2018/2019

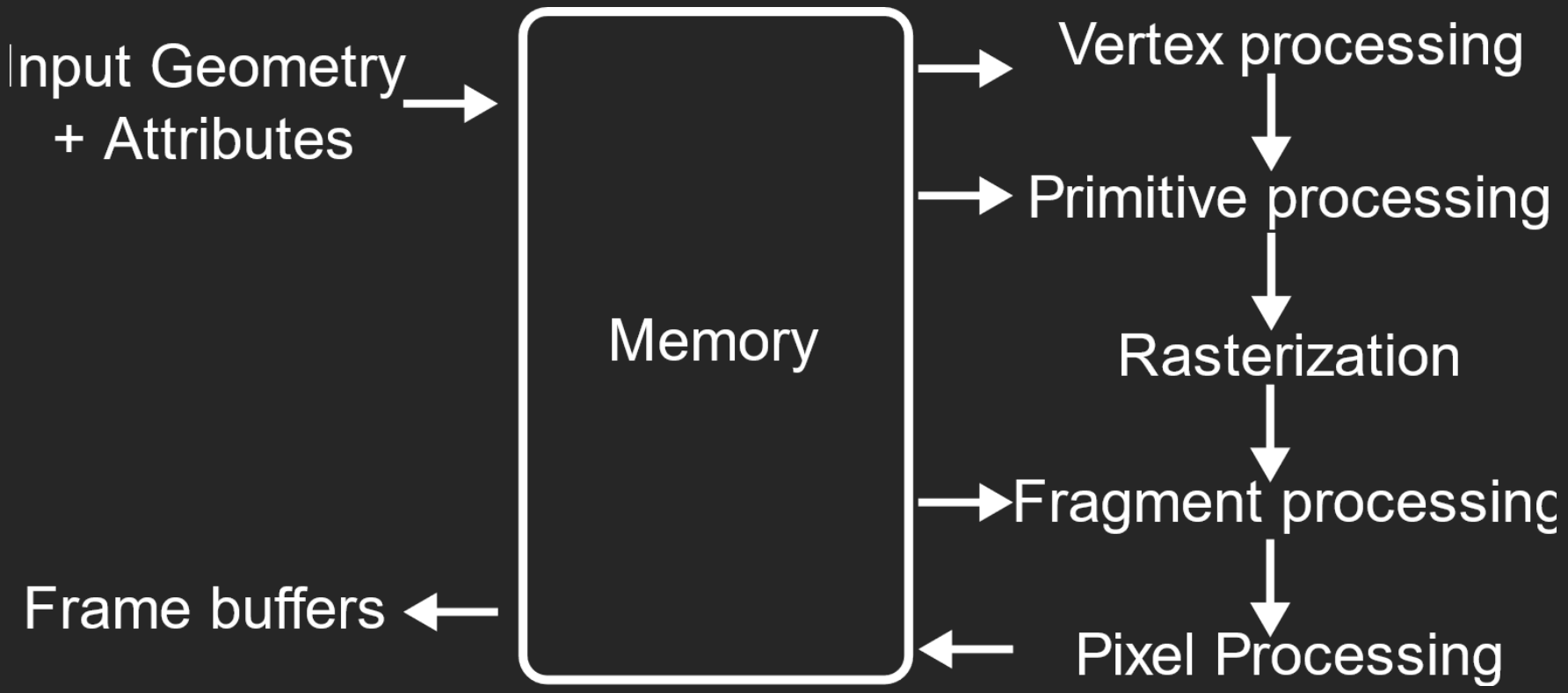
Rendering Pipeline

Evolução do OpenGL

- OpenGL 1.0 foi lançado em Julho de 1994.
- A pipeline era inteiramente "*fixed-function*"
  - As únicas operações disponíveis foram fixadas pela implementação do próprio OpenGL.

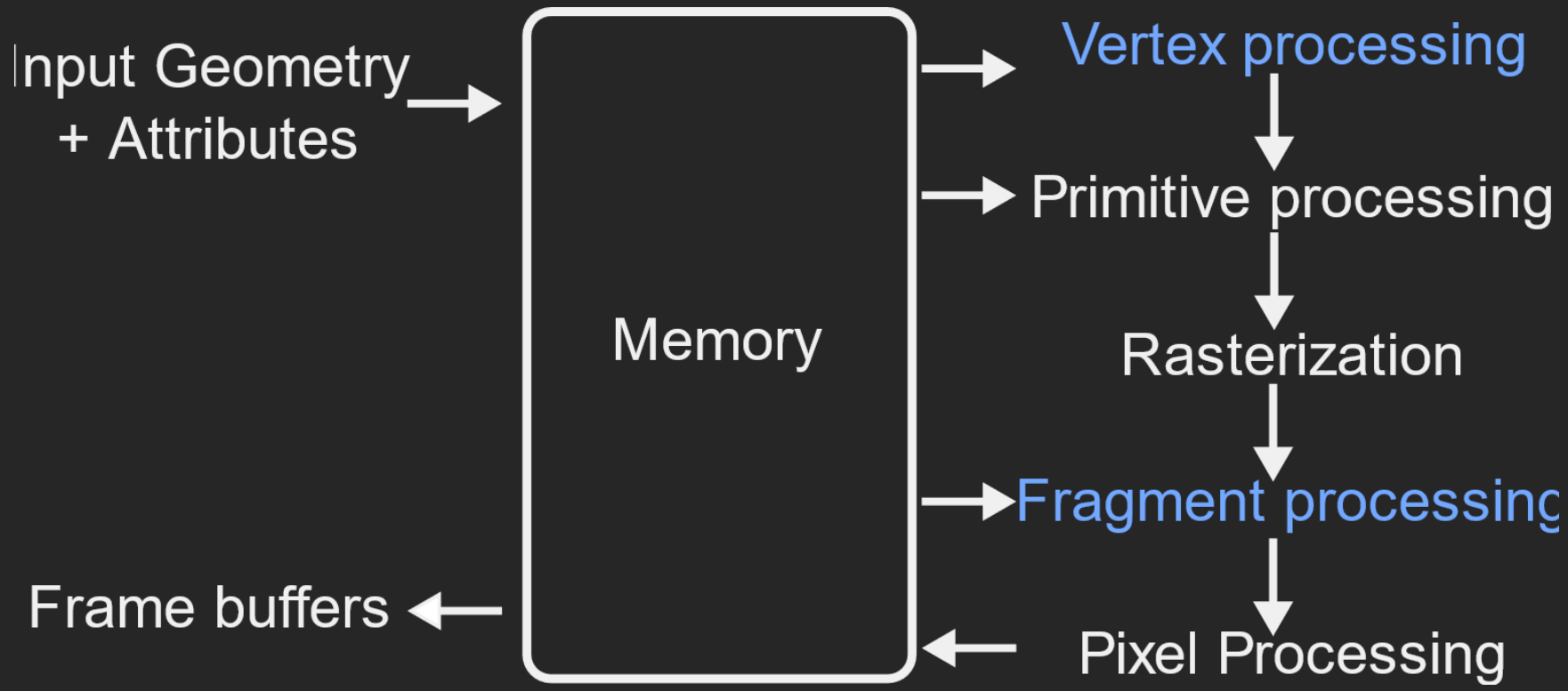
- A pipeline evoluiu
  - Mas manteve-se "fixed-function" desde as versões OpenGL 1.1 até 2.0 (Sept. 2004)

Pipeline stages overview



OpenGL 2.0

- OpenGL 2.0 acrescentou shaders programáveis
  - *vertex shading* para o processamento de geometria
  - *fragment shading* para o cálculo da cor de cada fragmento (pixel)
- No entanto, a fixed pipeline manteve-se disponível.


nput Geometry
+ Attributes
Memory
Vertex processing
Primitive processing
Rasterization
Fragment processing
Pixel Processing
Frame buffers

OpenGL 3.0 (2009)

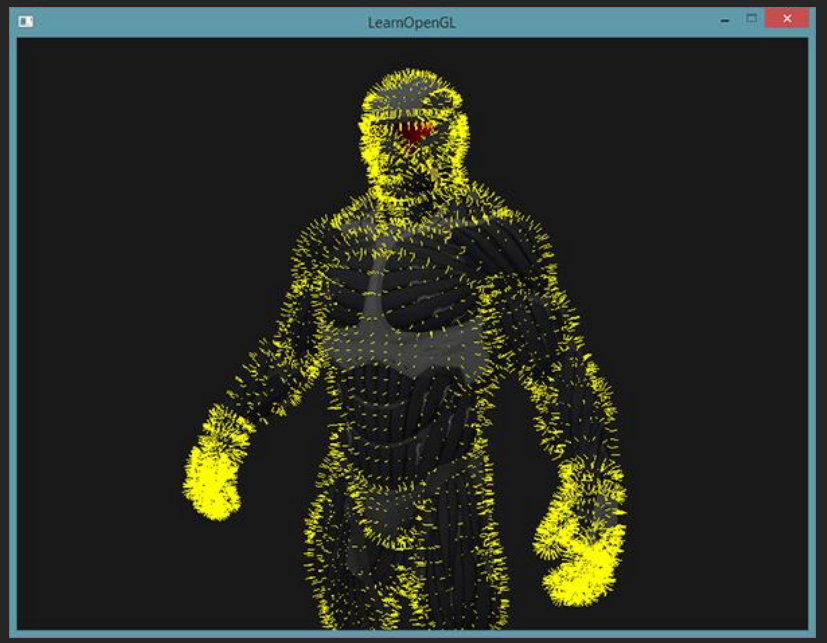
- OpenGL 3.0 começou a remoção de features
  - Troca de compatibilidade por eficiência
- Introduziu dois tipos de contexo diferentes para o uso de OpenGL:
  - Full
  - Forward-Compatible

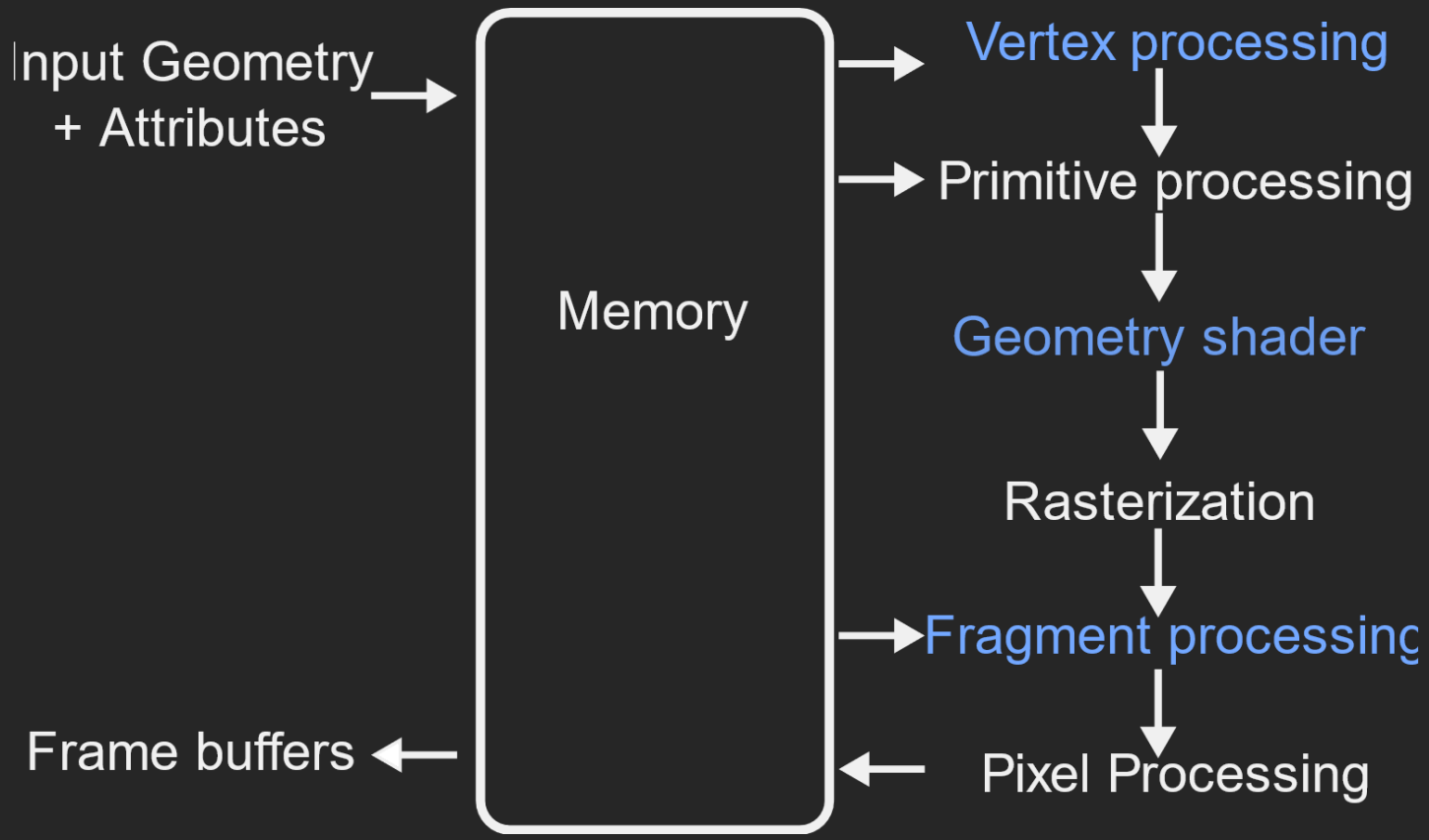
OpenGL 3.1

- Remoção completa da fixed-pipeline
  - Tornou-se obrigatório para as aplicações criarem os seus próprios shaders
- Tambem foram introduzidos os *buffers*, e quase todos os dados processados pelos shaders passam a residir em memória GPU

OpenGL 3.2

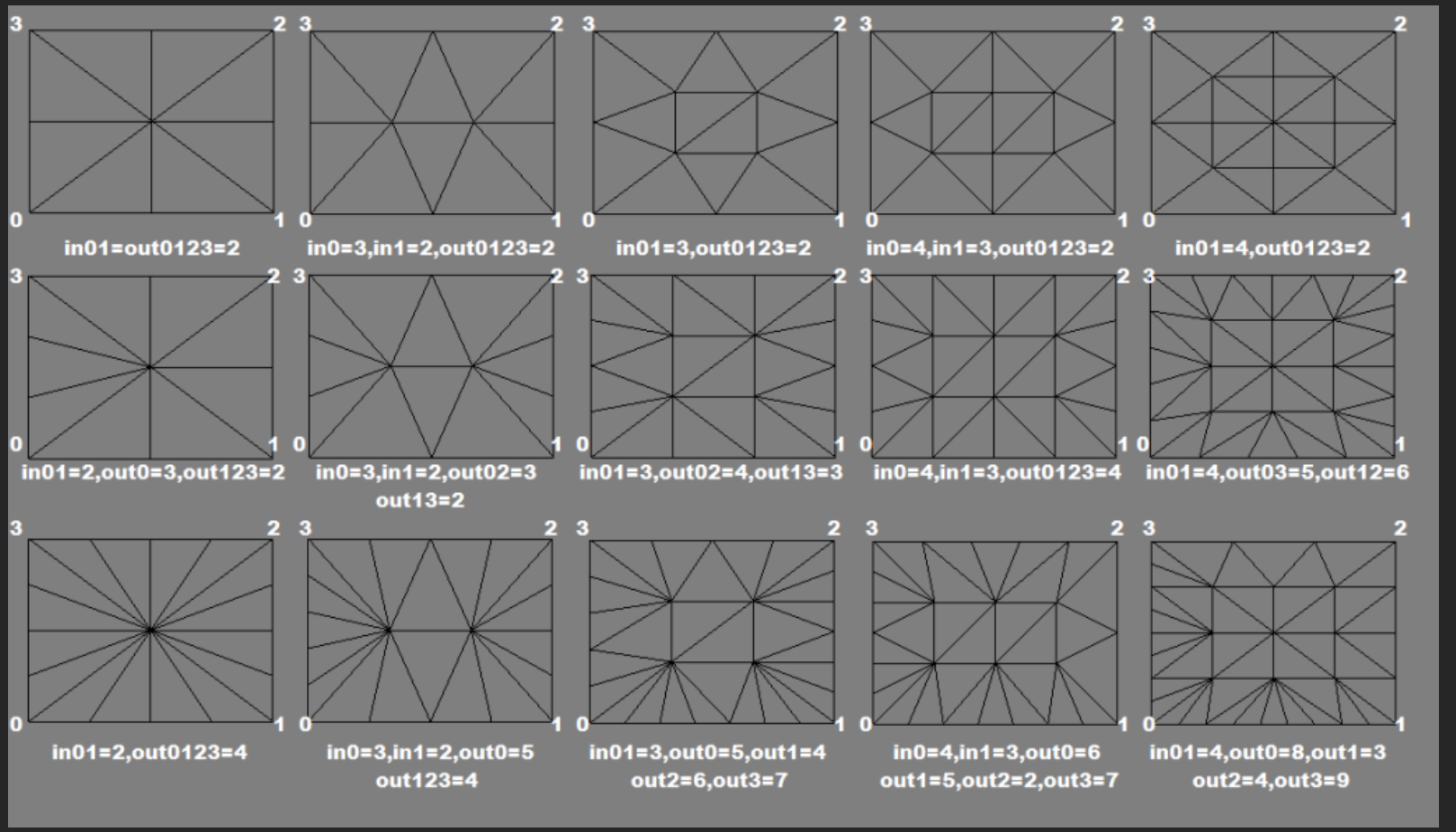
- Acrescentou uma nova fase programável opcional:
  - Geometry Shader
- Permite gerar nova geometria a partir dos dados fornecidos ao vertex-shader
- Usado por exemplo para debug




Input Geometry
+ Attributes
Memory
Vertex processing
Primitive processing
Geometry shader
Rasterization
Fragment processing
Pixel Processing
Frame buffers

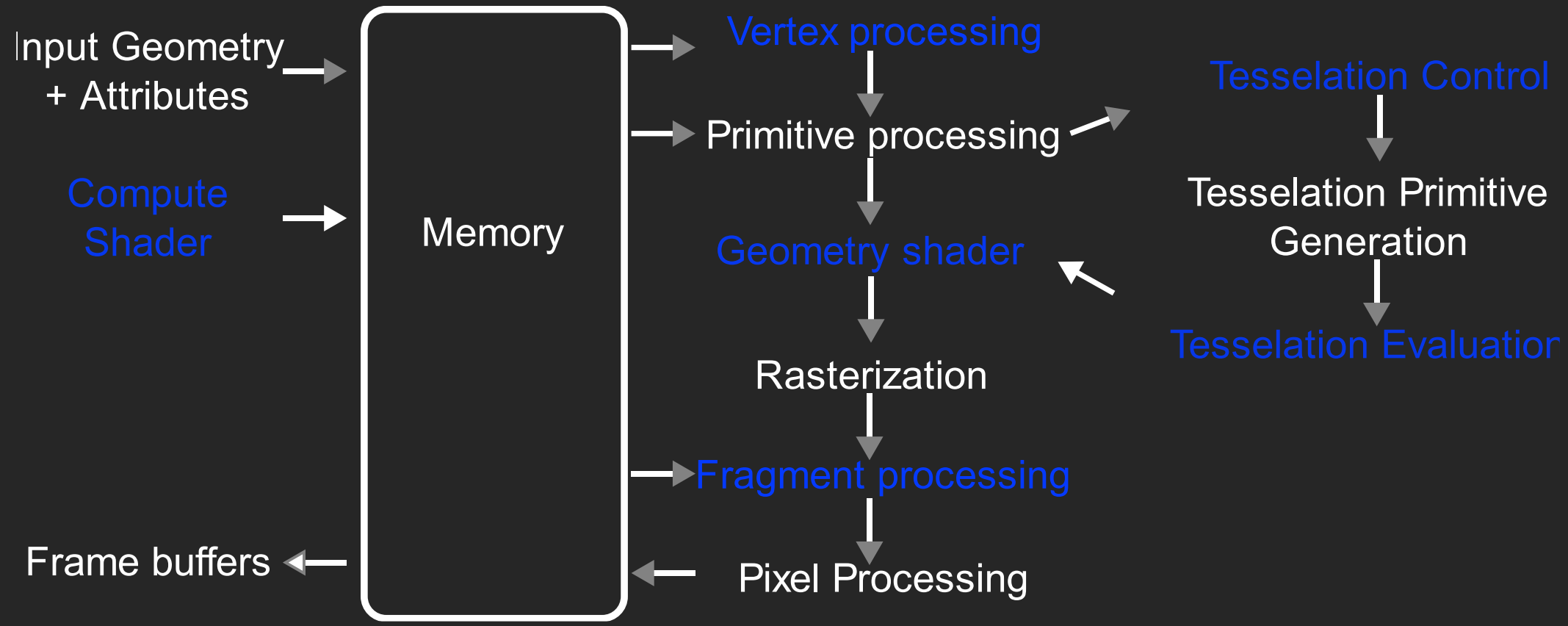
OpenGL 4.1

- OpenGL 4.1 (Jullho, 2010) introduziu duas etapas de processessamento adicionais – *tessellation-control* and *tessellation-evaluation*
- Usos:
  - LOD (Level of Detail)
  - Subdivisão adaptativa
    - Criar meshes de melhor resolução
  - Desenho de superficies implicitas, como as curvas/patches de Bezier.


3 2 0 1
in01=out0123=2
3 2 0 1
in0=3,in1=2,out0123=2
3 2 0 1
in01=3,out0123=2
3 2 0 1
in0=4,in1=3,out0123=2
3 2 0 1
in01=4,out0123=2
3 2 0 1
in01=2,out0=3,out123=2
3 2 0 1
in0=3,in1=2,out02=3
out13=2
3 2 0 1
in01=3,out02=4,out13=3
3 2 0 1
in0=4,in1=3,out0123=4
3 2 0 1
in01=4,out03=5,out12=6
3 2 0 1
in01=2,out0123=4
3 2 0 1
in0=3,in1=2,out0=5
out123=4
3 2 0 1
in01=3,out0=5,out1=4
out2=6,out3=7
3 2 0 1
in0=4,in1=3,out0=6
out1=5,out2=2,out3=7
3 2 0 1
in01=4,out0=8,out1=3
out2=4,out3=9

OpenGL 4.3

- OpenGL 4.3 introduziu o Compute Shader, que permite utilizar a placa gráfica para computações arbitrárias.
- A última versão de OpenGL é a 4.6 (Julho de 2017)

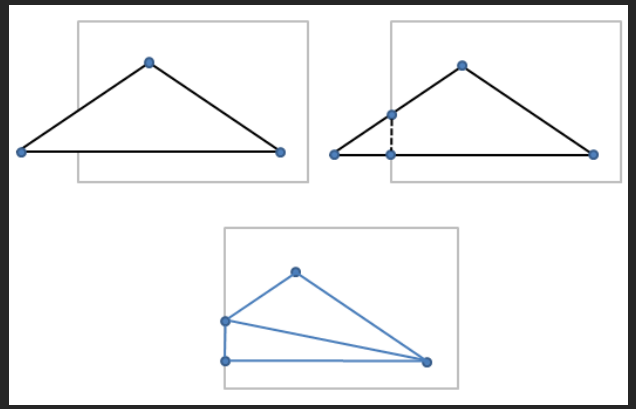

Input Geometry
+ Attributes
Compute
Shader
Memory
Frame buffers
Vertex processing
Primitive processing
Geometry shader
Rasterization
Fragment processing
Pixel Processing
Tesselation Control
Tesselation Primitive
Generation
Tesselation Evaluatior

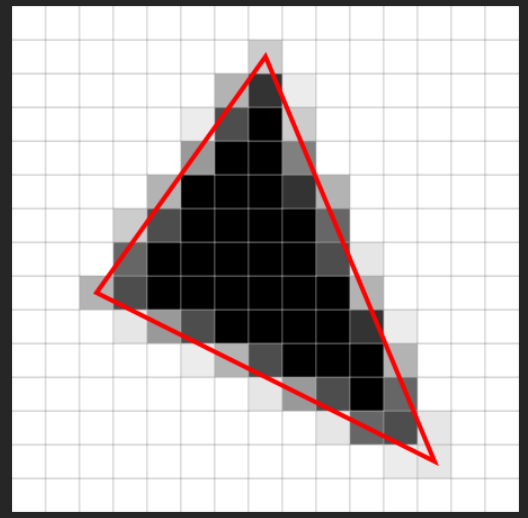
Vertex Shader

- É responsável por processar cada vertice fornecido à placa gráfica.
- Uso mais comum:
  - Converter as coordenadas locais de cada modelo para clip space.

Rasterização e Interpolação

- Antes da rasterização é realizado *clipping*
- A rasterização determina quais pixeis desenhar.
- A **interpolação** estima os valores dos parâmetros variaveis que vão dar entrada no fragment shader, a partir dos seus valores conhecidos nos **vertices**.



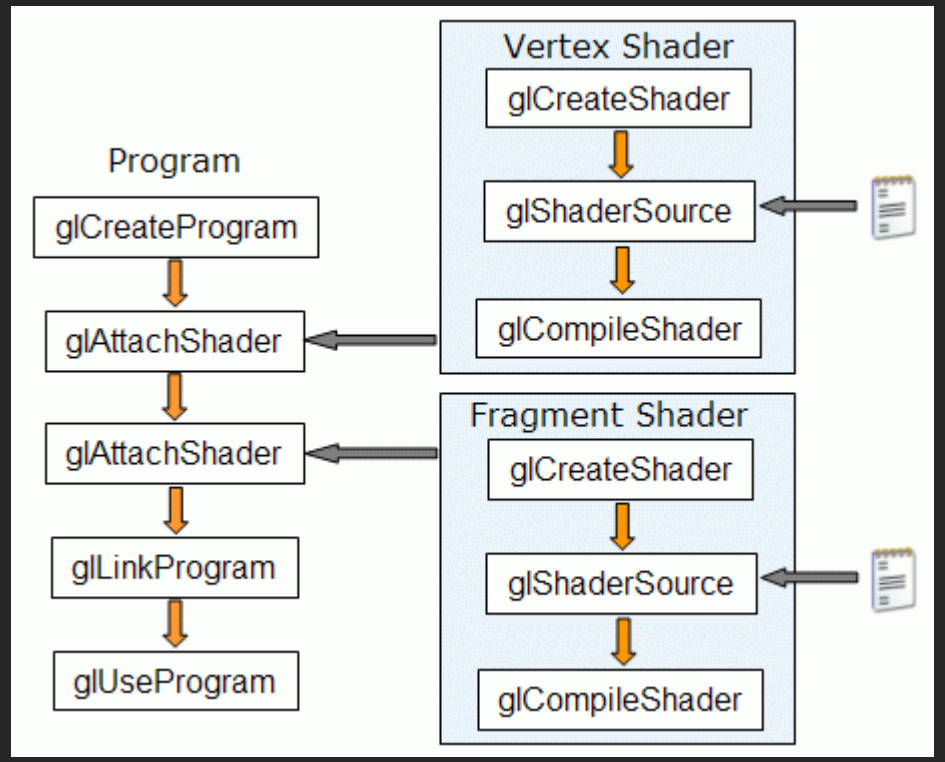


Fragment Shader

- Irá executar uma vez por cada fragmento resultante da rasterização.
- É o shader que irá efetivamente definir a cor de cada pixel de cada triângulo.
- Usos comuns:
  - Cálculo de iluminação
  - Aplicação de texturas
  - Efeitos de pós-processamento

Como utilizar OpenGL moderno?

- Shaders são programas, logo devem ser:
  - Alocados
  - *Attached*
  - *Compilados*



Comunicação de dados?

- São usados *Vertex Array Objects:*
  - Compostos por vários *Vertex Buffer Objects*
    - *Cada VBO representa um atributo*
      - *Posição*
      - *Normal*
      - *Textura*
- Texturas
- *Uniforms*
- *Draw Calls:*
  - *glDrawArrays*
  - *glDrawElements*

Exemplo de Vertex Shader: Phong

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 tex;

layout (std140) uniform Matrices
{
    uniform mat4 projection;
    uniform mat4 view;
};

uniform mat4 model;

out vec3 vPosition;
out vec3 vNormal;
out vec2 vTexCoord;

void main()
{
    vPosition = (model * vec4(position, 1.0)).xyz;
    vNormal = normal;

    vTexCoord = tex;

    gl_Position = projection * view * model * vec4(position, 1.0);
}
```

Exemplo de Fragment Shader: Phong

```
#version 330 core

in vec3 vPosition;
in vec3 vNormal;
in vec2 vTexCoord;

out vec4 FragColor;

/*
...
*/

uniform vec3 cameraPos;

void main()
{
    vec3 specColor = vec3(0);
    vec3 diffColor = vec3(0);

    vec3 vdir = normalize(cameraPos - vPosition);
    vec3 ldir = normalize(light.xyz - vPosition);
    vec3 normal = normalize(vNormal);

    float diffInt = dot(ldir, normal);

    if (diffInt > 0)
    {
        float distance = length(light.xyz - vPosition);
        float attenuation = 1.0 / (constant + linear * distance +
quadratic * (distance * distance));

        diffColor = attenuation * lightIntensity * diffuse *
diffInt;
        vec3 halfVector = normalize(vdir + ldir);

        float specInt = max(dot(halfVector, normal), 0.0f);
        specColor = vec3(1.0) * specIntensity * pow(specInt,
shininess);
    }

    FragColor = vec4(max(diffColor + specColor, diffuse * 0.25),
1.0f);
}
```