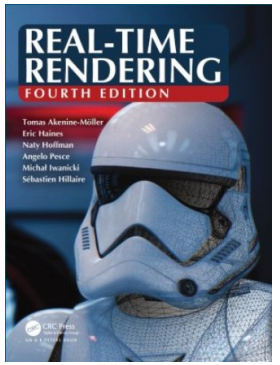
Introducción al Pipeline Gráfico en GPUs

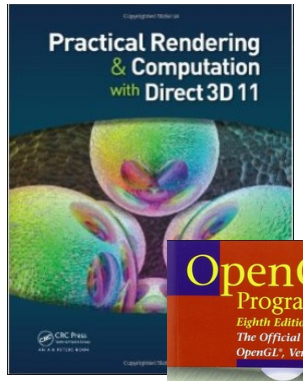
Francisco Lopez Luro (fll@bth.se)

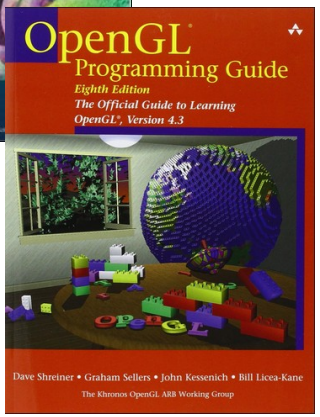
Overview

- Big Picture of the modern rendering pipeline
- CPU side APIs
- The driver
- GPU stages in more detail
- Walk through code sample – OpenGL
- Discussion about next-gen APIs

Referencias







Text books

- Real-Time Rendering, 3rd ed (or 4th)
- Practical Rendering and Computation with Direct3D 11
- Game Engine Architecture https://www.gameenginebook.com/

Practical and code oriented

- Learn OpenGL tutorials (learnopengl.com)
- Many technical blogs!
- Game Developer Conference presentations (many free on youtube)
- Open Source projects (godot engine, gameplay3d, panda3d, sokol)

GPU Graphics Rendering Pipeline

- Green boxes, fixed functionality (still configurable)
- Blue boxes, FULLY programmable stages (C-like language)

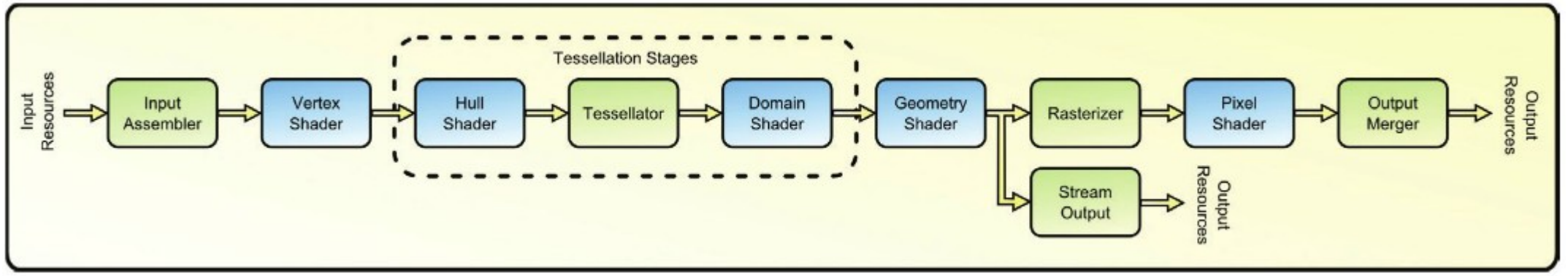
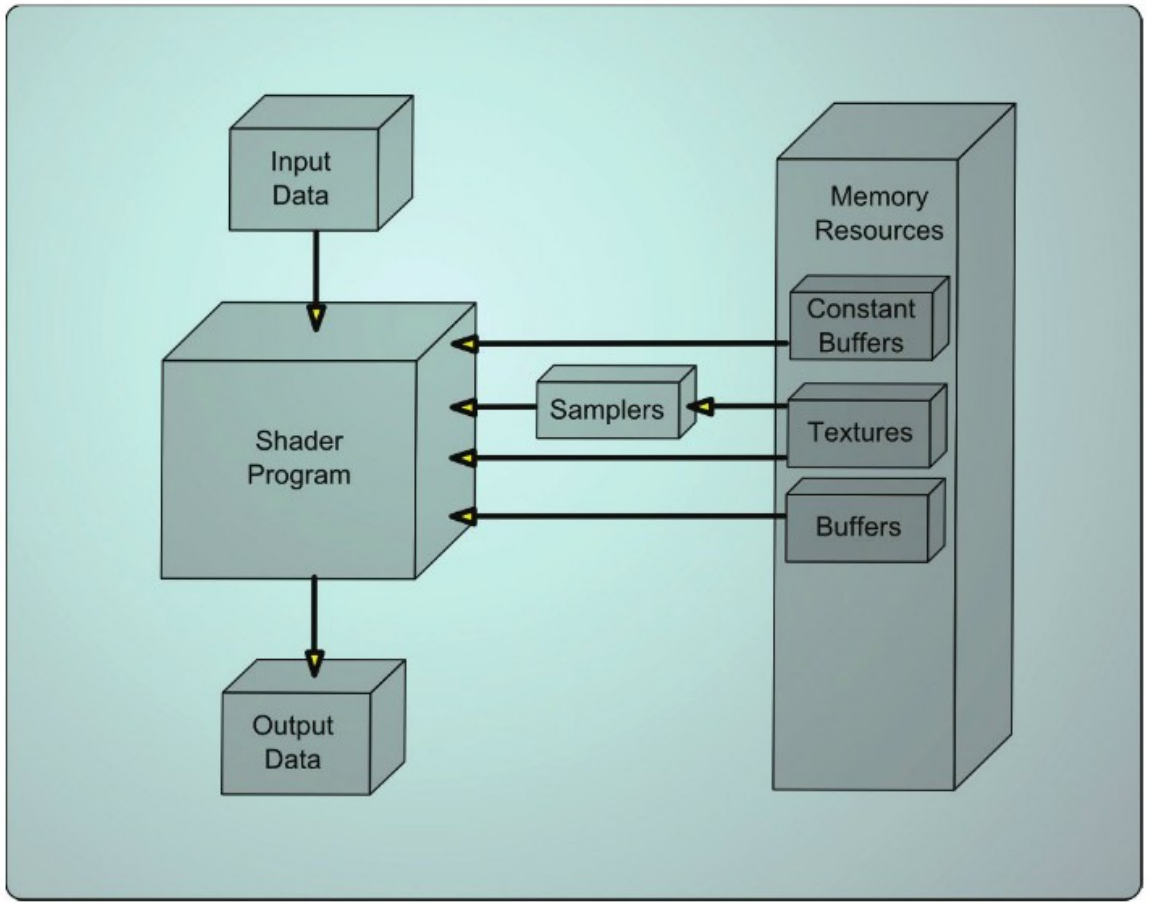


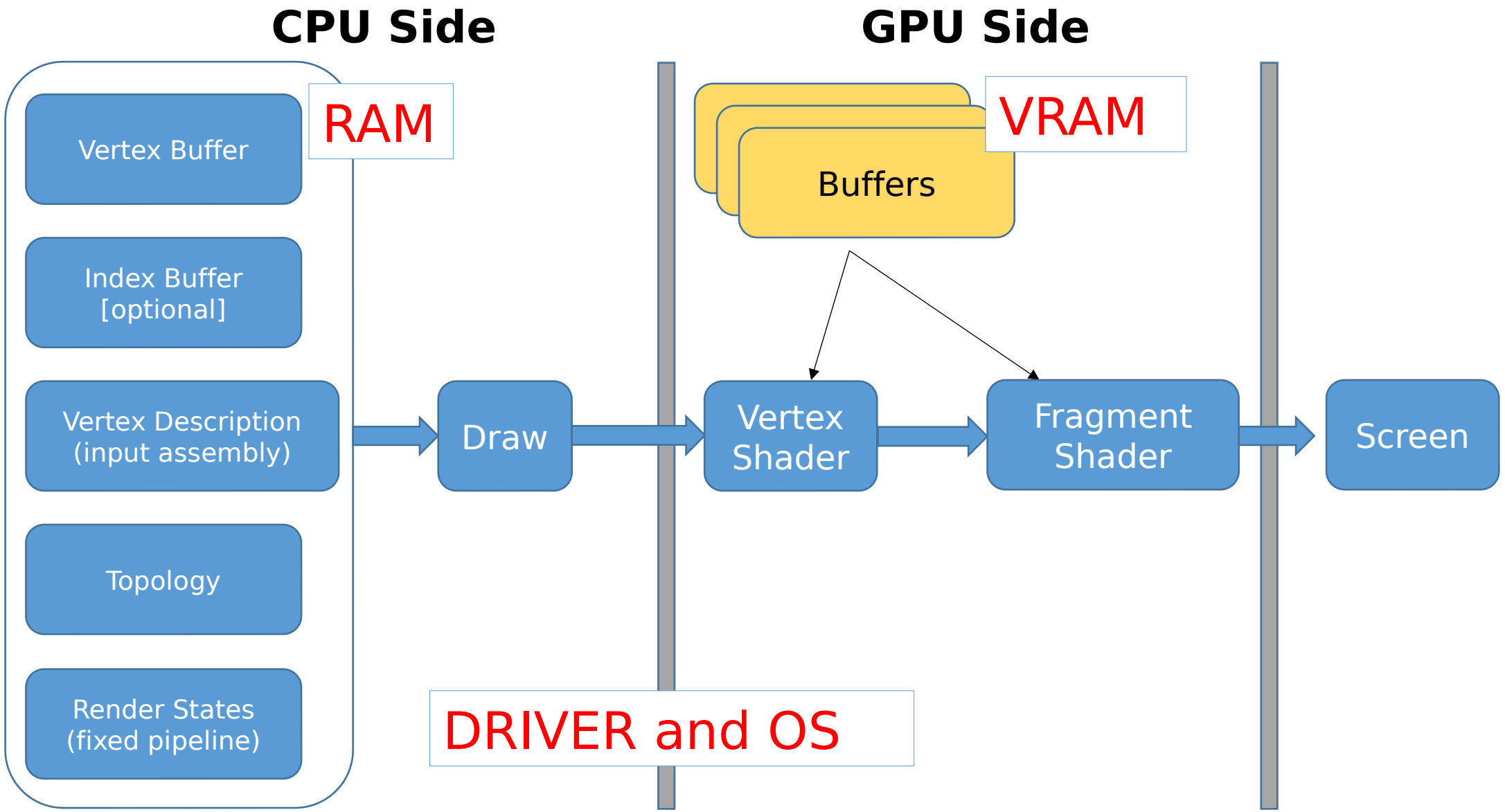
Image from Practical Rendering book

Rendering Pipeline

- **Common** to all programmable stages.
- Input and Output can be defined per stage (from and to other stages)
- They consume information from different sources:
  - Textures
  - Vertex Buffers
  - Index Buffers
  - Constants passed per “draw” action



Rendering Pipeline



Input Assembler Topology

[v0, v1, v2, …, v11]

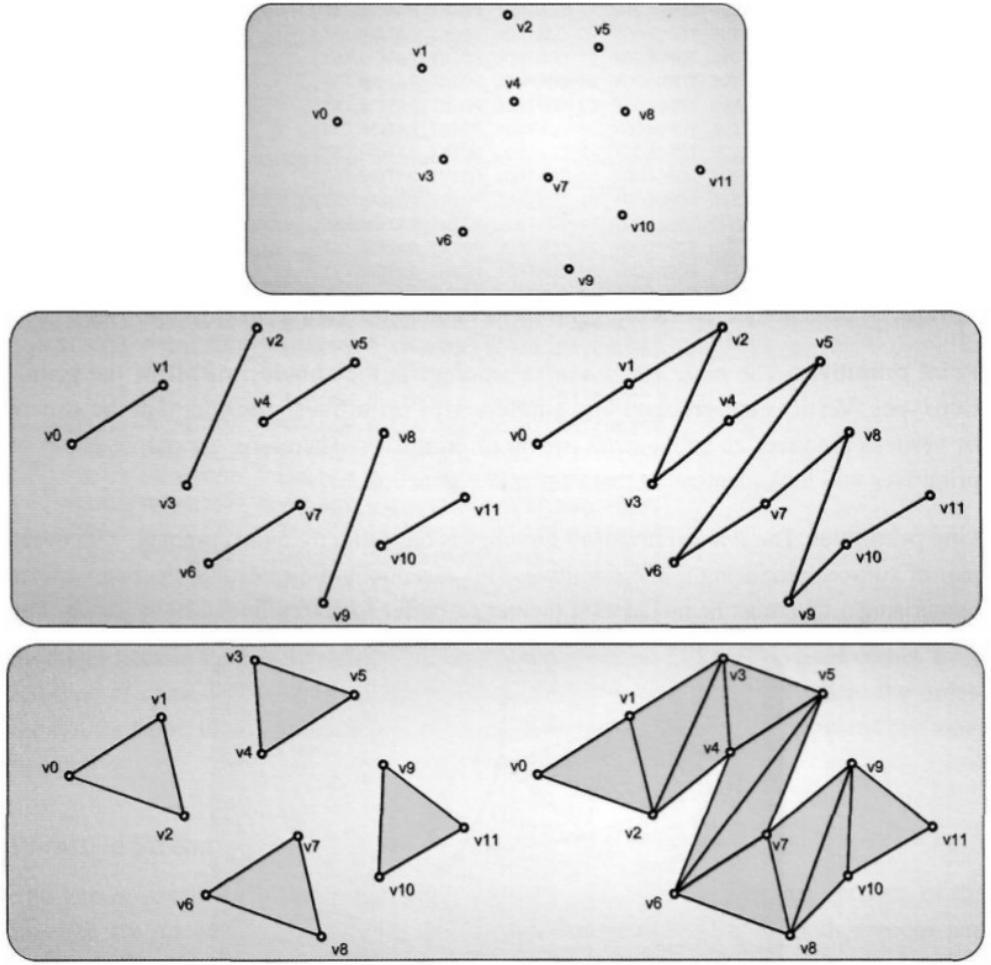
Points: [v0],[v1],..[v11]

Lines: [v0-v1], [v2-v3],…,[v10-v11]

Polyline: [v0-v1-v2-..-v10-v11]

Tri List: [v0-v1-v2],…,[v9-v10-v11]

Tri Strip: [v0-v1-v2],[v1-v2-v3],…, [v9-v10-v11]



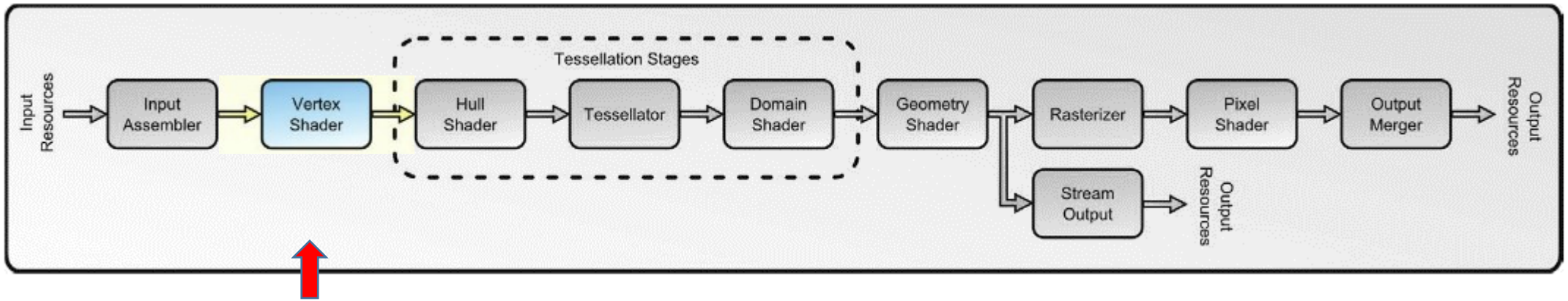
Input Assembler – Vertex “data”

- We do not only send Vertices…
  - Position
  - Normal
  - Tangent
  - Texture coordinates
  - Skeleton data
  - Vertex colour
  - **Anything** you want, can be sent along with the vertex as inputs to the IA

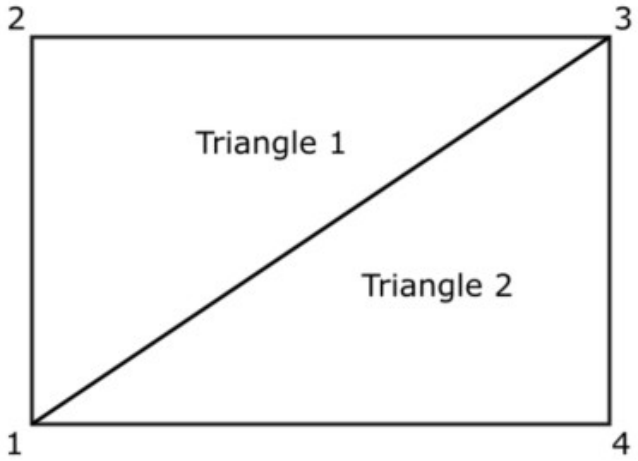
```
struct ColorVertex
{
  float x, y, z;        //Position
  float nx, ny, nz;     //Normal
  float u0, v0;         //Texcoord
};
```

Vertex Shader

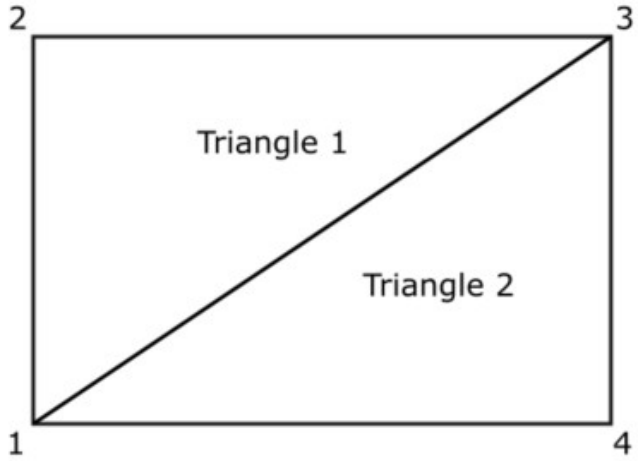
- First Programmable stage!
- Independent of the Topology (only sees a Vertex – with its data)
- The results can be sent to:
  - Hull/control shader
  - Geometry shader
  - Pixel Shader
- Send 900 Vertices, we have 900 executions of the same program.



Vertex Shader – Vertex vs Index Buffer



Vertex Buffer: [1, 2, 3 ; 1, 3, 4]
CLOCKWISE IN THIS EXAMPLE

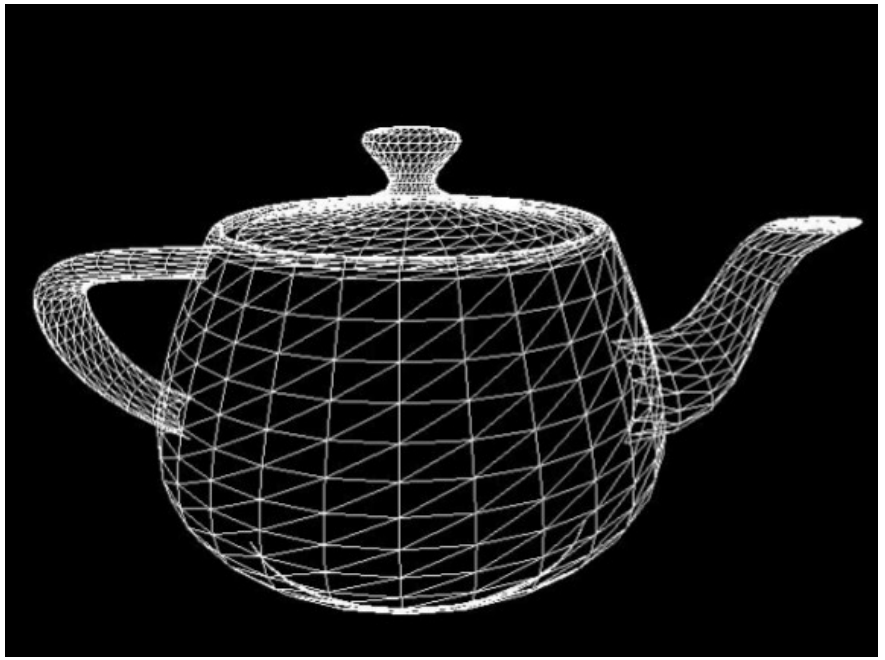


Vertex Buffer: [1, 2, 3, 4]
The VB needs no particular order

Index Buffer: [0, 1, 2 ; 0, 2, 3]
CLOCKWISE
(indexing from 0, like in C)

3D Object representation

- If you can render one Triangle, you can render anything.



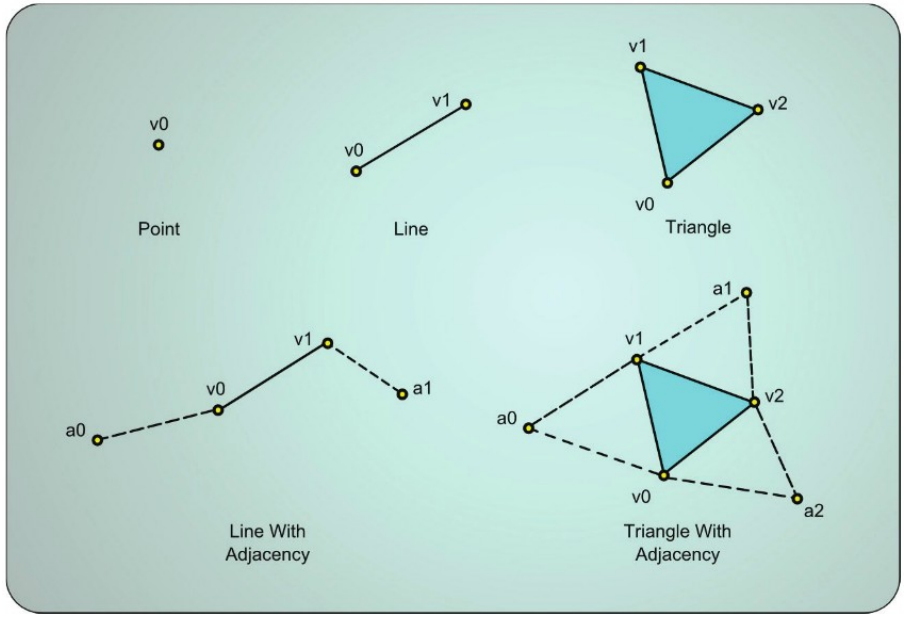
Utah Teapot

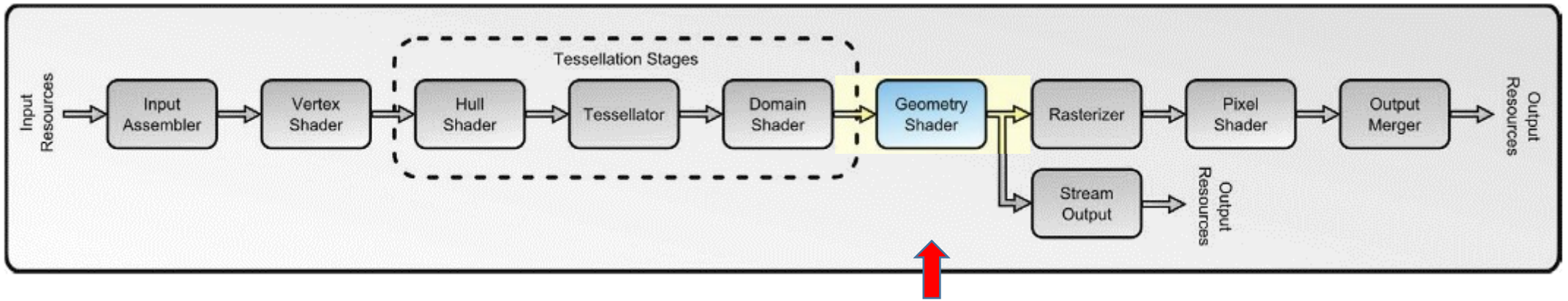
Transformations / Spaces (in WebGL)

http://www.realtimerendering.com/udacity/transforms.html

Geometry Shader (optional)

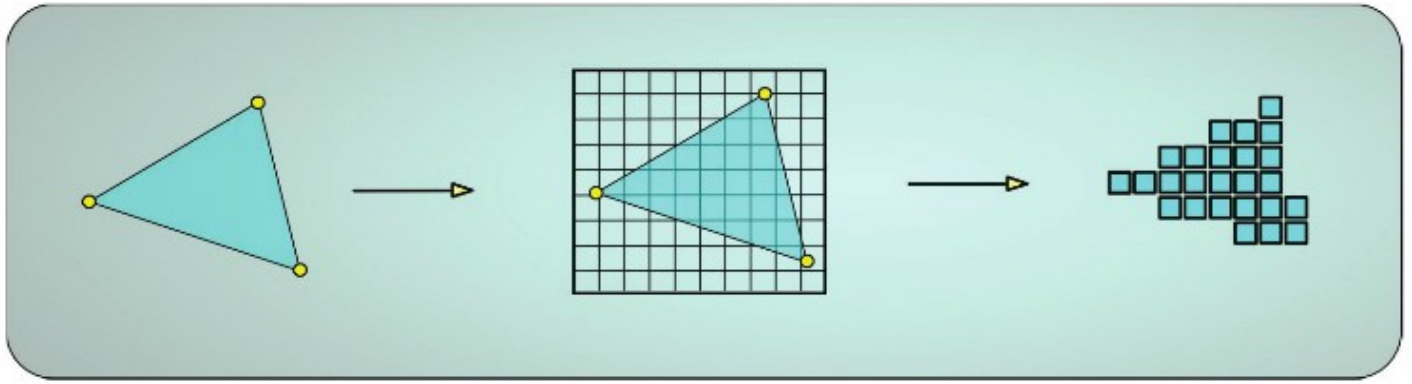
- A program is run **for each primitive** created
  - Adjacency information can be added to each primitive
- It can **discard** primitives, or **create** new primitives
  - Back face culling, Billboards for Particles (quads from points)
- Send 900 Vertices
  - With TRIANGLE LIST topology, gives 300 executions of the same program (300 triangles)
  - With TRIANGLE STRIP, gives (900 – 2) executions of the same program! (898 triangles)
- Results can go to Pixel Shader or Memory (for reuse)



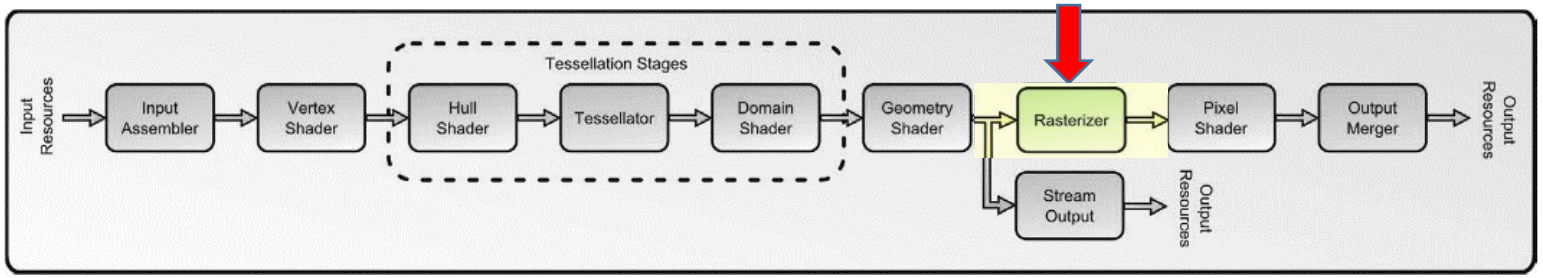


Rasterization

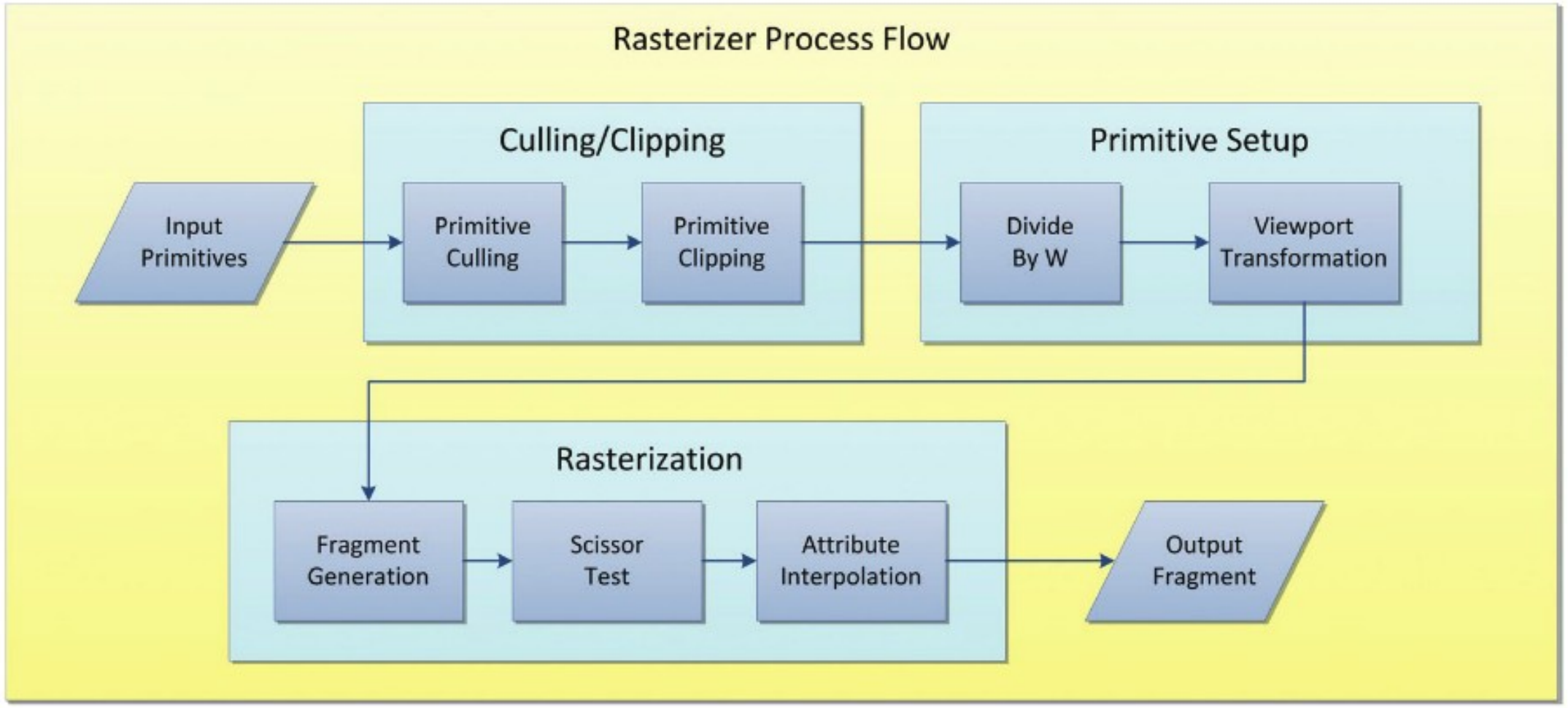
- The transformed geometry is mapped onto the active Render Target
- 1 **primitive can give** many **fragments**!



From Geometry to Fragments

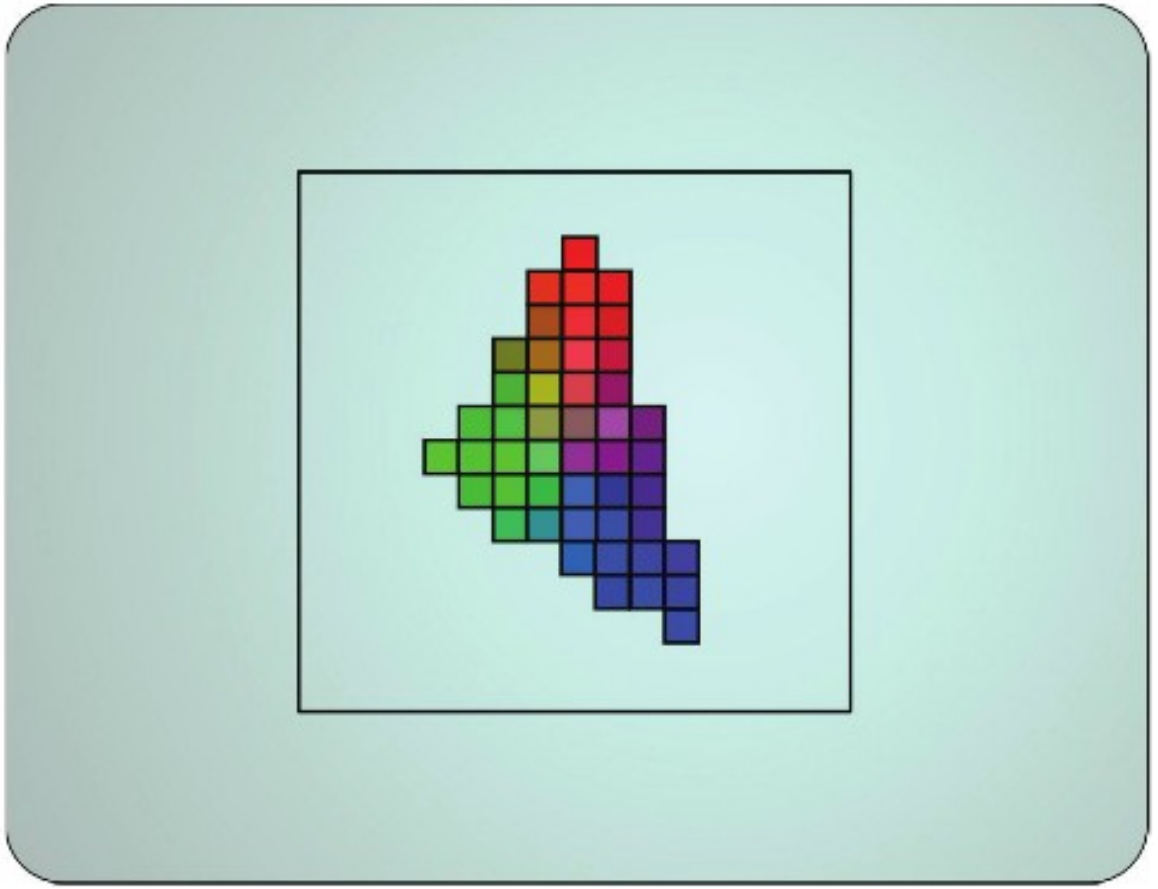


Rasterization



Rasterization (4)

- Attribute interpolation
  - Position
  - Colour
  - Normals
  - Others…
- Attribute interpolation can be disabled on an attribute basis. (each API uses different syntax)

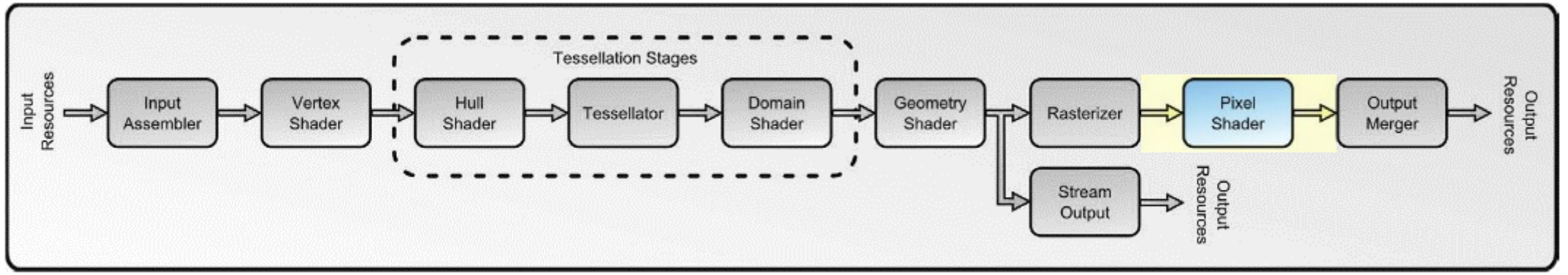


Rasterizer state

- User configuration of the Rasterizer stage
  - Culling front/back facing triangles
  - Winding order of triangles (to determine front vs back)
  - Wireframe / Solid
  - Multisampling
  - Antialiasing of Edges (Lines or Triangle Edges)
  - Scissor test enabled
  - ....
- API
  - OpenGL uses one function call for each option.
  - DirectX uses a Struct with all the options packed in.

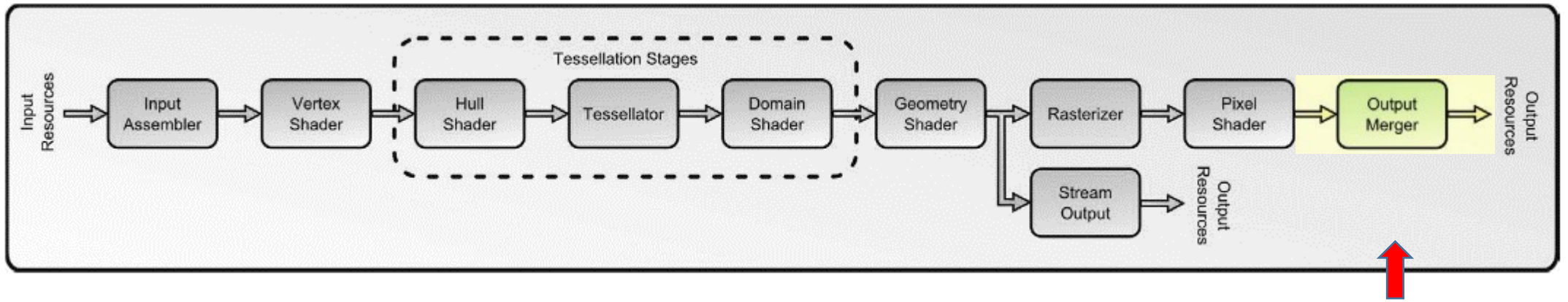
Pixel/Fragment Shader

- Last programmable stage in the pipeline
- One execution for each fragment (no primitives)
- Cannot change the location of the fragment!
- Works with interpolated data from the previous step
- Possible results are written to
  - Render Target
  - Memory Buffer (any location!) – this is a rather new feature



Output Merger

- Receives a colour and a depth value
- Depth test to check if new value overwrites existing
- Stencil test to only allow certain regions to be updated
- If Depth and Stencil pass,
  - Blending operations (combining colours with existing values)
- Can output to:
  - Render Targets, Memory Buffers (Random access) and Depth Stencil Buffer

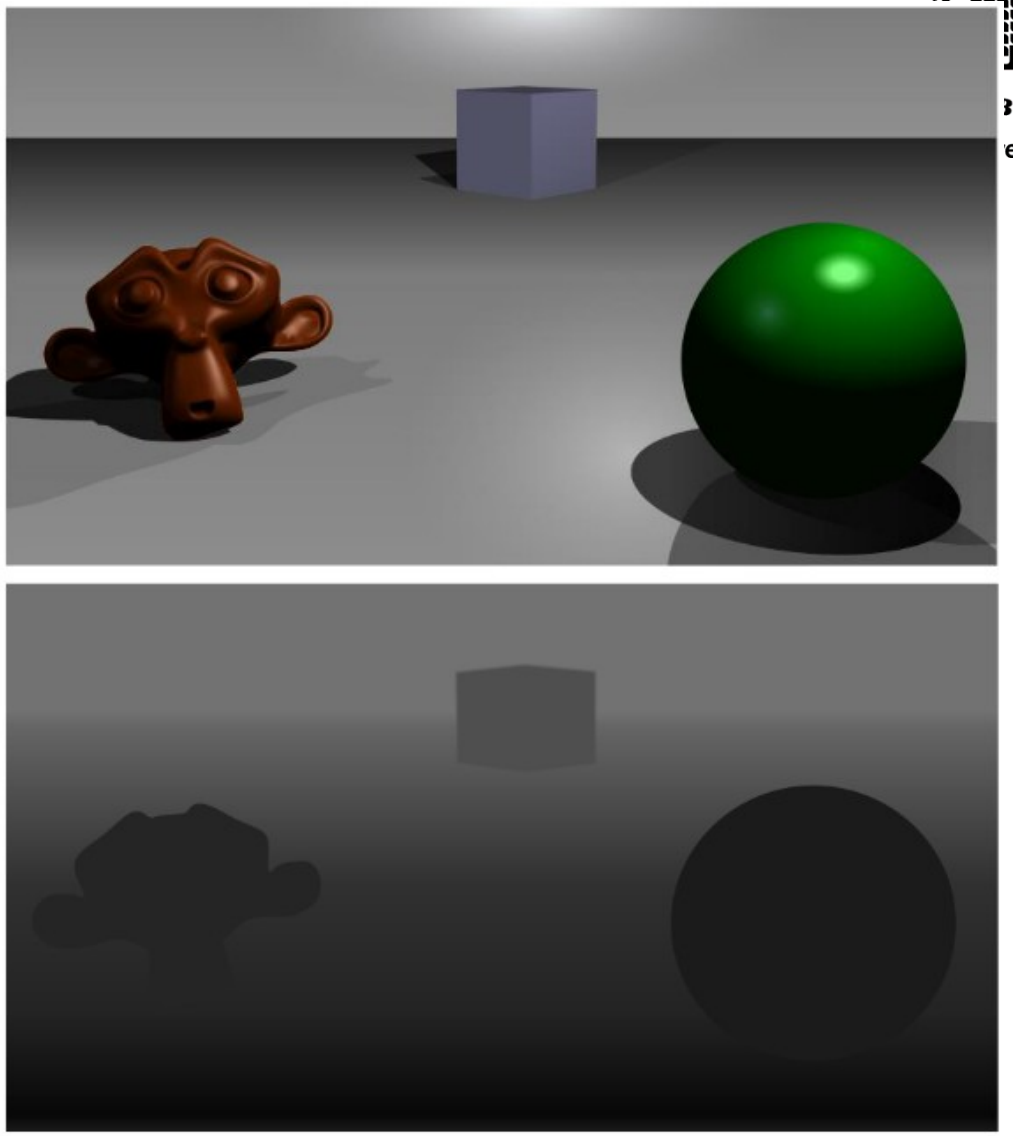


Outputs

- Store the distance of each fragment to the camera.
  - **After** all primitives have been rendered
  - 1 is far away, 0 is closer
    (check API settings)

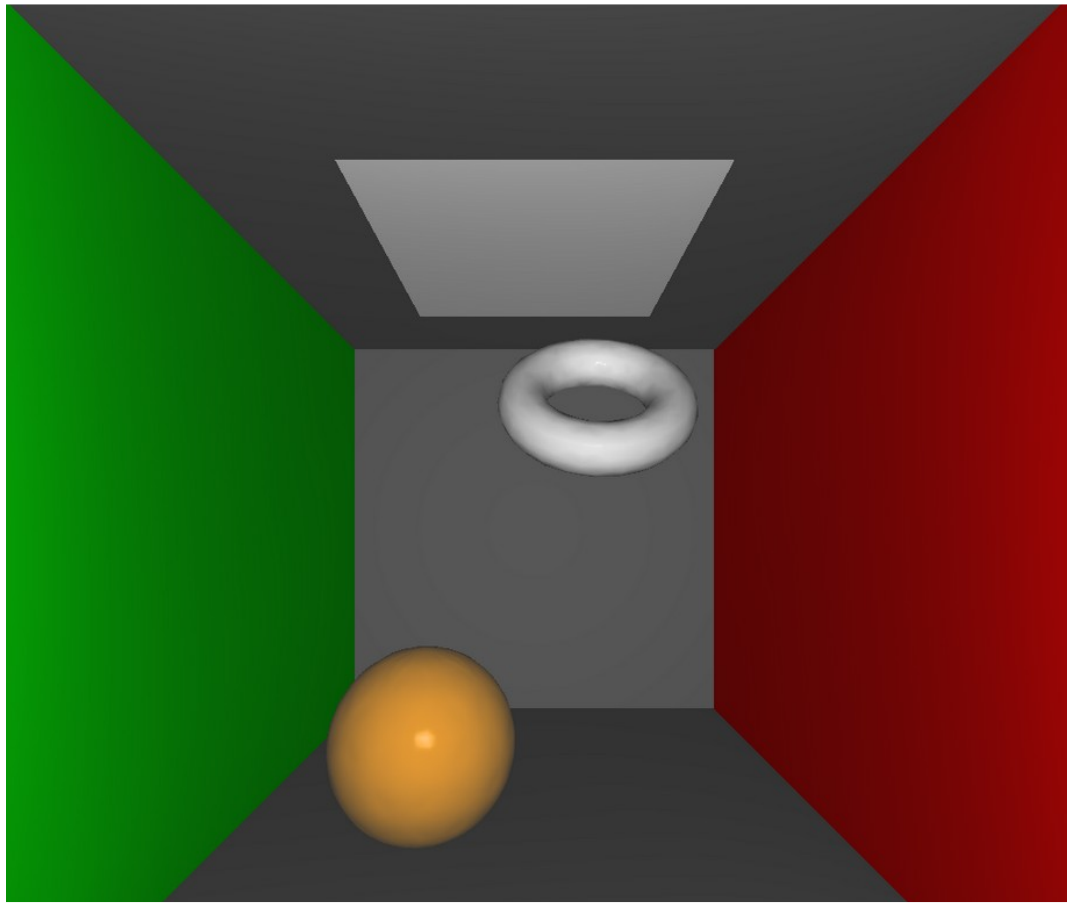
- An early test of Depth can minimize impact of Fragment Shader (pixel overdraw!)

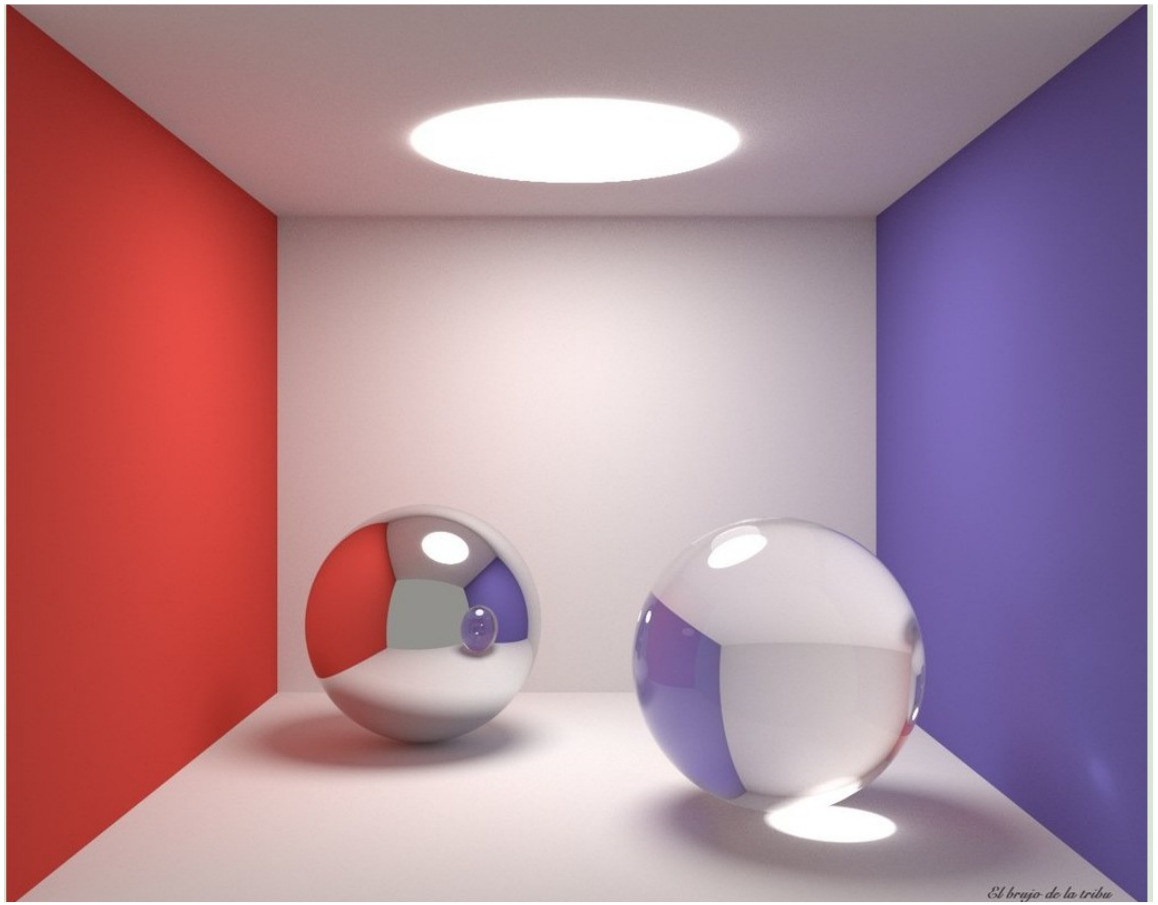
Colour buffer



Depth buffer

Lighting – local vs global



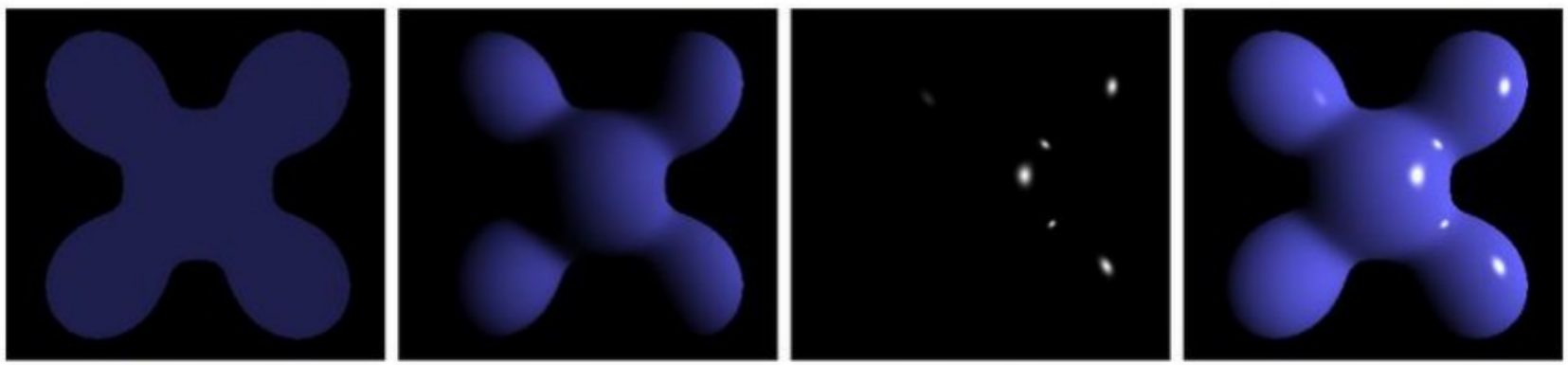


Phong – complete model

$$I_{\mathrm{p}} = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m \in \text{lights}} (k_{\mathrm{d}}(\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_{\mathrm{s}}(\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}})$$

$i_{m,s} = light\ m\ \boldsymbol{specular}$
$i_{m,d} = light\ m\ \boldsymbol{diffuse}$
$i_a = \boldsymbol{ambient}\ light$
$k_a, k_d, k_s = material$

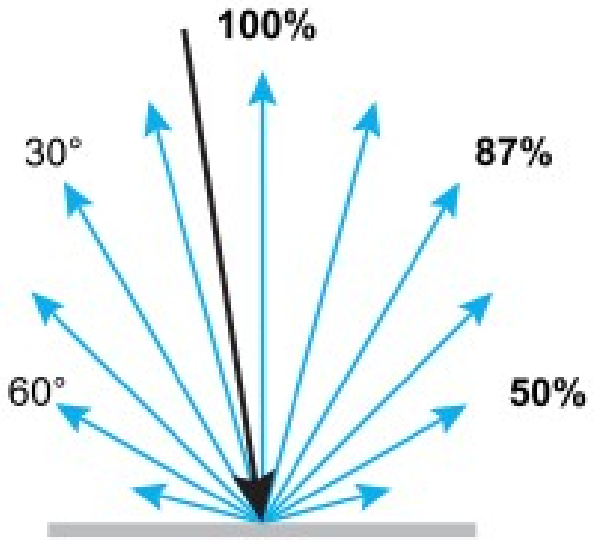
ALL HAVE THREE VALUES FOR R, G, B!



Ambient + Diffuse + Specular = Phong Reflection

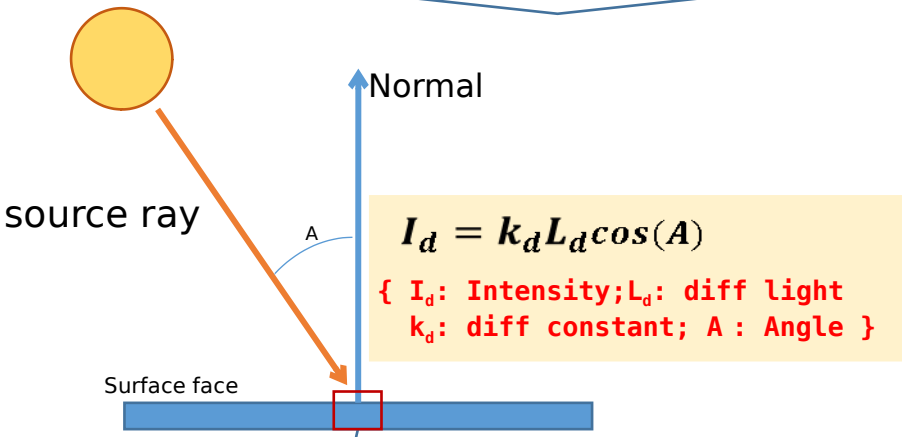
Phong – diffuse part

- Light “scatters” in all directions equally.
- Think in wood, clothes, anything far from perfect surface.



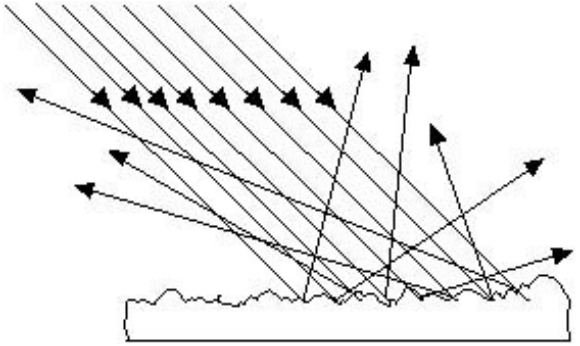
empirical model

- **viewer independent!**
- cos(A) must be >= 0



$$I_d = k_d L_d cos(A)$$

{ I_d: Intensity; L_d: diff light
k_d: diff constant; A : Angle }



reality

zoom in

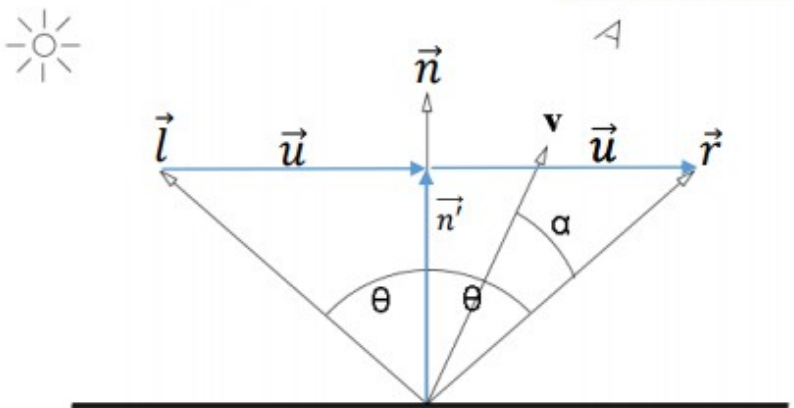
Phong – specular part

- Light “reflects” in one direction
- Think in mirror, metals, polished surfaces, some plastics.
  - “reflective” surfaces.

$$\vec{n}' = (\vec{n} \cdot \vec{l})\vec{n}$$

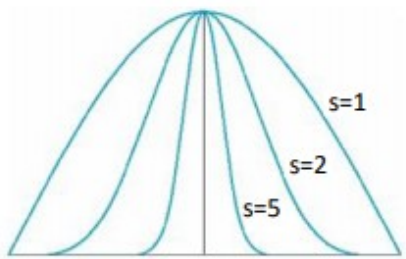
$$\vec{u} = \vec{n}' - \vec{l}$$

$$\vec{r} = \vec{l} + 2\vec{u} = \vec{l} + 2(\vec{n}' - \vec{l}) = 2(\vec{n} \cdot \vec{l})\vec{n} - \vec{l} = \vec{r}$$



$$I_s = k_s L_s (\vec{r} \cdot \vec{v})^s$$

- All vectors are assumed normalized
- **r** is the **reflection** of **l** with normal **n**
  - Calculation of **r** → demo
- **v** is the viewer direction
- **s** is the shininess factor



Ejemplos - OpenGL

- CPU – main, shader
- GPU – Vertex Shader, Fragment Shader

UV mapping

- On complex meshes and objects, usually we use hand-tailored UV mapping, and or multiple projector functions

Gracias...

...a los profes de la UNCOMA