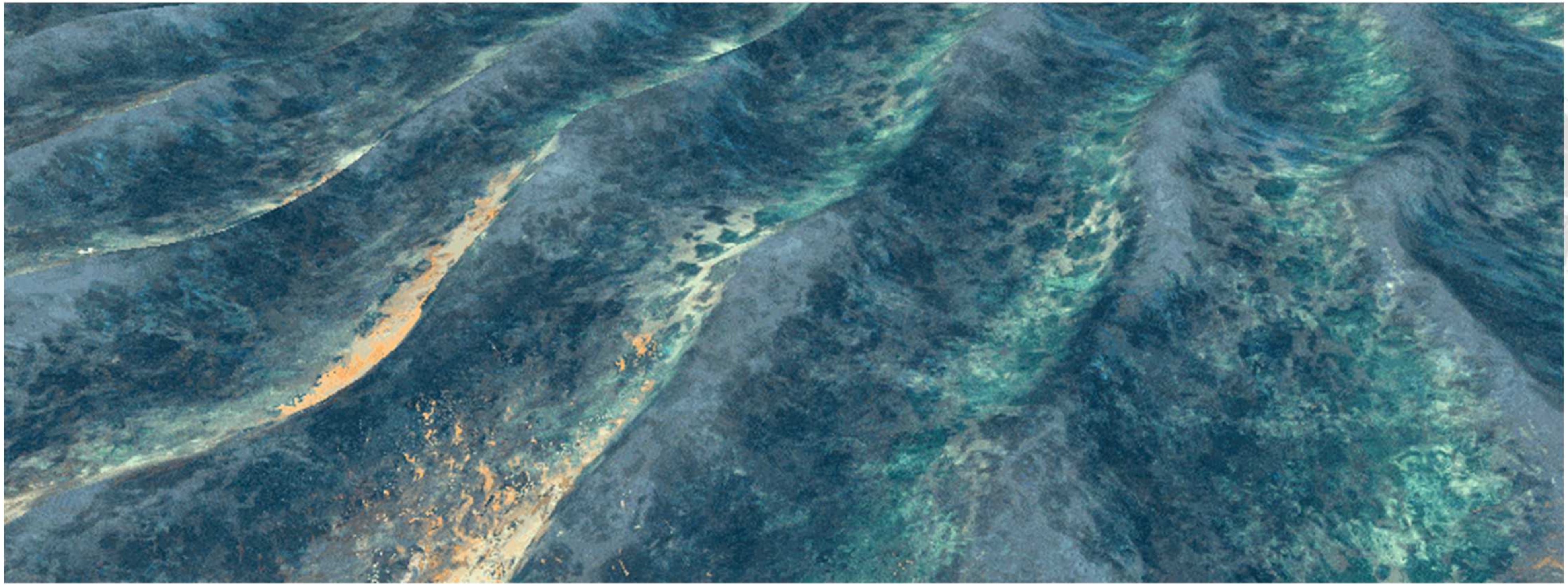
# *Vertex Shader*

- Processes vertices
  - Transformation
  - Lighting
  - Displacement
- Operate on a single input vertex and produce a single output vertex



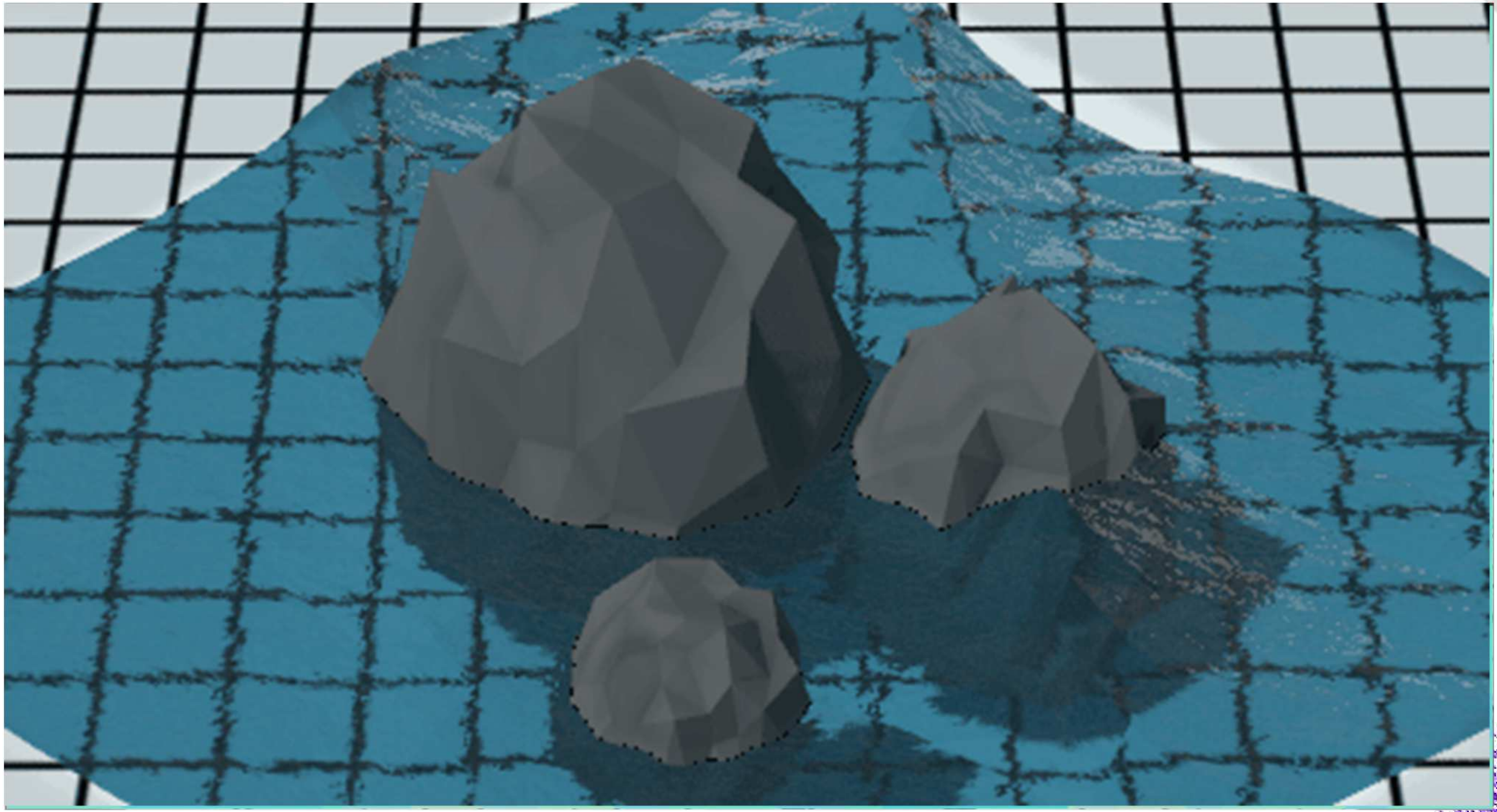
# Vertex Shader Applications

- **Wave Simulation**



# Vertex Shader Applications

- Wave Simulation



# *Vertex Shader Applications*

- **Hair/Fur**

© 2000 Columbia Pict

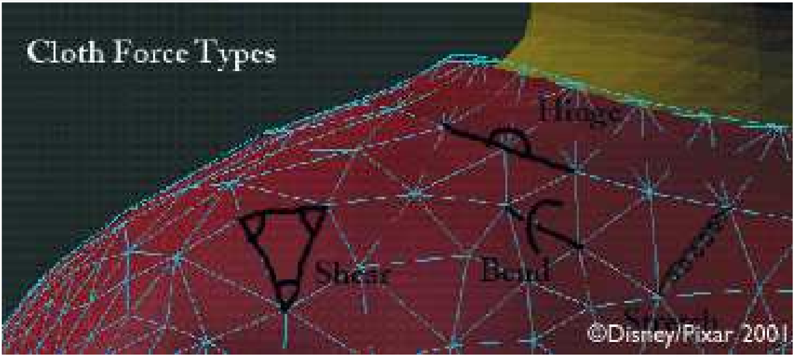
UND
ORANGES
PRODUCT
BRAND

interpolated hairs
©Disney/Pixar 2001

# *Vertex Shader Applications*

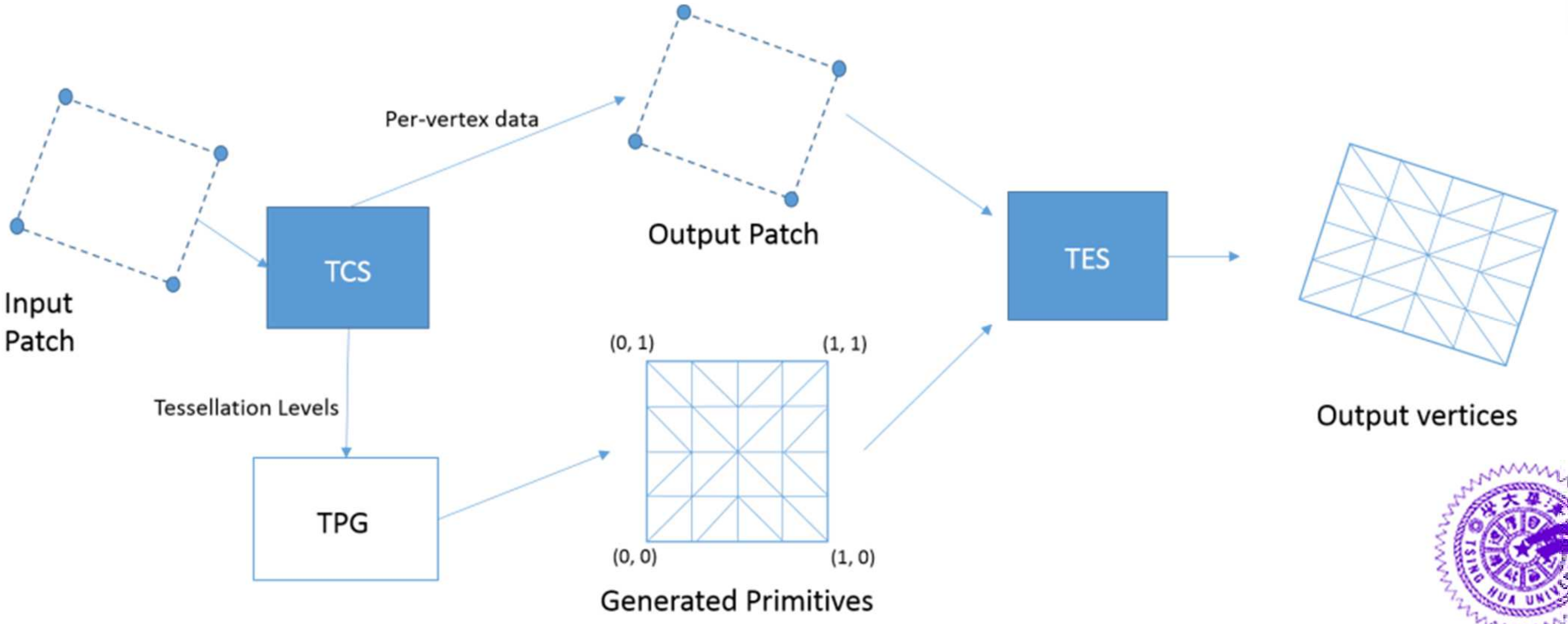
- **Surface Displacement**

©Disney/Pixar 2001



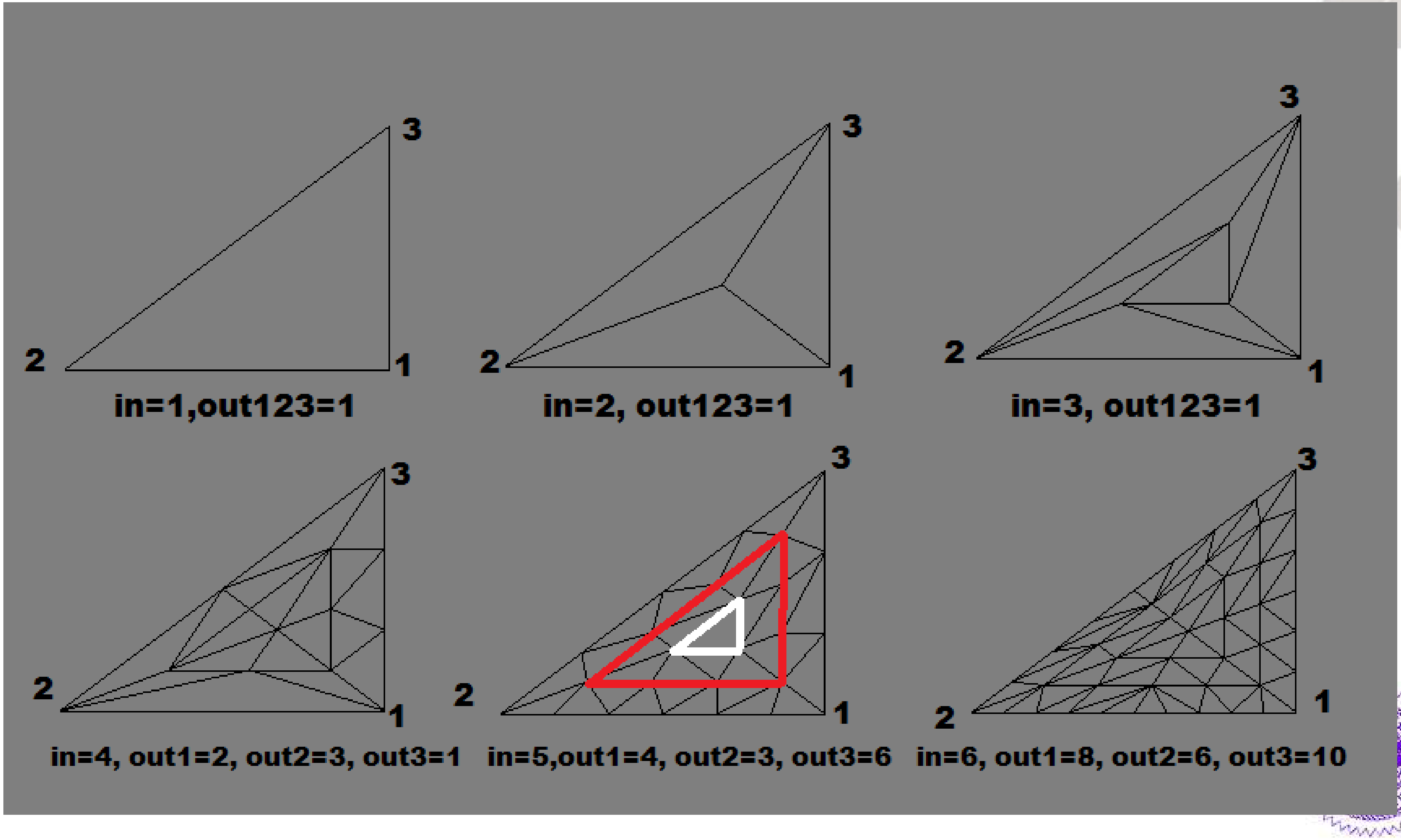
# *Tessellation Shaders*

- Subdivide Surface Patches
  - Tessellation Control Shader → tessellation levels
  - Tessellation Primitive Generator → primitives
  - Tessellation Evaluation Shader → new vertices



# *Tessellation Examples*

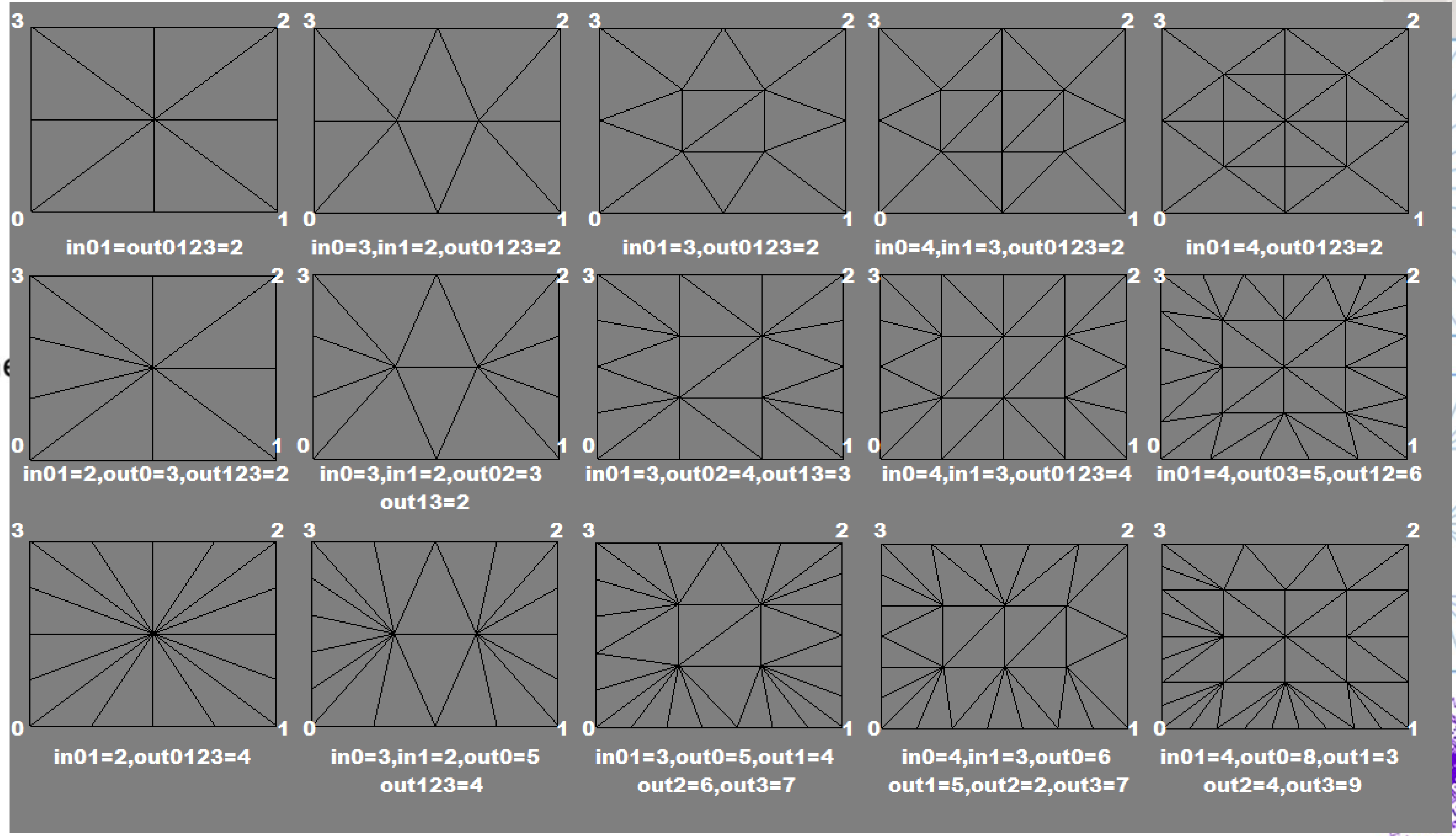
- Primitive type: Triangle



3
2
1
in=1,out123=1
3
2
1
in=2, out123=1
3
2
1
in=3, out123=1
3
2
1
in=4, out1=2, out2=3, out3=1
3
2
1
in=5,out1=4, out2=3, out3=6
3
2
1
in=6, out1=8, out2=6, out3=10

# Tessellation Examples

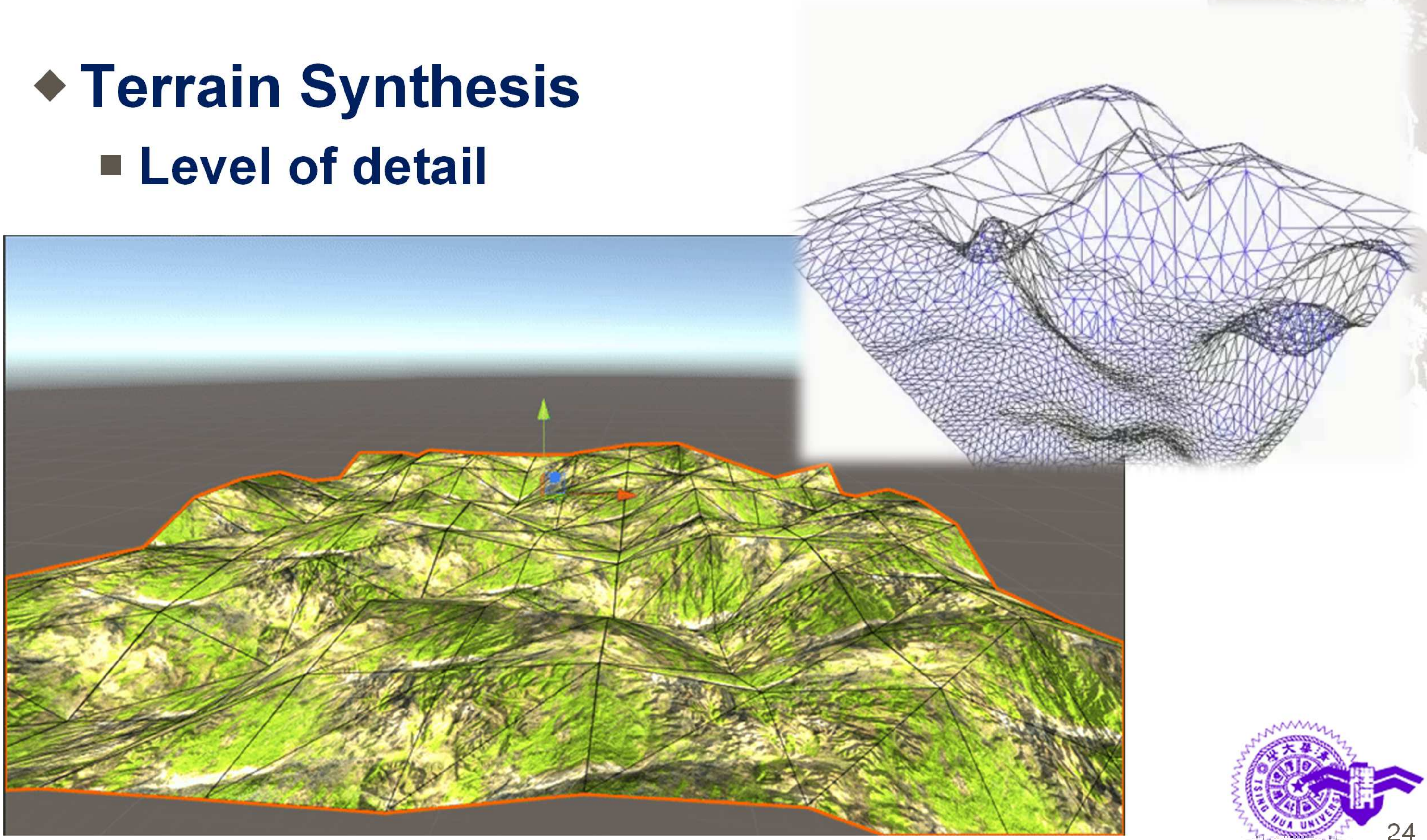
- **Primitive type: Quad**



Inne

# Tessellation Shader Applications

- Terrain Synthesis
  - Level of detail

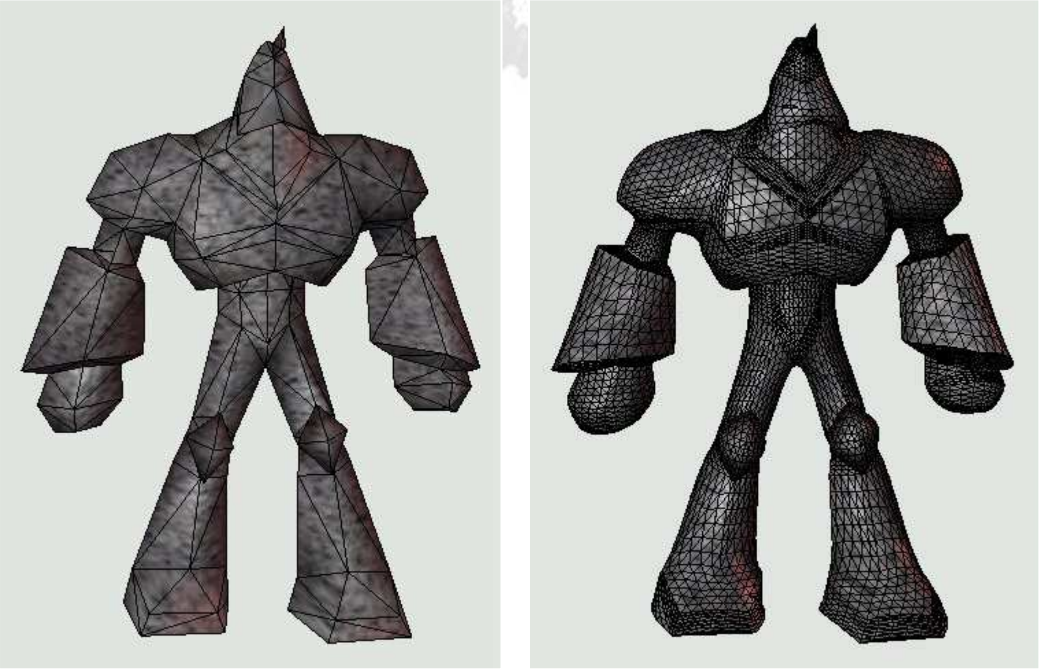


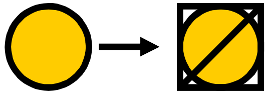
# Tessellation Shader Applications

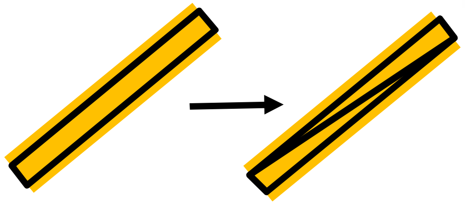
DX11 Tessellation off

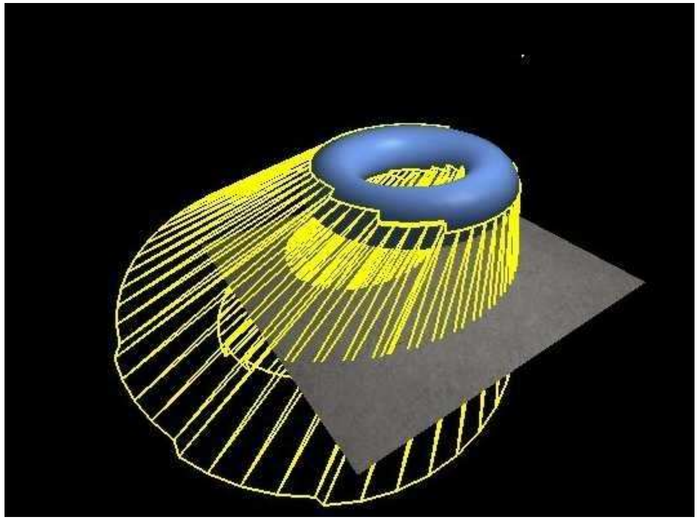
# Geometry Shader

- Process primitives
  - Point sprite tessellation
  - Wide line tessellation
  - Shadow volume generation
  - Surface subdivision
- Inputs one primitive. Outputs can be more than one primitives



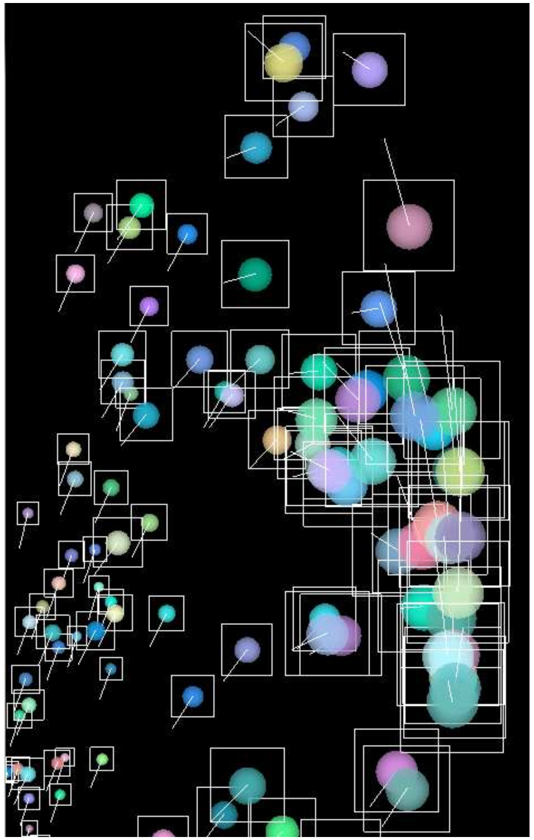






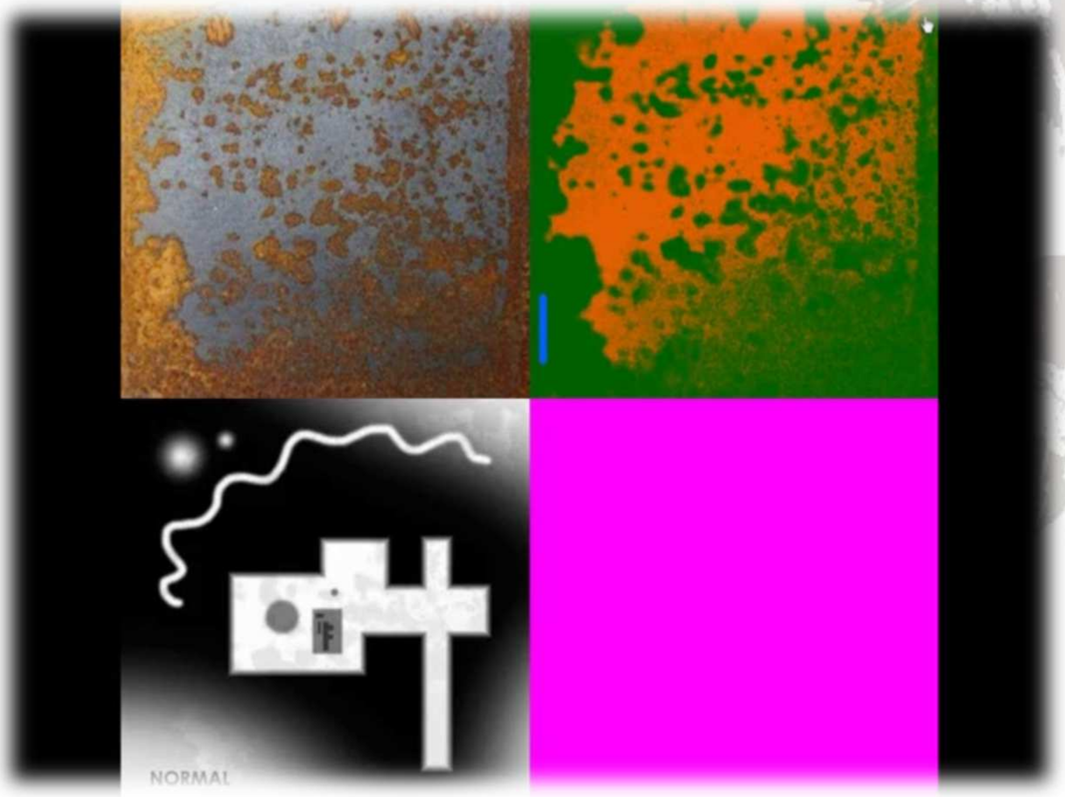
# Geometry Shader Applications

- Fireworks Particles
  - Point primitives to quad primitives



# *Pixel Shader*

- Process pixels
  - Texture mapping
  - Color combine
  - Per-pixel lighting
  - …
- Inputs one pixel. Outputs one pixels at same position, or no pixel.



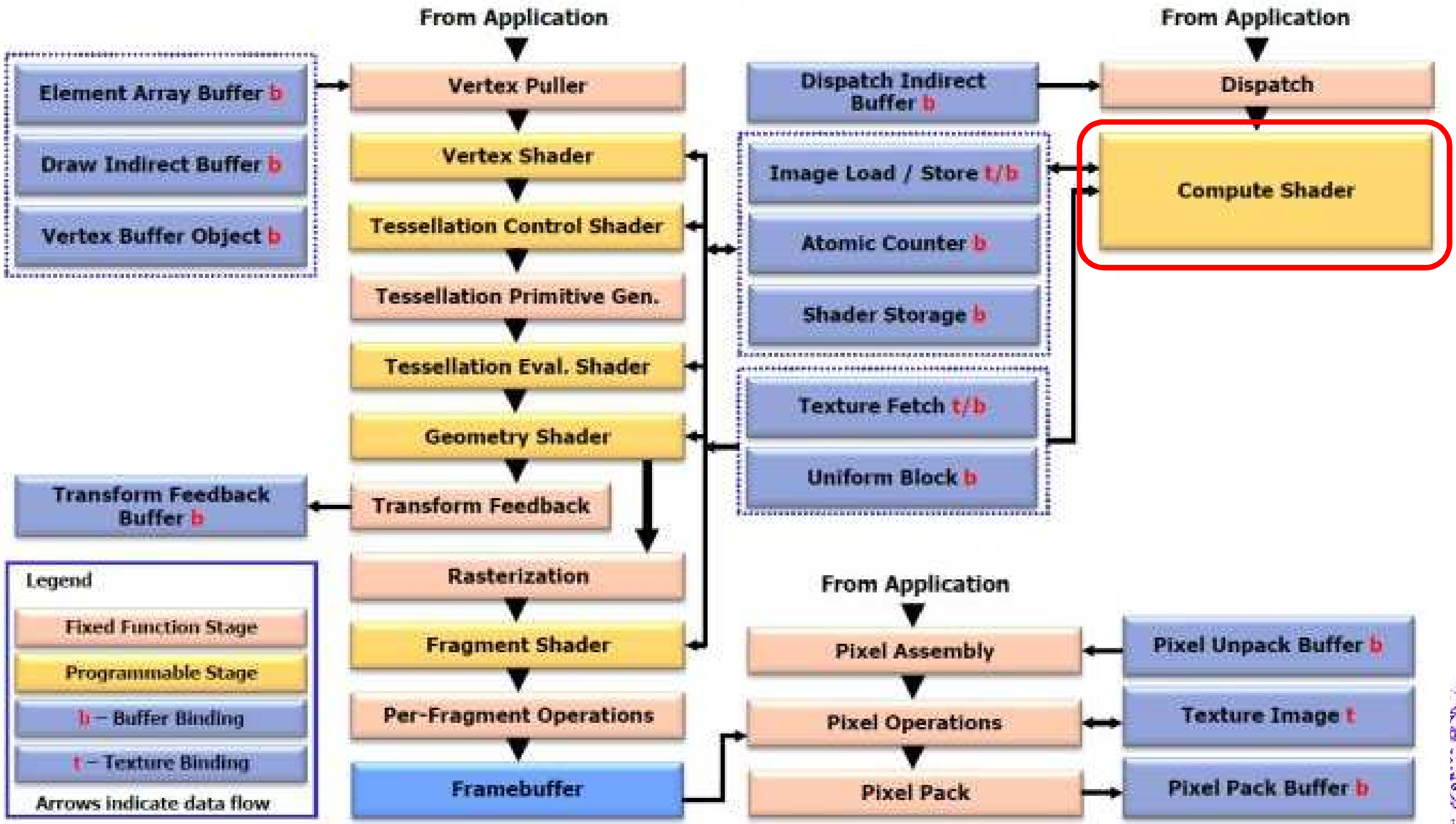
# *Pixel Shader Applications*

- **Multi-Texturing**

Normal Color
Side Color
Side Spec Bump
Spec
Translucence
Normal Shade
Side Shades
Final Mix
with Hair

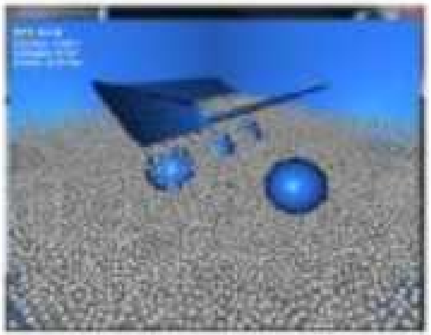
# Computer Shader

- Use the power of GPU for general purpose computing



# Computer Shader Applications

- Applications with complex and intensive computations



Physics

AI Simulation

Ray Tracing

Wave Simulation

Global Illumination

# *Unified Shader Model*

- Instruction set is consistent across all types of shaders
  - Vertex shader can also read texture too
    - Eg. Displayment map
  - But the capability is not all the same
    - Eg. Only geometry shader can generate new primitives

# *Unified Shader Model*

- Due to unified shader, the GPU can be used as general purpose computing device
  - GPGPU – General Purpose GPU
  - Other shading languages used for utilizing GPGPU
    - NVIDIA's CUDA (Compute Unified Device Architecture)
    - Khronos' OpenCL (Open Computing Language)
    - Microsoft CS (Compute Shader)

# *Shading Language*

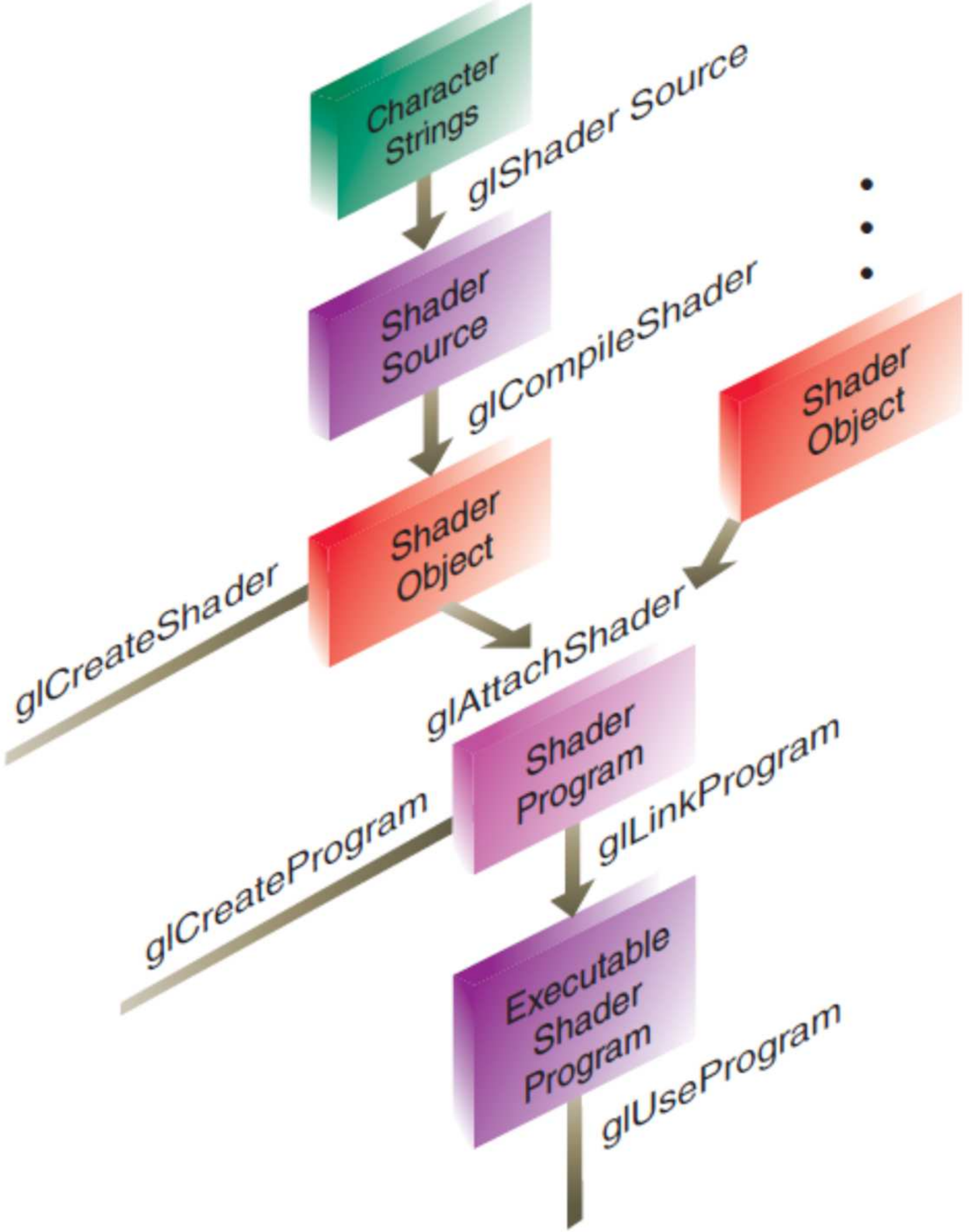
- Image synthesis can be divided into two basic concerns
  - Shape: Geometric Objects, Coordinates, Transformations, Hidden-Surface Methods…
  - Shading: Light, Surface, Material, Texture, …
- Control shading not only by adjusting parameters and options, but by *telling the shader what you want it to do directly* in a form of procedure

# *Shader Compiler*

- Shader compiler is used to compile the shader codes into hardware supported instructions
- Performance is highly depending on the compiler optimization

# *OpenGL Programmable Shaders*

# Shader Application Flow



# *Applying Shader in OpernGL*

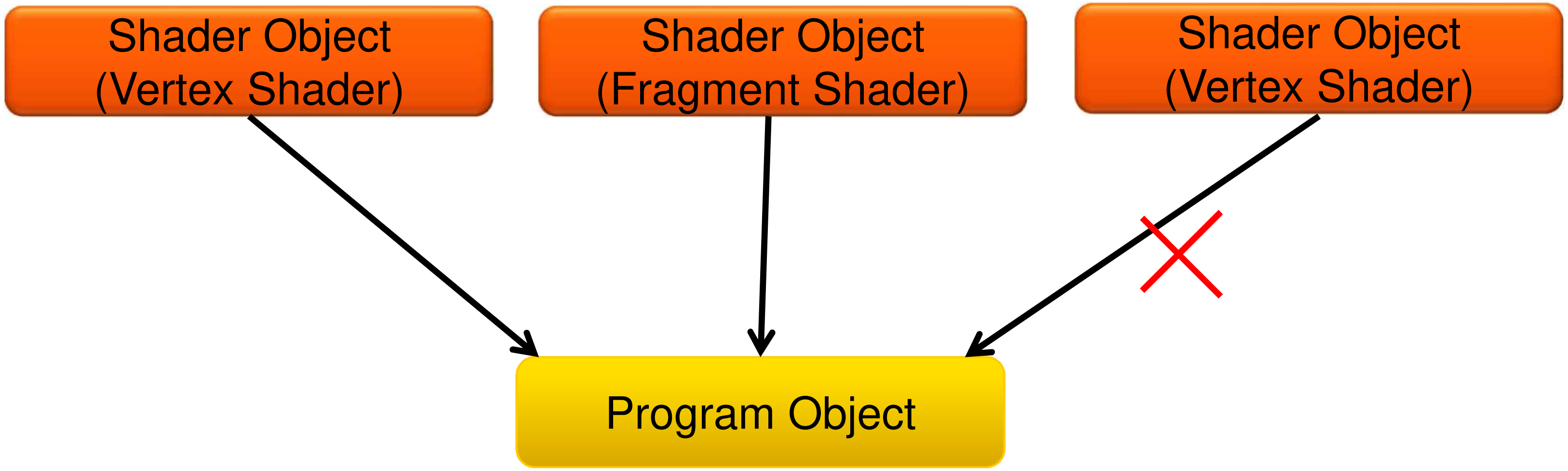
- Compile Phase
  - Create a shader object
  - Compile the shader source
  - Verify the status of compilation

# Applying Shader in OpernGL

- Link Phase
  - Create a shader program
  - Attach the shader object to shader program
  - Link the shader program
  - Verify the status of linking
  - Use the shader program

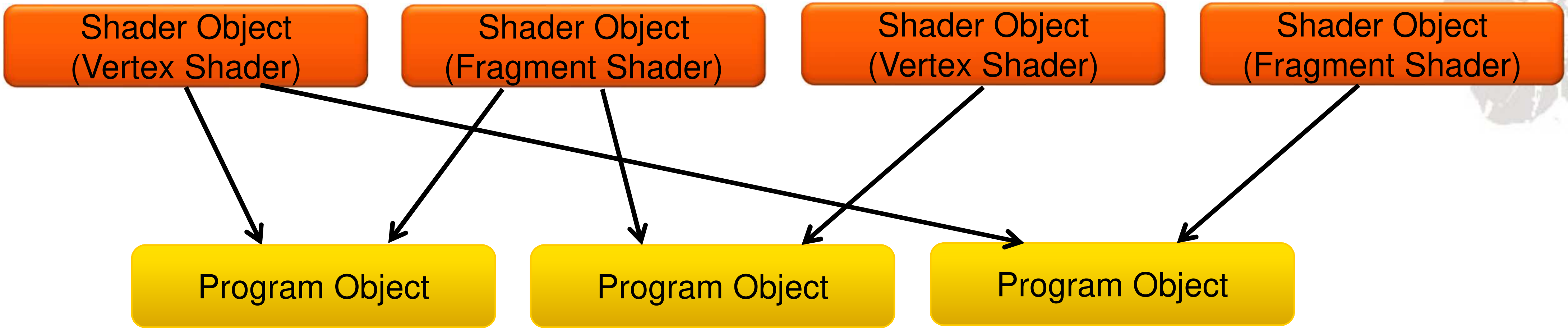
# Attach Shader

- Multiple shader objects of the same type may not be attached to a single program object


Shader Object (Vertex Shader)
Shader Object (Fragment Shader)
Shader Object (Vertex Shader)
Program Object

# Attach Shader

- A single shader object may be attached to more than one program object.


Shader Object (Vertex Shader)
Shader Object (Fragment Shader)
Shader Object (Vertex Shader)
Shader Object (Fragment Shader)
Program Object
Program Object
Program Object

# Using Shader

- GLuint **glUseProgram**(GLuint *program*)
  - The *program* is either for vertex or fragment processing depending on the type of shader created with glCreateShader()
  - In OpenGL version prior to v3.1, if *program* is zero, it will reverts to fixed-function operation. For OpenGL version higher than or equal to v3.1, the result is undefined if *program* is zero.

# *Loading Shader Binaries*

- Binary shaders are vender specific
- Application Scenario
  - Include all required GLSL shaders in your application
  - Compile the shaders during installation, or the first time your application runs
  - Save the program binary after successfully compiled and linked the shaders. (glGetProgramBinary())
  - Use the binary version on subsequent execution. (glProgramBinary())
- Note that this usage scenario only supported after OpenGL v4.1

# Q&A