Shader

Introduction

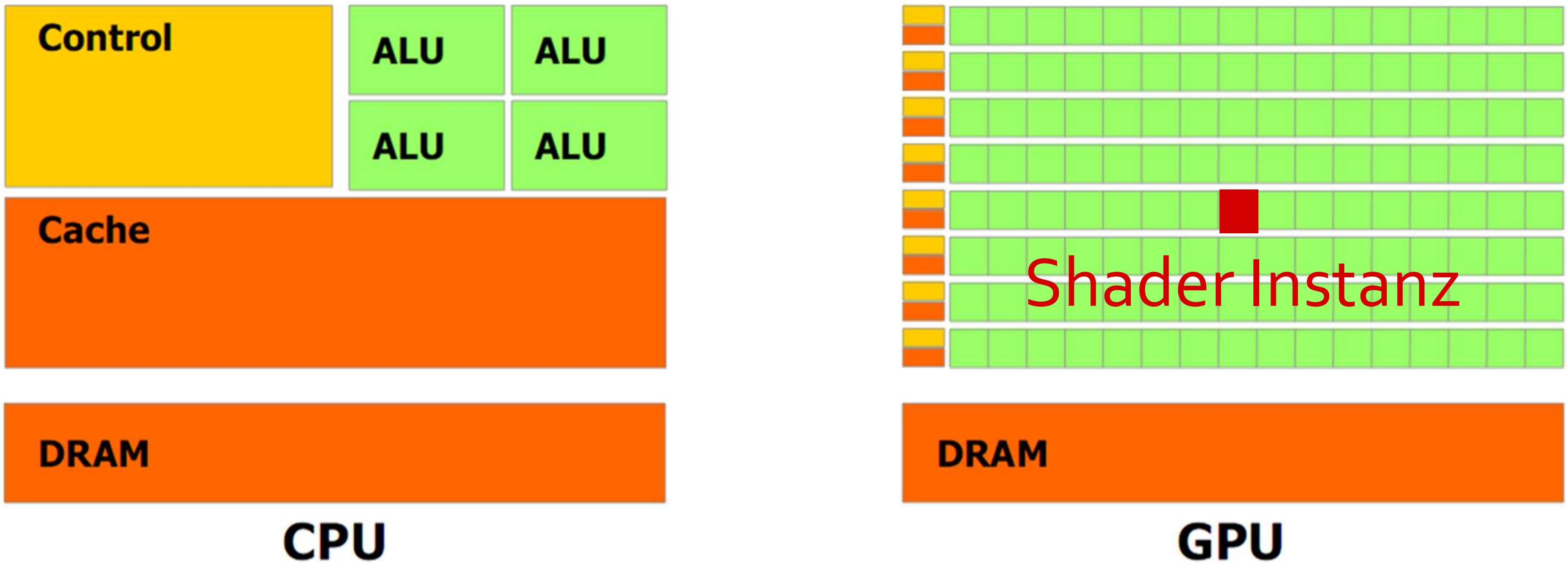
What are Shaders?

- Program
- Originally
  - „to shade“ (engl.): etwas schattieren (ge)
  - RenderMan, Pixar
  - Implements rendering effect
- Today
  - Executed on **GPU**
  - Many instances in **parallel** (>> 1000)
  - Processes stream of data **„stream programming**“

GPU = Graphics Processing Unit

Hardware comparison

- CPU uses more transistors for control and buffering
- GPU uses more transistors for processing



What are Shaders?

- Wikipedia:
  „A shader in the field of computer graphics is a set of software instructions, which is used primarily to calculate rendering effects on graphics hardware with a high degree of flexibility."

Why not use the CPU?

- CPUs are „general purpose“
  - Execute different tasks
- Programmed with general purpose languages
  - C++, Java, ...
- Not particulary good at graphics tasks
  - Repeated similar largly independent tasks

Why not use the CPU?

- GPU (Graphics Processing Unit)
  - Processes tens of millions of vertices per second
  - Rasterizes billions of pixels per second
- Cannot execute arbitrary, general purpose programs like the CPU
- Need for (specialized) language for programming the GPU
  → a shader language!

Assembly or high-level….

Assembly

```
…
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

or

GLSL

```
…
vec4 cPlastic =
    Ca +
    Cd * dot(Nf, normalize(L)) +
    Cs * pow(max(0,
    dot(Nf,normalize(H))),
    phongExp);
…
```

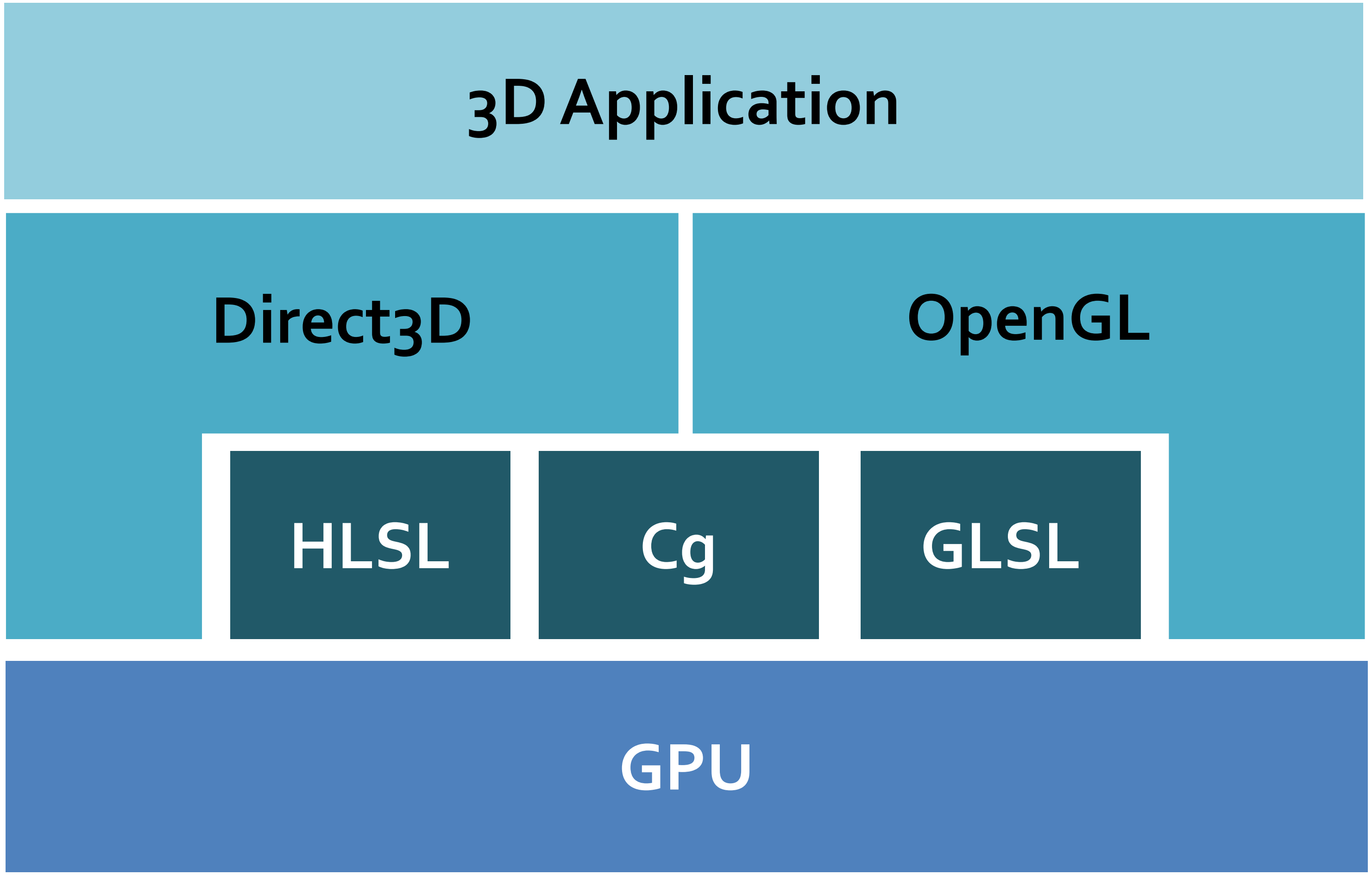
High-level Shader Languages

- **HLSL** (Direct3D)
- **Cg** (Direct3D, OpenGL)
- **GLSL** (OpenGL)

very similar to C++/C#

- We focus on GLSL
- Concepts easy to transfer to other shading languages

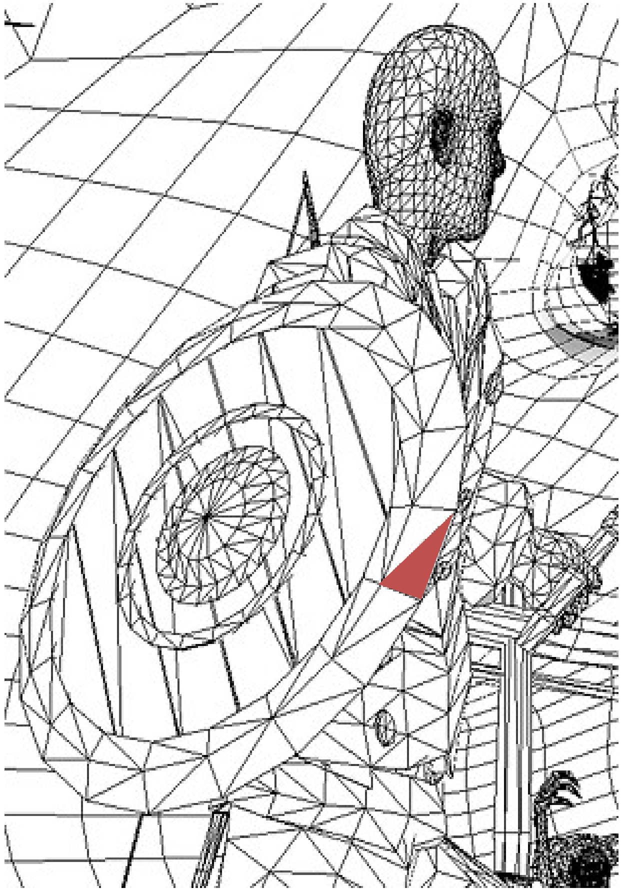
Application & API Layers



Shader

Principles

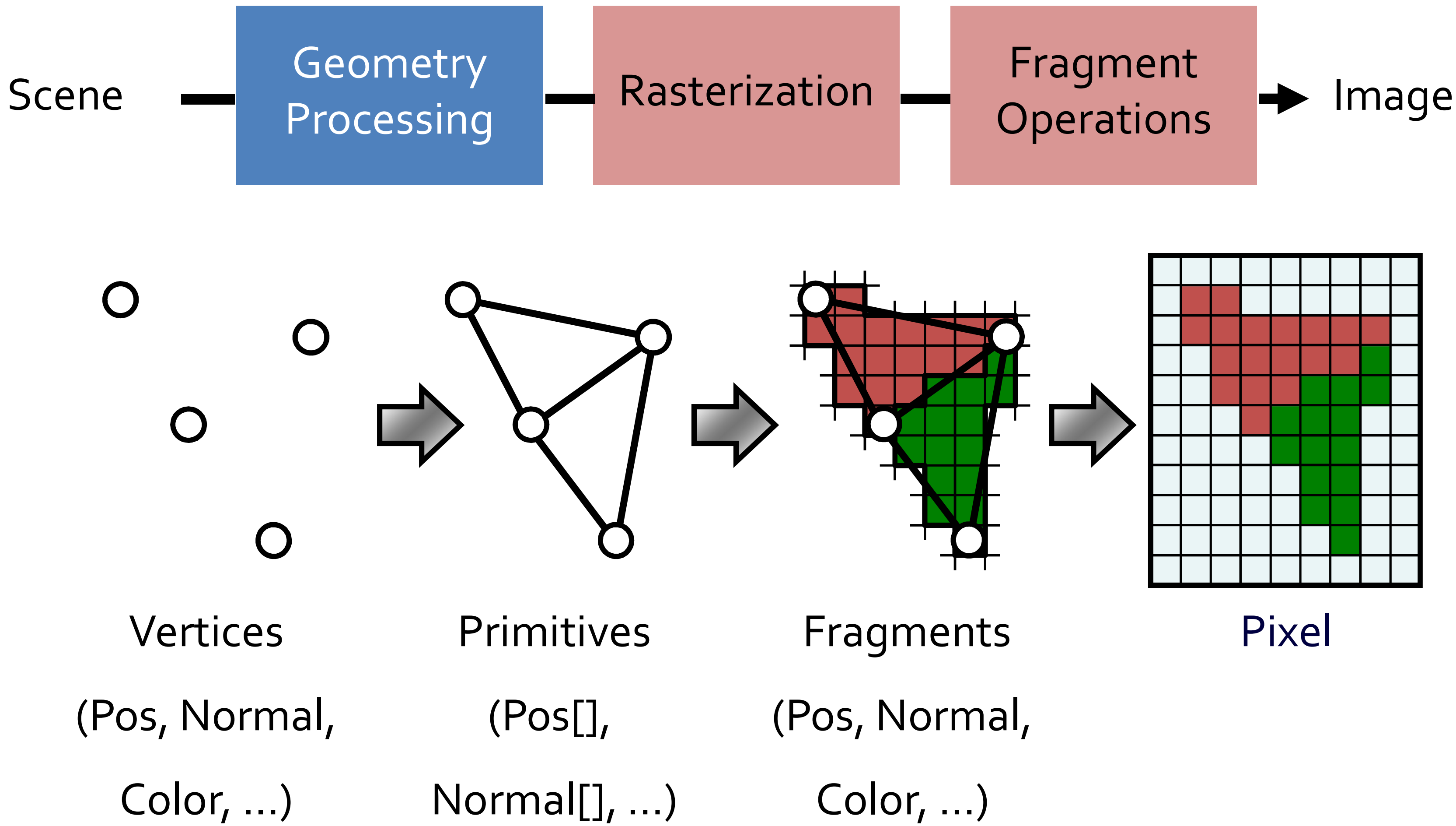
Rendering



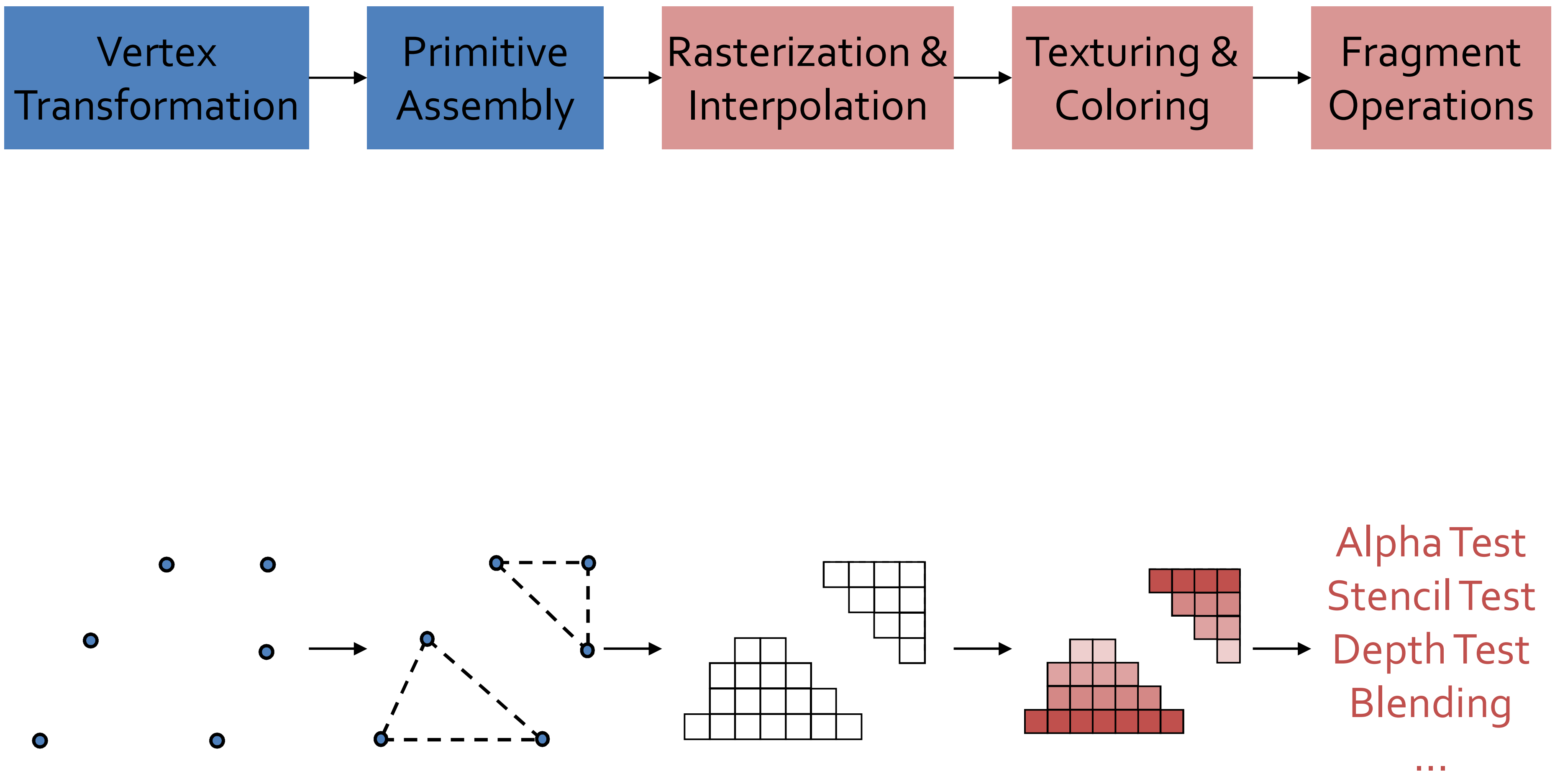
Rendering by Graphics Hardware

Scene → Image

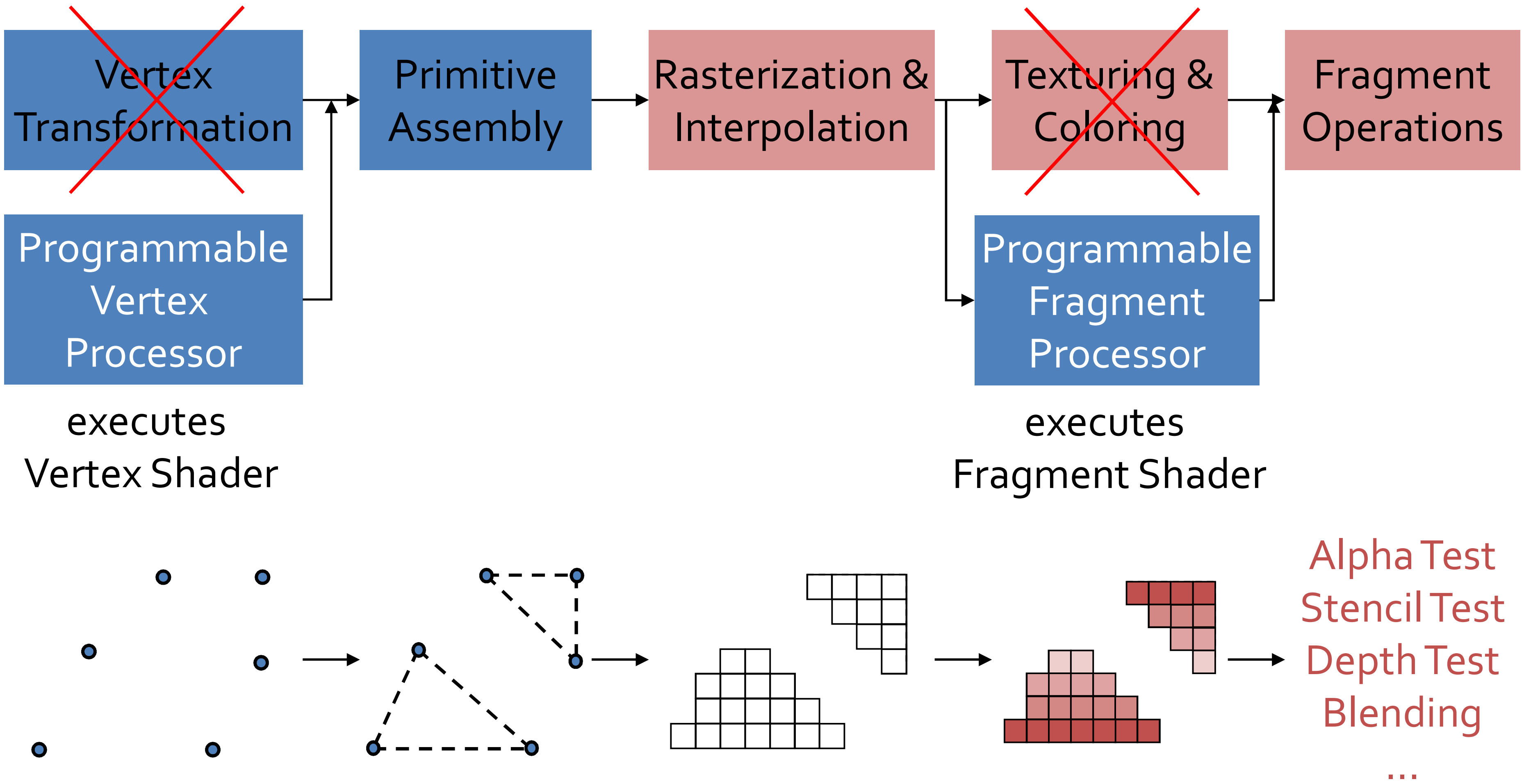
Rendering by Graphics Hardware



The Hardware 3D Pipeline



The Programmable Hardware Pipeline

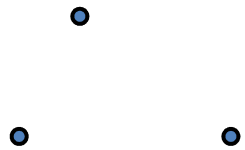


The Hardware 3D Pipeline

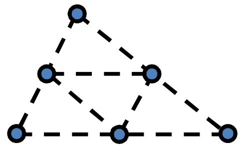
- Vertex and Fragment Shaders replace parts of the fixed function pipeline
- Implement replaced functionality yourself
  - Transformation & Lighting
  - Texturing & Coloring
- Gain freedom and flexibility how to implement

Types of Shader

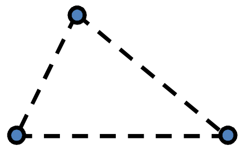
- Vertex Shader
  - Per vertex



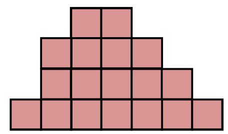
- Tesselation Shader
  - Per patch
  - Introduced with DirectX 11 / OpenGL 4



- Geometry Shader
  - Per primitive (i.e. Triangle)
  - Introduced with DirectX 10 / OpenGL 3.2

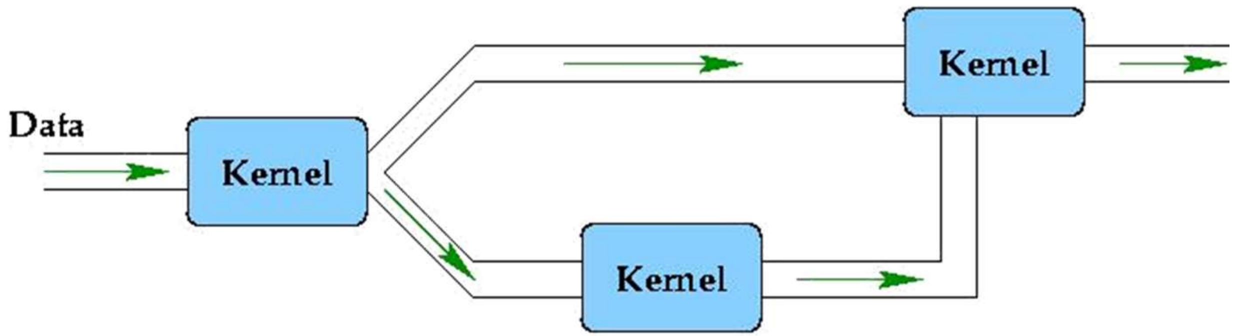


- Fragment Shader (aka Pixel Shader)
  - Per fragment

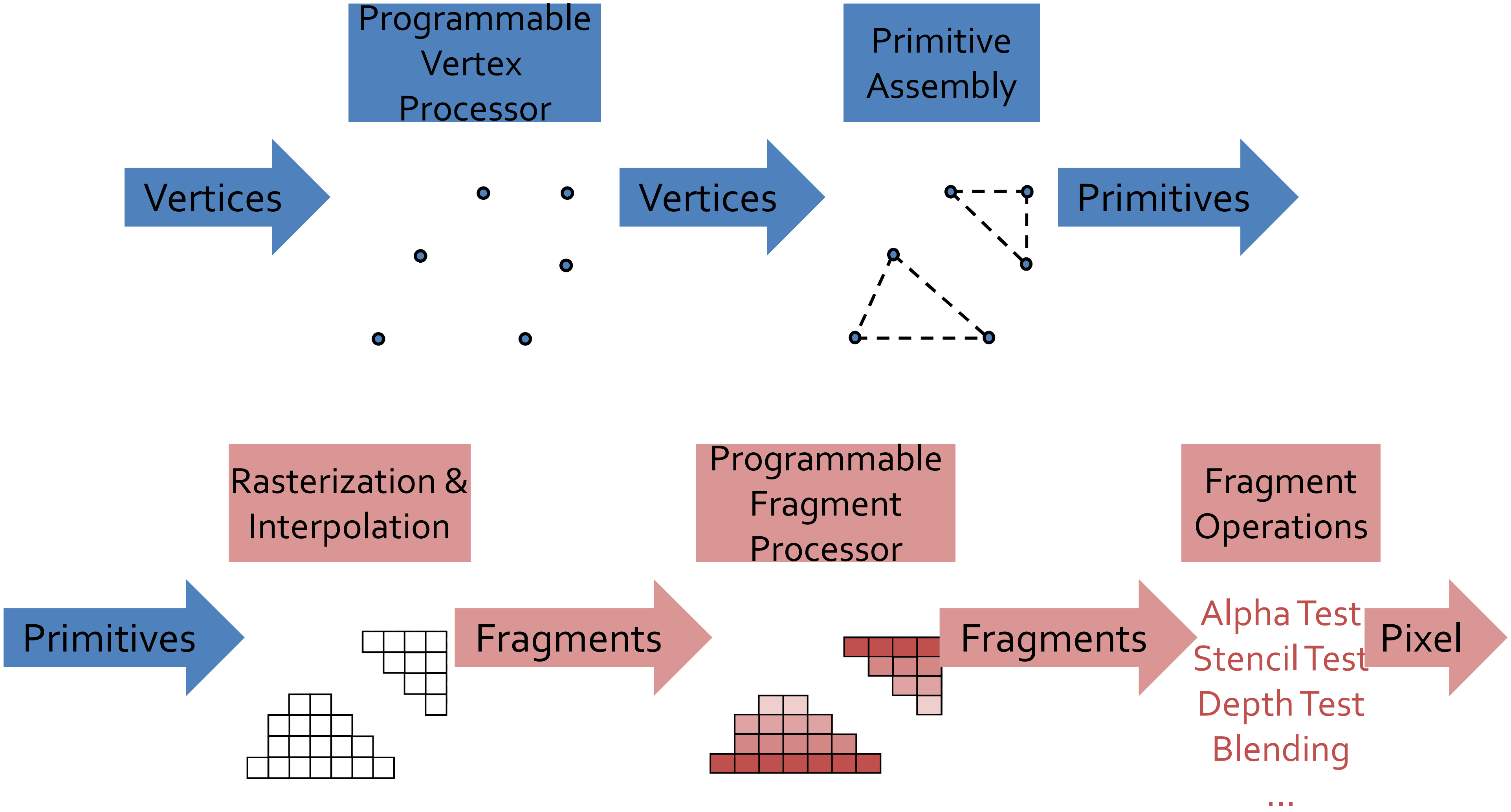


Stream programming

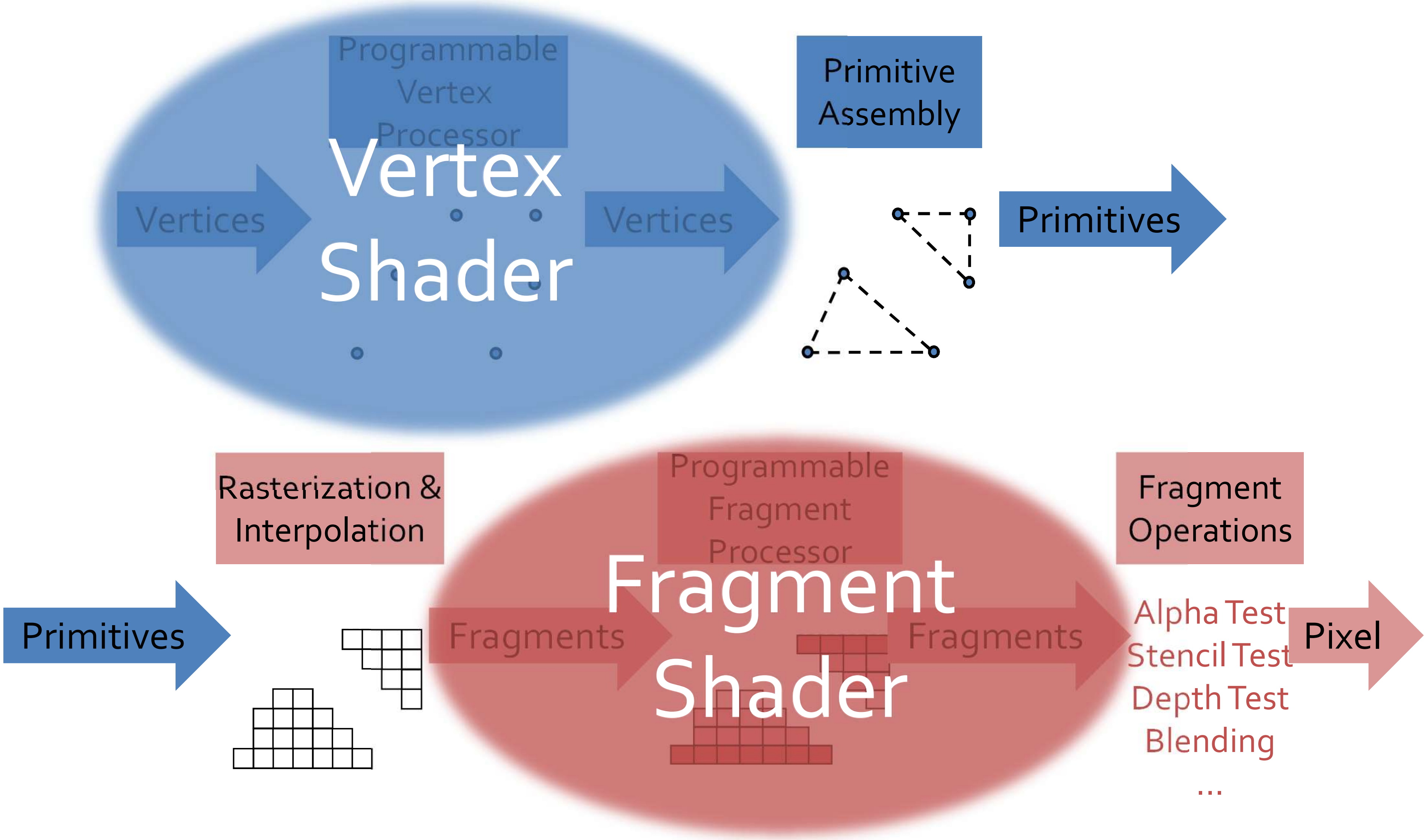
- **Stream:** sequence of data
  - Skalars, vektors, colors, …
  - Example: { (pos, color); (pos, color); … }
- Shader (“kernel”) program
  - Each processes one element of input stream
    - **Read only**
  - Creates output stream
    - **Write only**



The Hardware 3D Pipeline as a stream



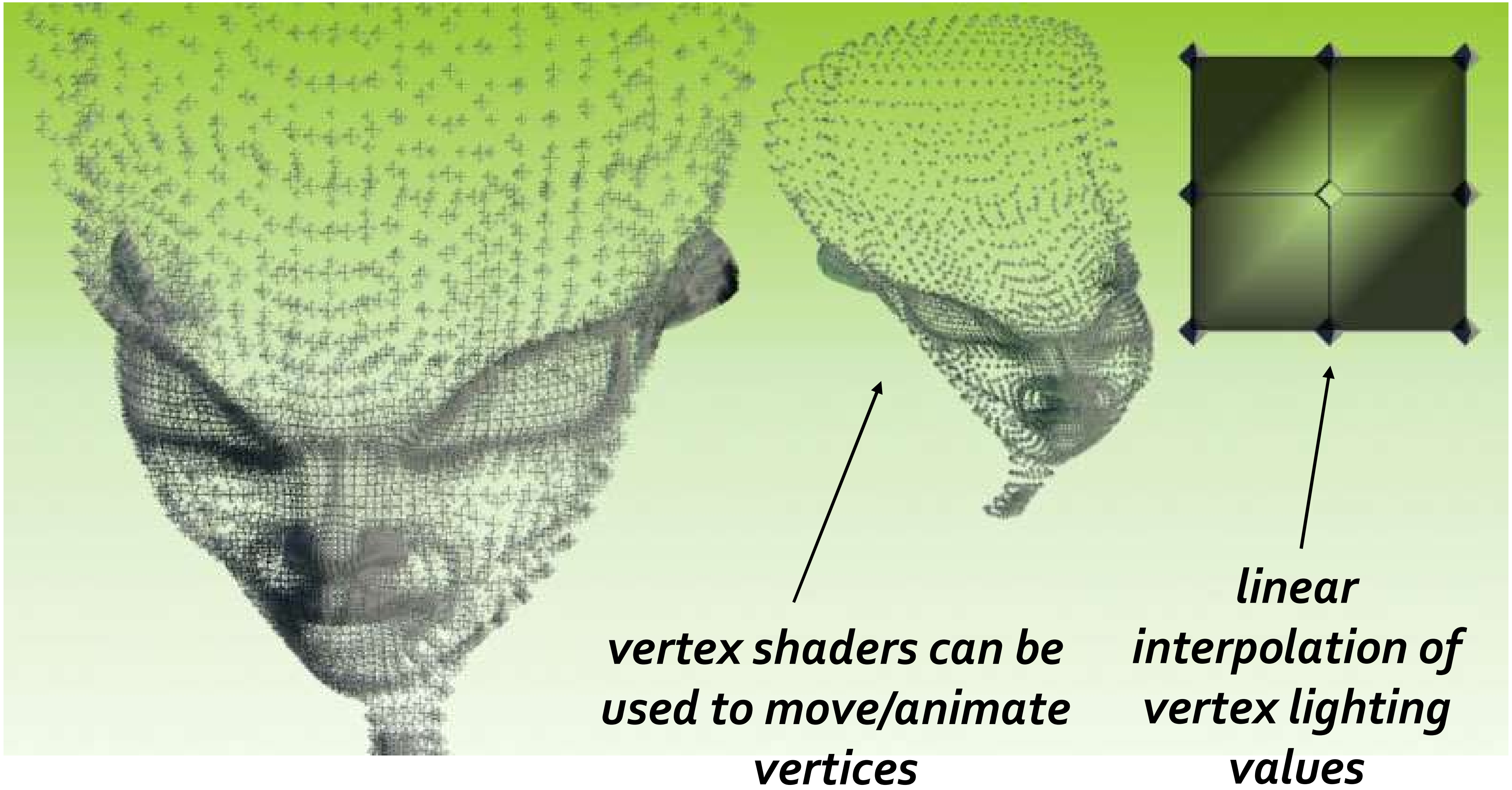
The Hardware 3D Pipeline as a stream



Shader

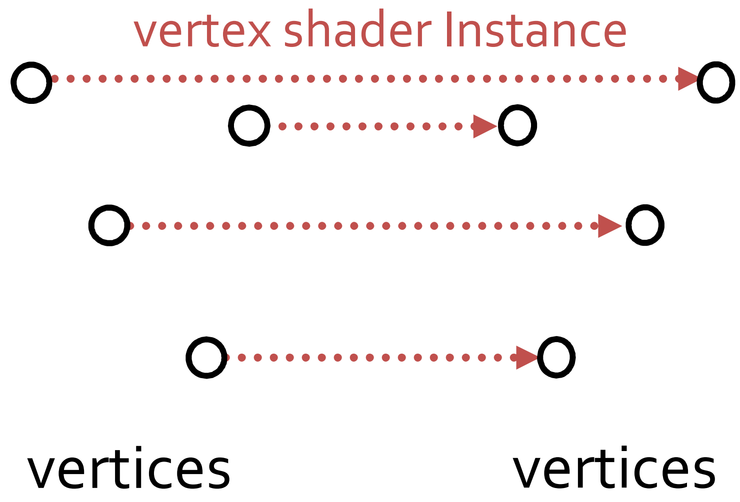
Vertex and Fragment Shader

Vertex shader



Vertex shader

- Input stream: vertices
- Output stream: vertices

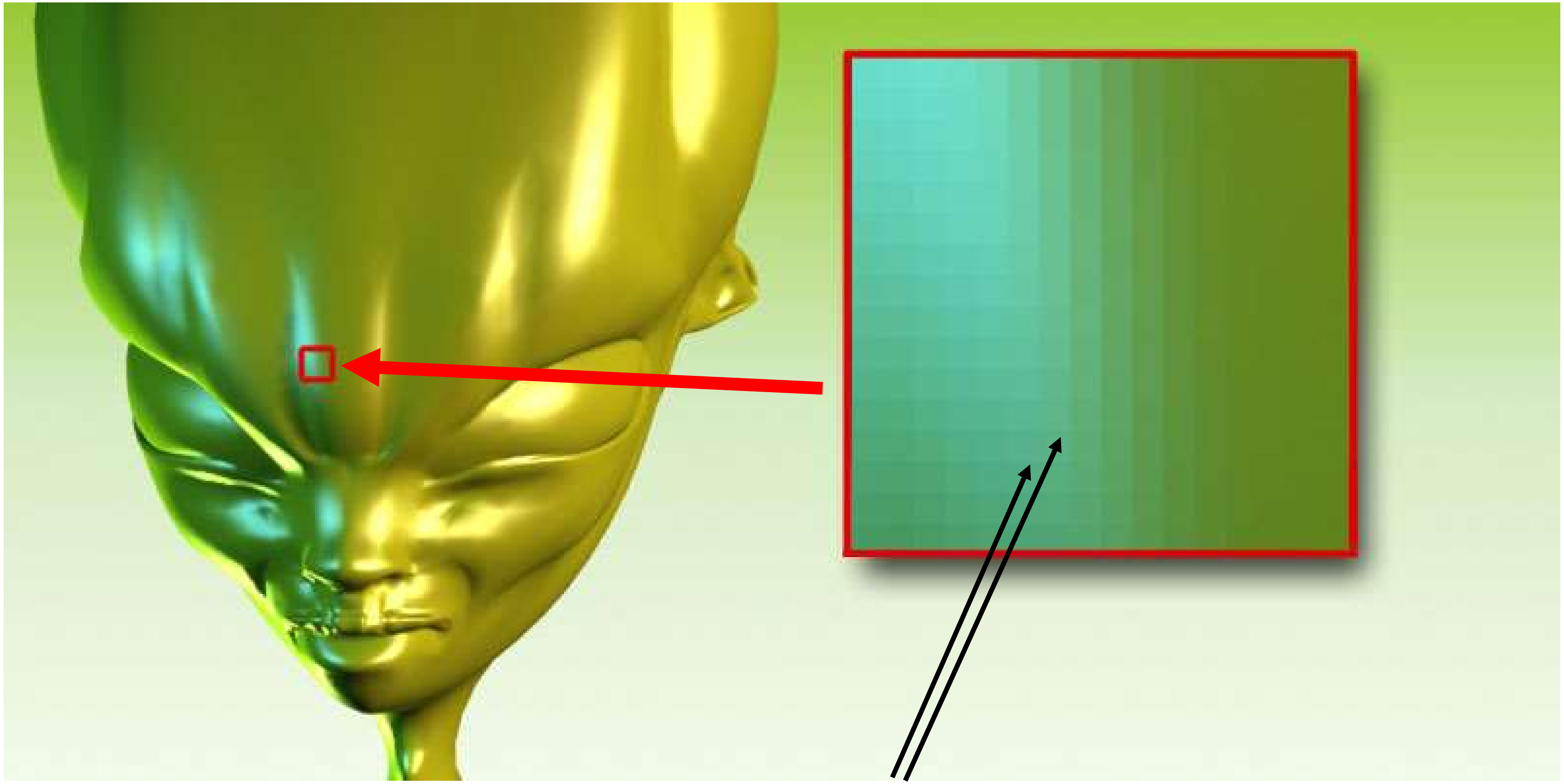


- One instance processes one vertex
- No knowledge of neighbouring vertices

Typical applications

- Transformation of vertices
  - Object space → clipping space
  - Animation
  - Particle systems
  - Displacement Mapping
- Lighting
  - Per vertex lighting
  - Cartoon shader
  - ...

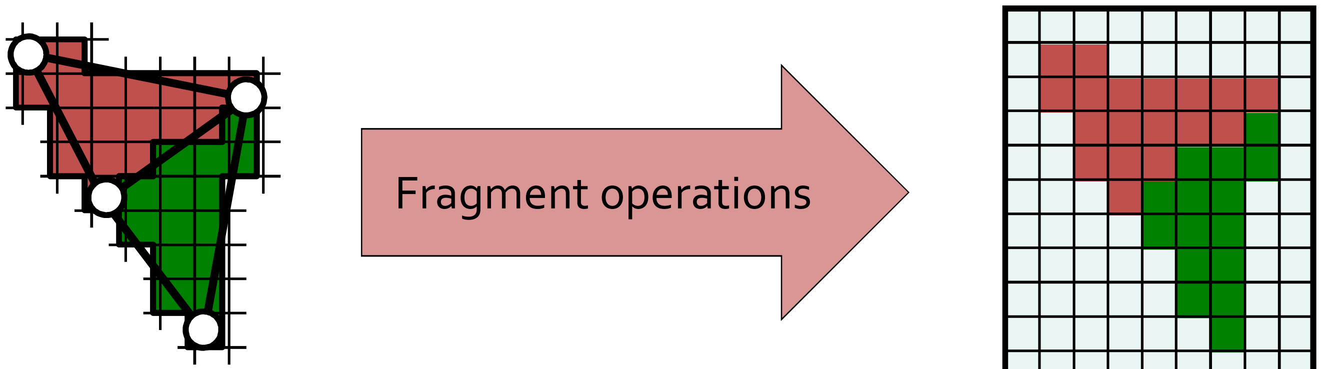
Fragment shader



Each fragment is calculated individually

Fragment = „potential pixel"

- For each pixel that a primitive covers a fragment is created
- If a fragment passes the various rasterization tests (Stencil Test, Depth Test …) it updates a pixel in the frame buffer.

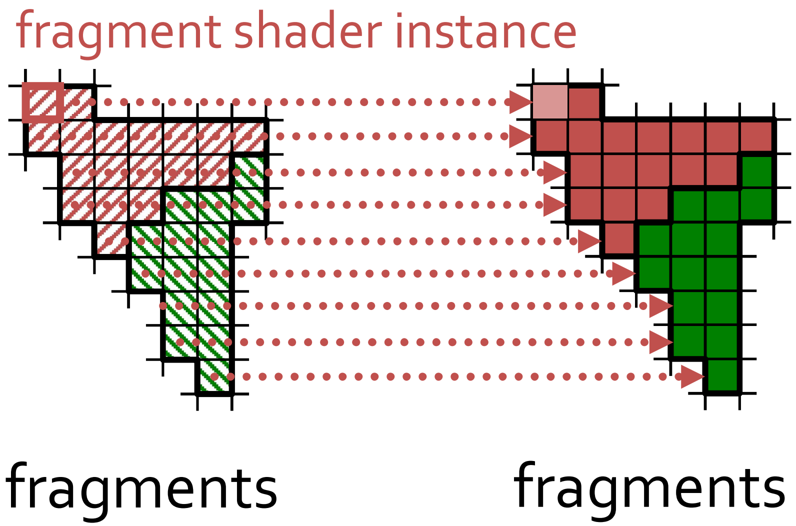


Fragments (Pos, Normal, Color, …)

Pixel (Color)

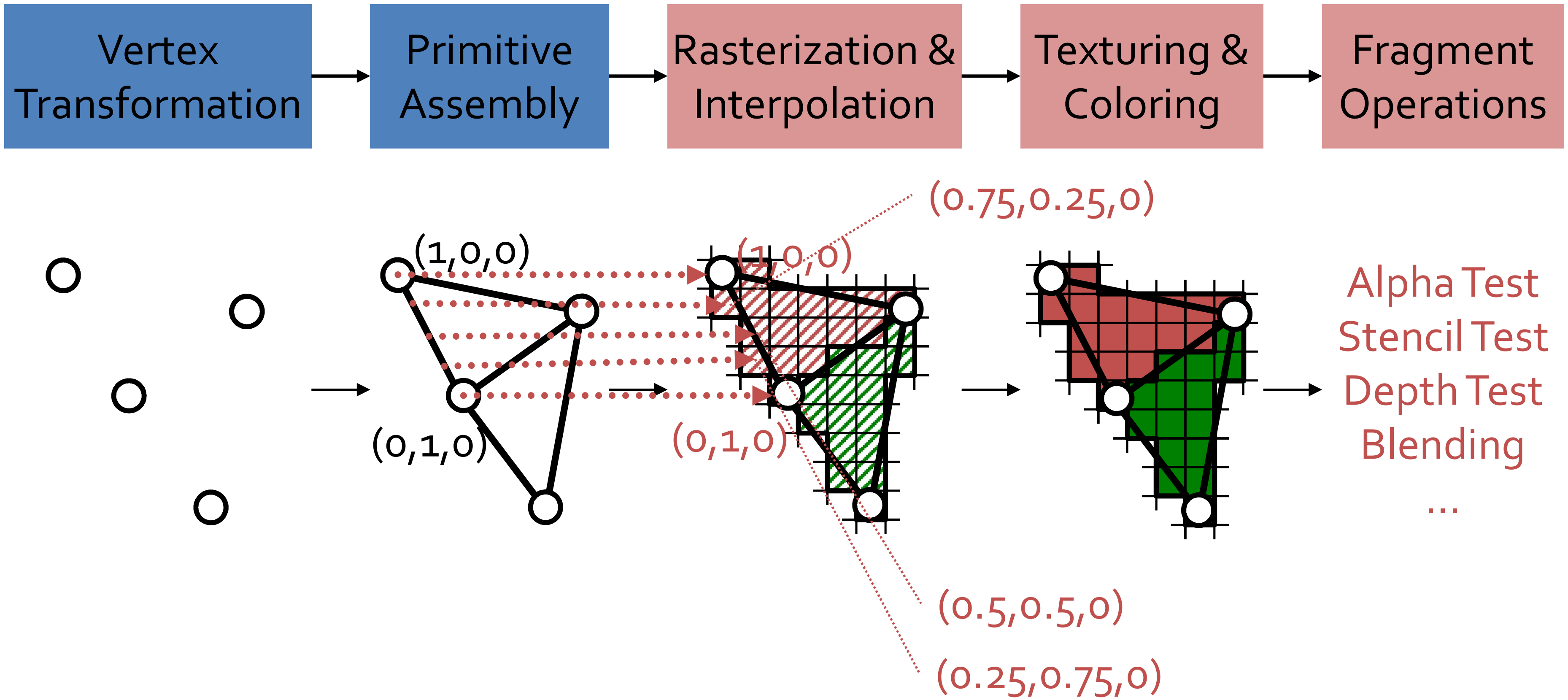
Fragment shader

- Input stream: fragments
- Output stream: fragments



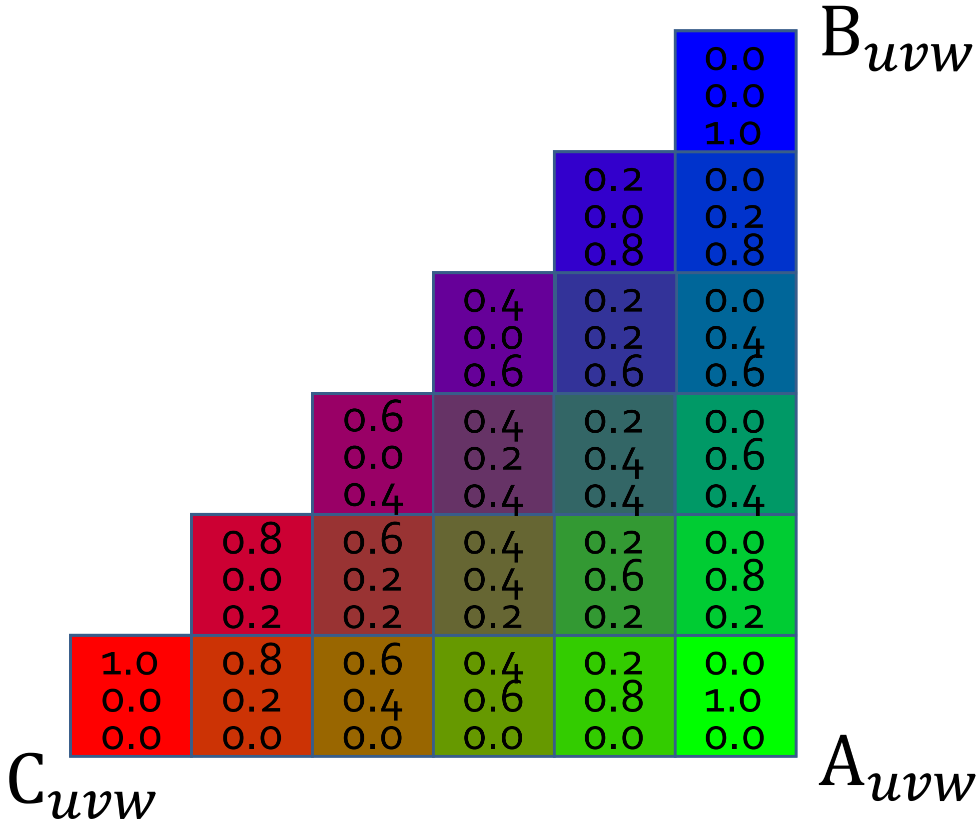
- One instance processes one fragment
- No knowledge of neighbouring fragments

Interpolation



Color Interpolation

- A.k.a. Gouraud interpolation
- $\mathrm{P} = u\langle Green\rangle + v\langle Blue\rangle + w\langle Red\rangle$

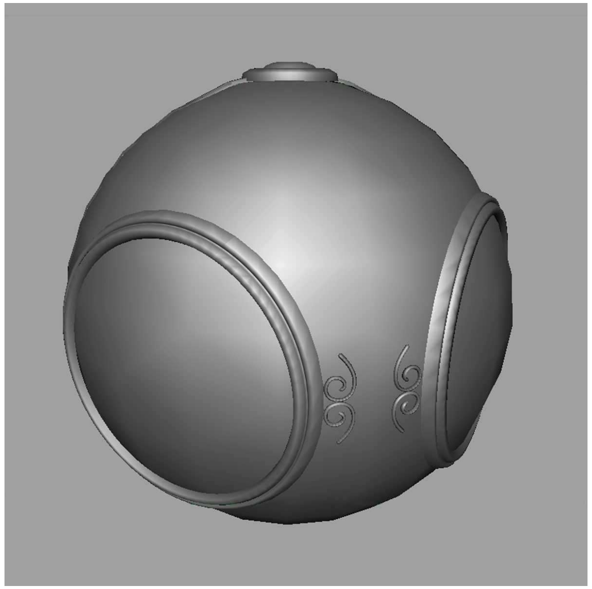


Application - per fragment lighting

per vertex

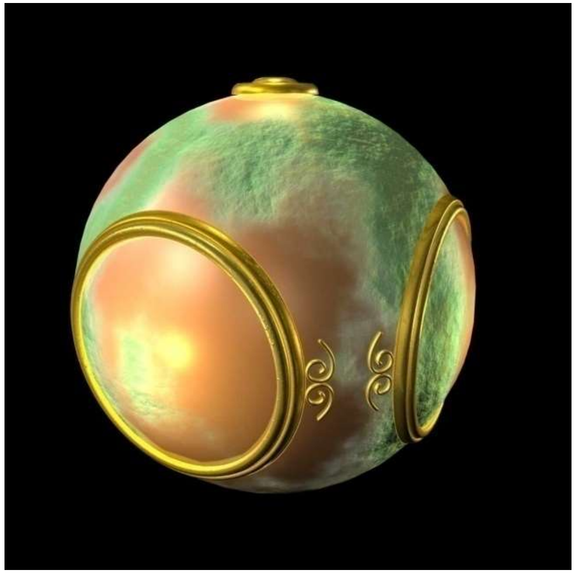
per fragment

Application - texture mapping



smooth shading

environment mapping



bump mapping