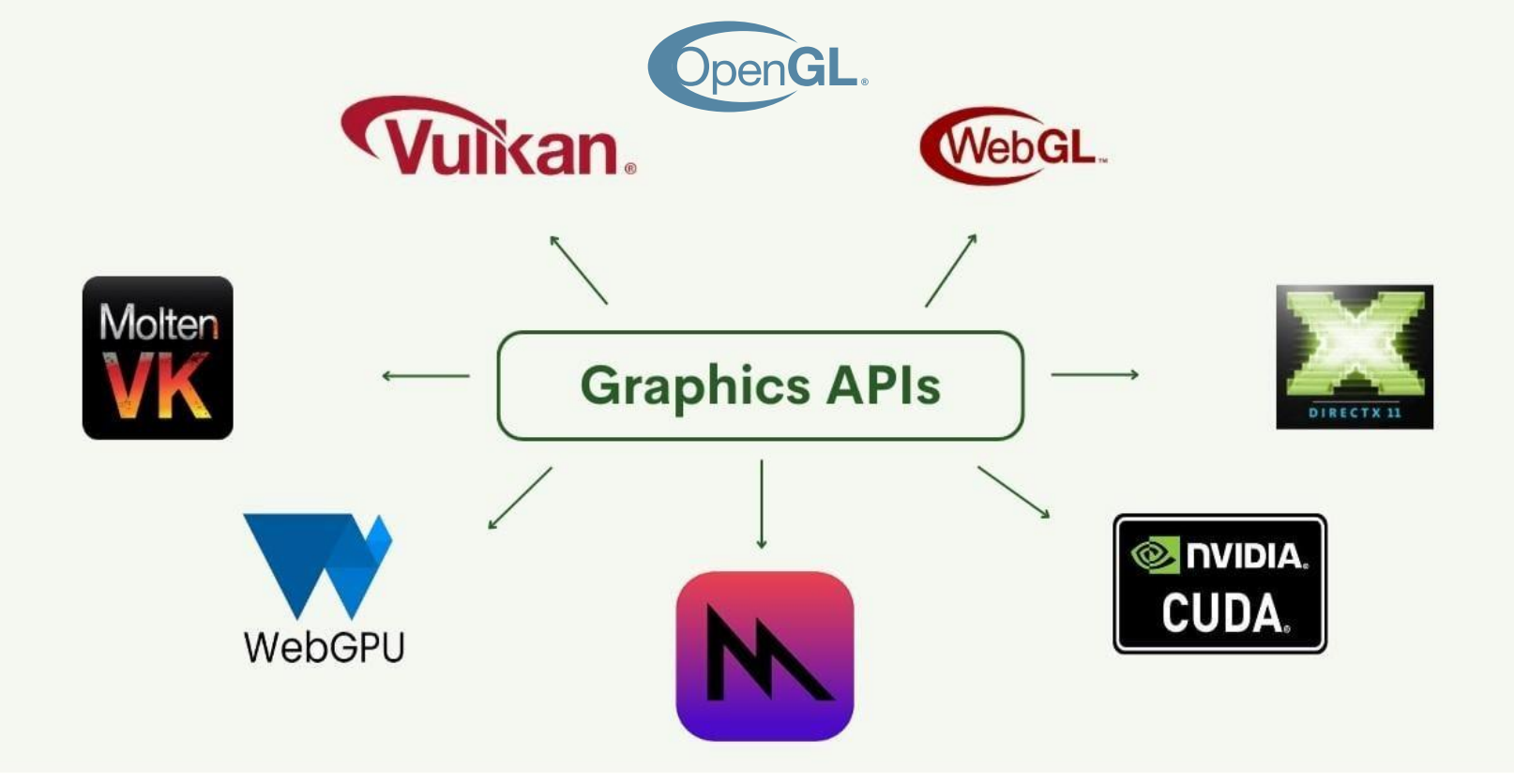
Insper

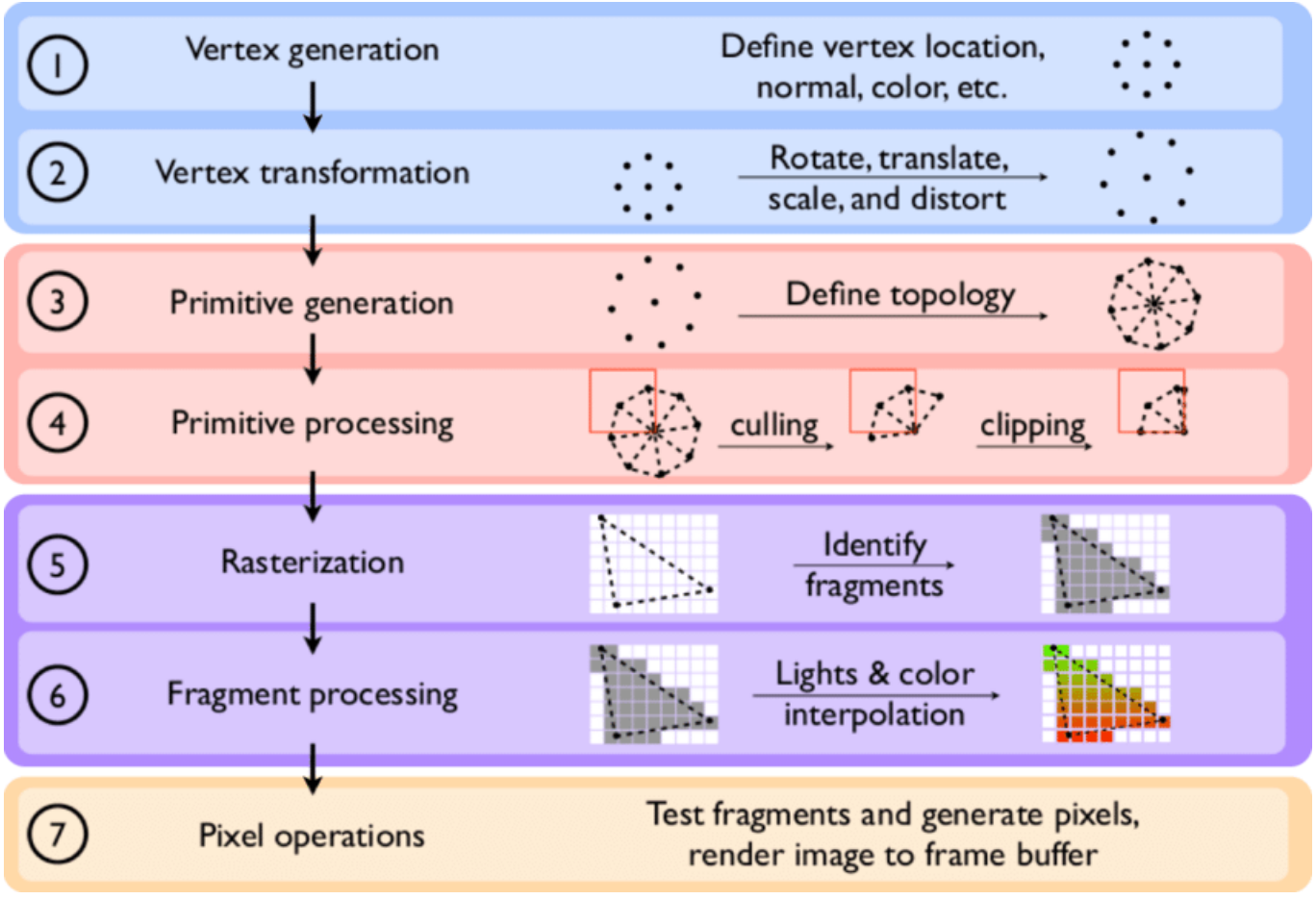
Computação Gráfica

Pipeline Gráfico & Shaders 1

Pipeline de Rasterização

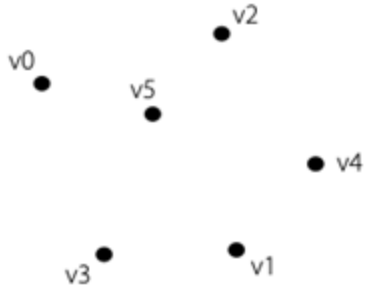


Pipeline de Rasterização

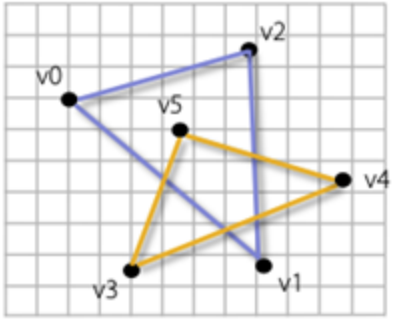


Insper

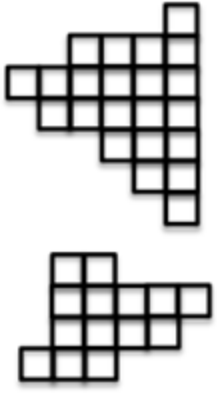
Entidades na Pipeline



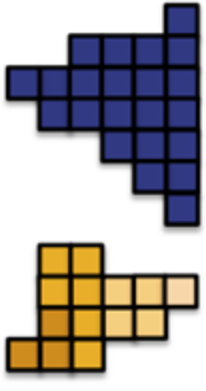
Vertices



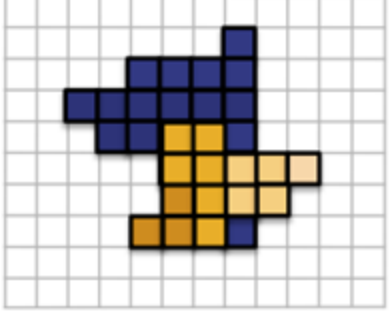
Primitives



Fragments



Fragments (shaded)



Pixels

Insper

Implementações de Pipelines Gráficos : GPUs

- Tudo poderia ser feito em CPU, porém, mais lento.
- Mais intuitivo/otimizado.
- Simulações em tempo real.

16.384 CUDA cores

32 threads

Insper

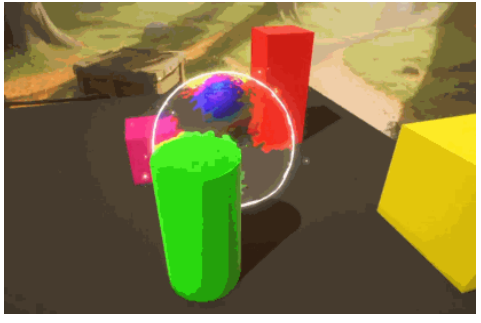
Implementações de Pipelines Gráficos : GPUs

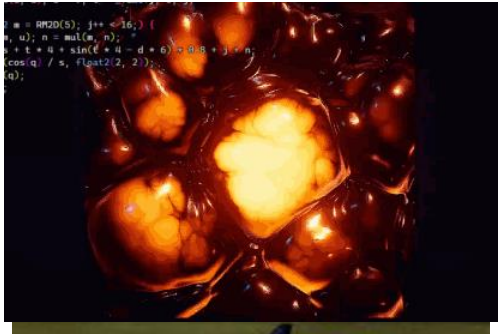
Papel da CPU

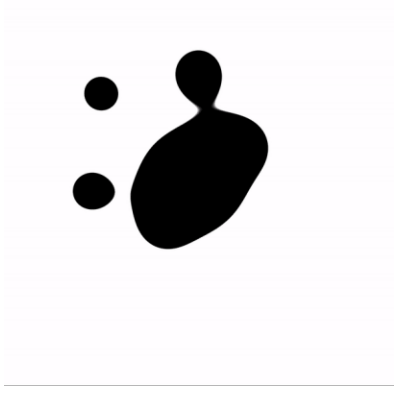
- A CPU é responsável por enviar os dados para a GPU, gerenciar o pipeline gráfico e memória (Buffers).
- Controla o que a GPU vai renderizar pois cuida do movimento de objetos e players em uma cena.
- Detecção de colisões (na maioria das vezes)
- Culling de objetos, escolha de LOD.
- Mais otimizada para certas operações.
- Outros papéis mais intuitivos: Comunicação multiplayer, áudios e sons.

O que são shaders?

- Códigos feitos para atuar em GPU, geralmente com ênfase visual.







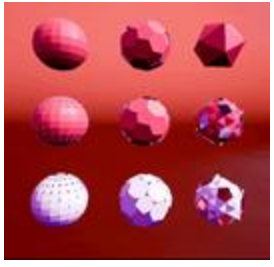
Shaders

Linguagens para shaders mais populares são:

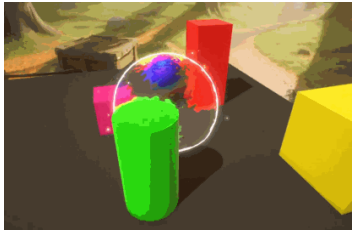
- OpenGL Shading Language (GLSL)
- High-Level Shading Language (HLSL)
- WebGPU Shading Language (WGSL)

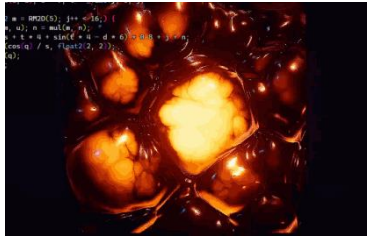
Tipos de shaders

- **Vertex**: Transforma vértices dos objetos, gerando transformações e efeitos visuais.

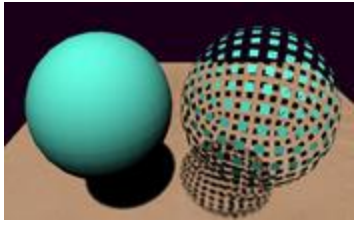


- **Fragment/Pixel**: Retorna a cor de cada pixel na tela com base nas informações recebidas do vertex shader ou de variáveis personalizadas.

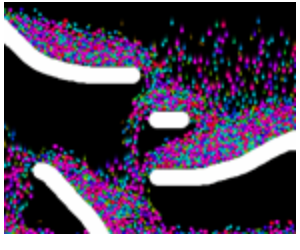


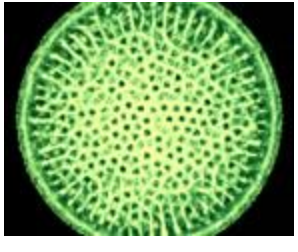


- **Geometry**: Opera sobre primitivas geométricas (pontos, linhas, triângulos) adicionando ou removendo vértices, gerando novas primitivas e realizando outras operações que modificam a geometria.

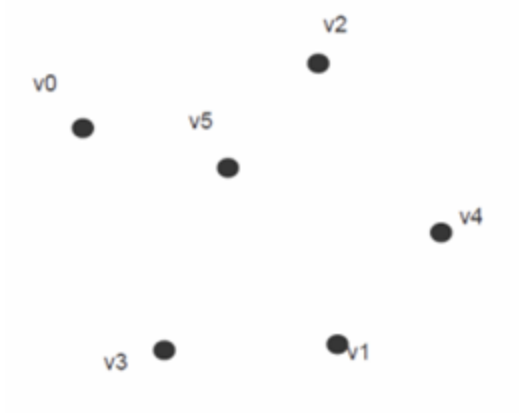


- **Compute**: Vamos ver em Raytracing & Raymarching

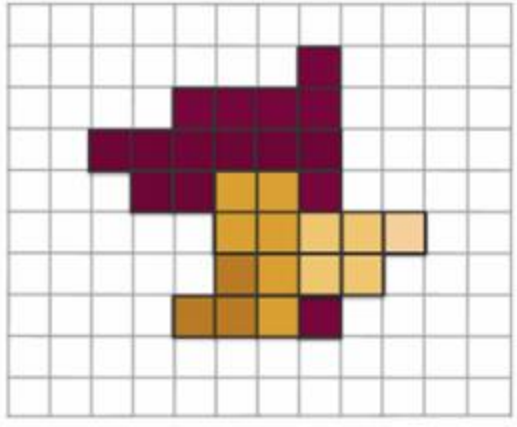




Como os Shaders são invocados



Vertex Shader
invocado 6 vezes



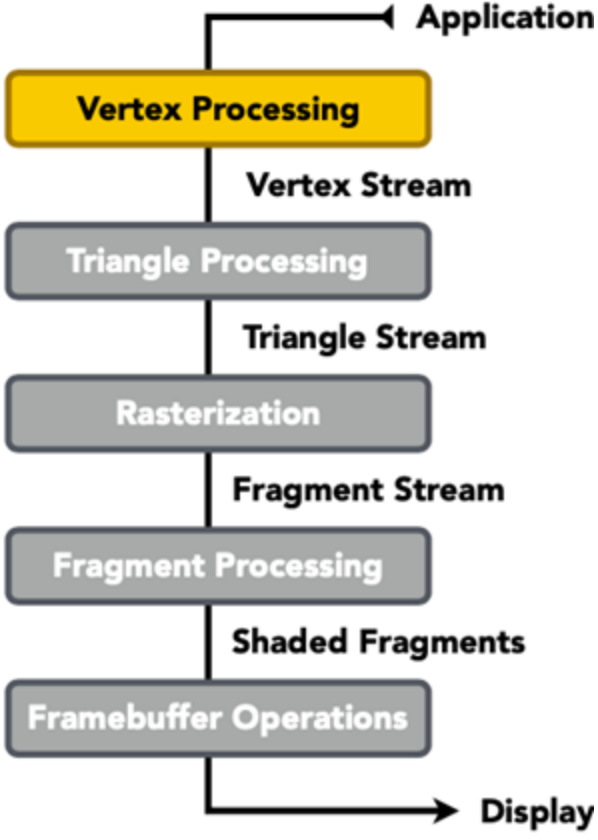
Fragment Shader
invocado 35 vezes

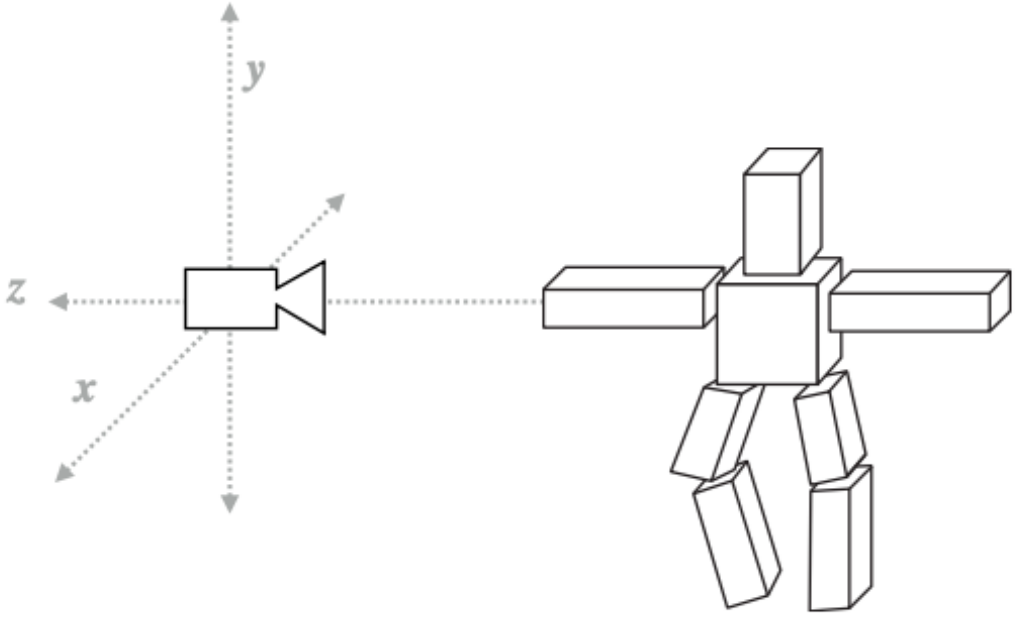
(para os fragmentos
ocultos também)

fonte: https://eclass.aueb.gr/modules/document/file.php/INF143

Insper

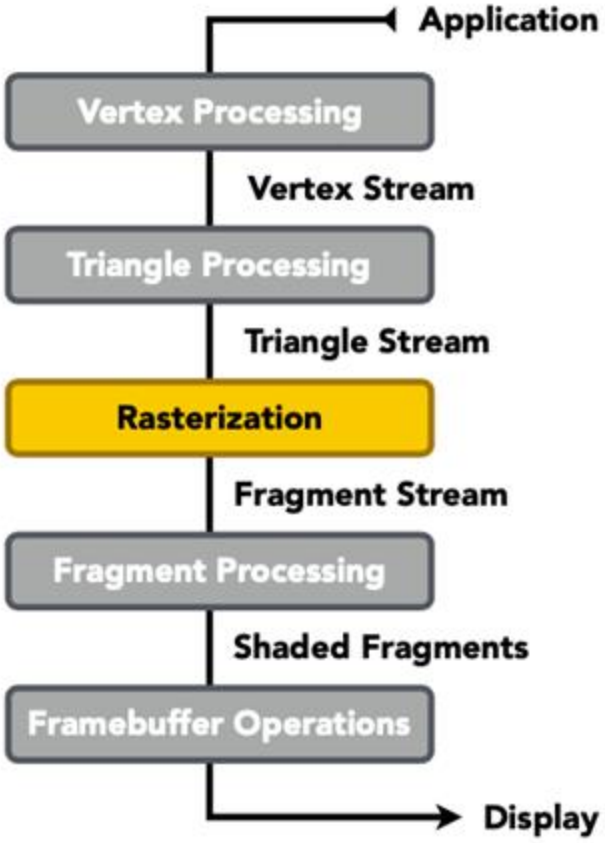
Transformações Geométricas e Visualização

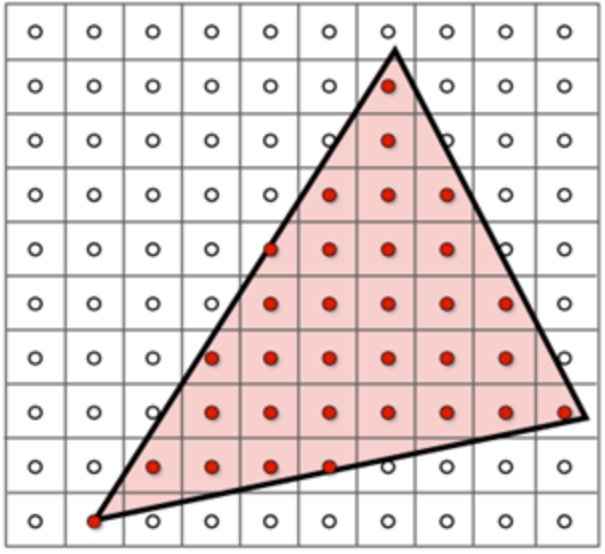




Insper

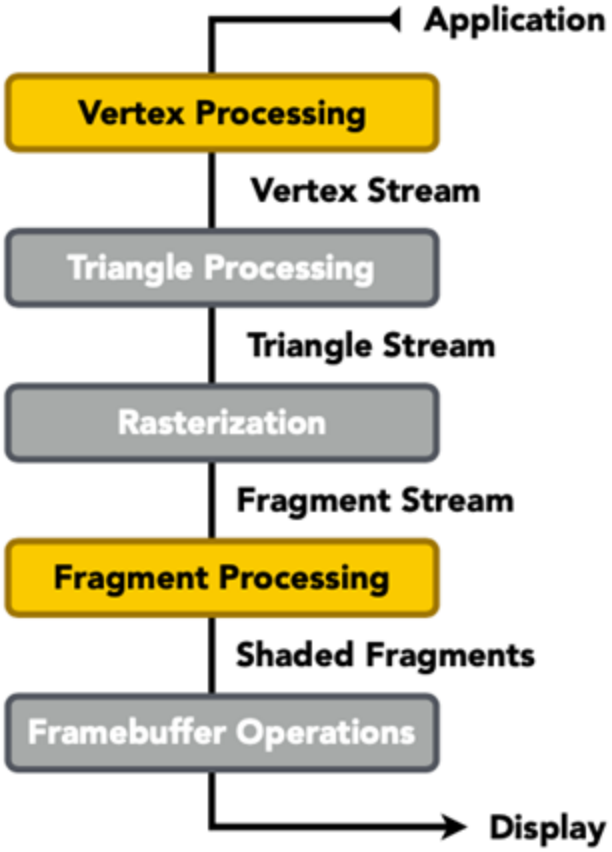
Amostrando os Triângulo

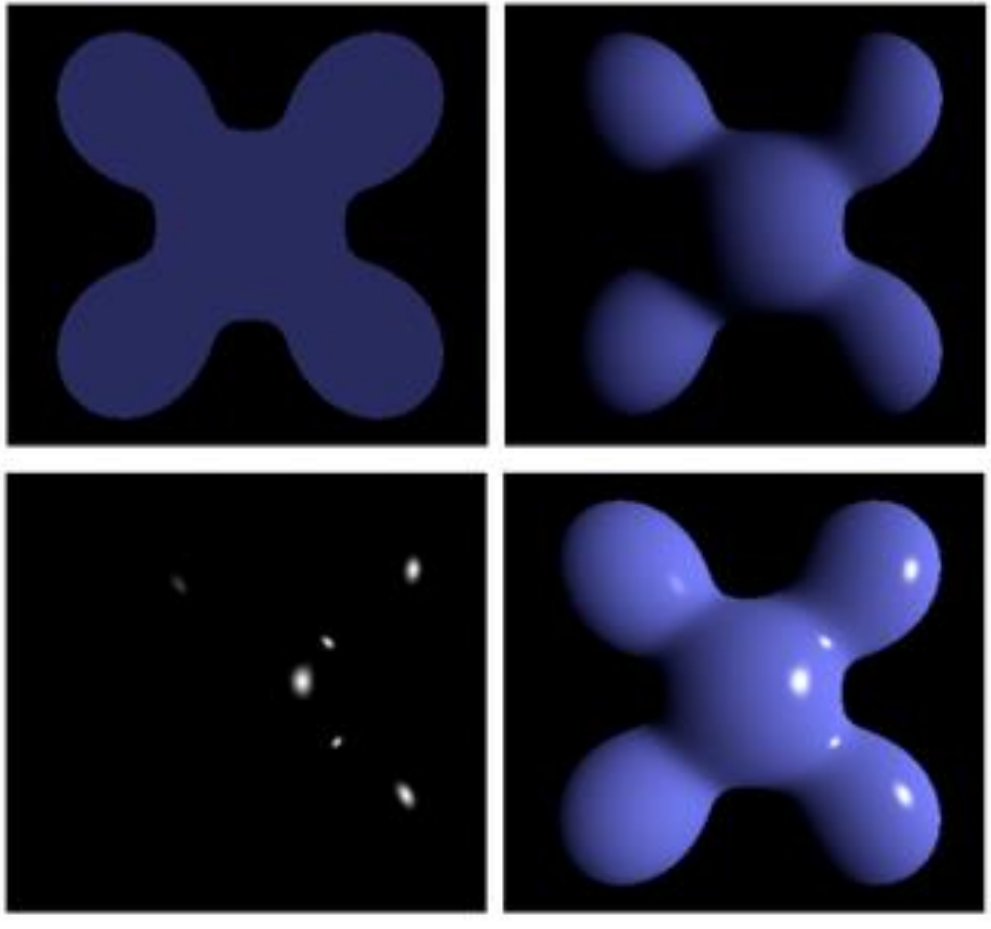




Insper

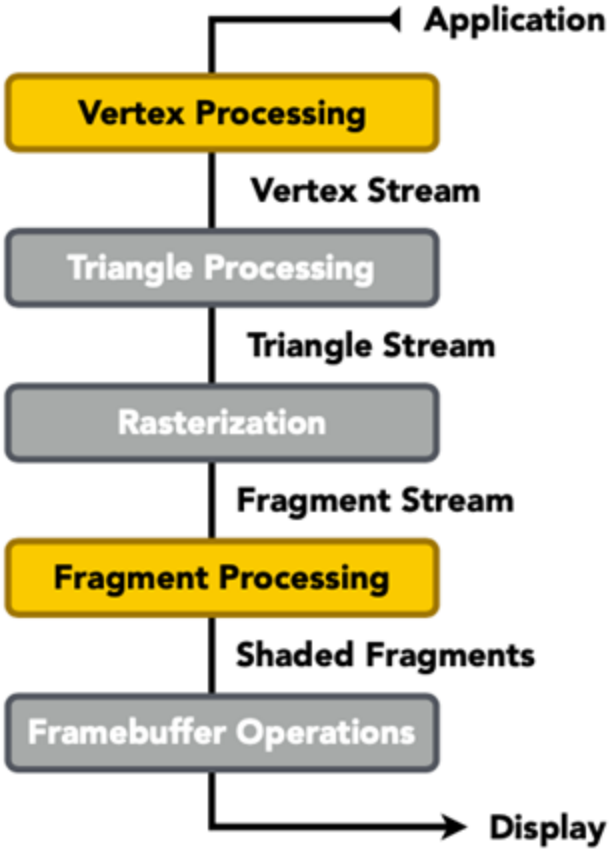
Avaliando a Função de Shading





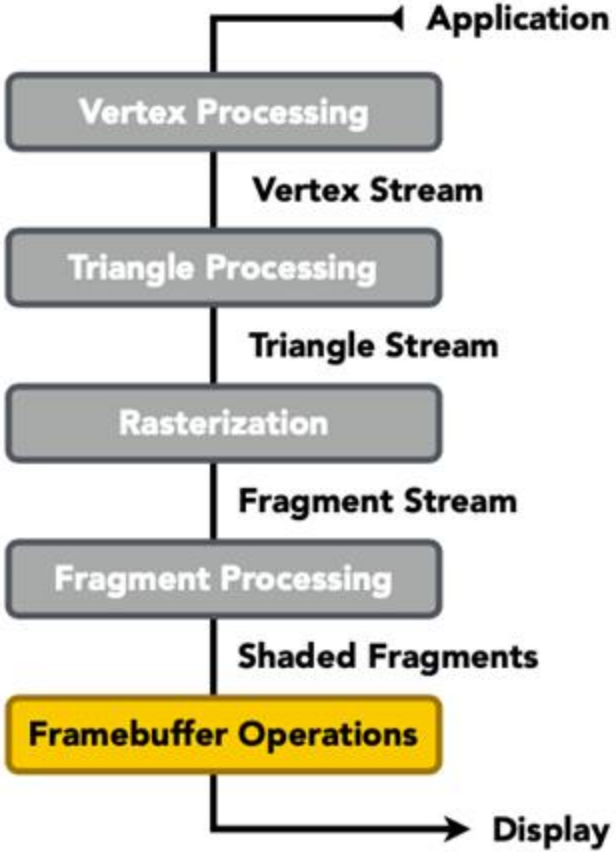
Insper

Mapeamento de Texturas



Insper

Teste de Visibilidade do Z-Buffer



Insper

Objetivo: Cenas 3D complexas em tempo real

Centenas de milhares a milhões de triângulos em uma cena
Cálculos complexos de vértices e fragmentos nos shaders
Alta resolução (2-4 megapixels + supersampling)
30-60 quadros por segundo (ainda mais alto para VR)

Unrecord - Official Early Gameplay Trailer

Shaders Programáveis

Estágios de processamento de vértice e fragmento do programa
Descrever a operação para um único vértice (ou fragmento)

Exemplo de programa de shader em GLSL

```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader() {
   vec3 kd;
   kd = texture(myTexture, uv);
   kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
   gl_FragColor = vec4(kd, 1.0);
}
```

A função é executada uma vez por fragmento.

Exibe a cor da superfície na posição de amostra da tela do fragmento atual.

Este *shader* executa uma pesquisa de textura para obter a cor do material da superfície no ponto e, em seguida, executa um cálculo de iluminação difusa.

Compilação de um Shader

1 fragmento de entrada não processado

```
sampler mySampler;
Texture2D<float3> myTexture;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTexture.Sample(mySampler, uv);
  kd *= clamp ( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

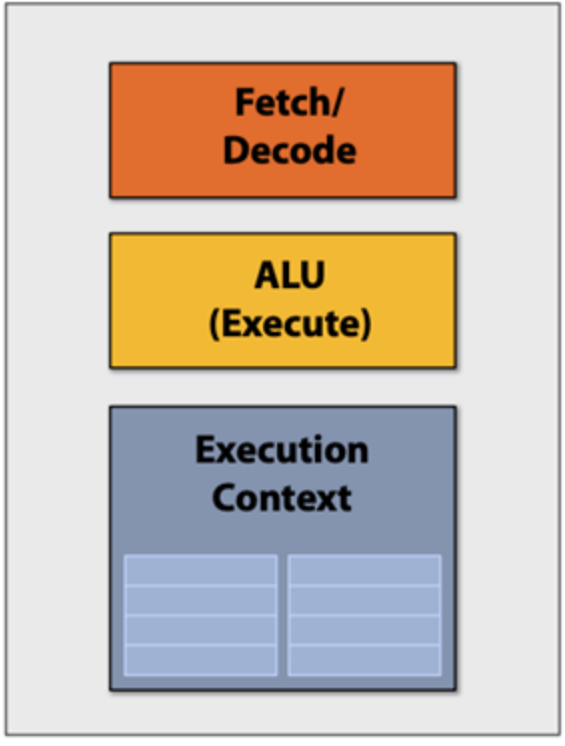
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

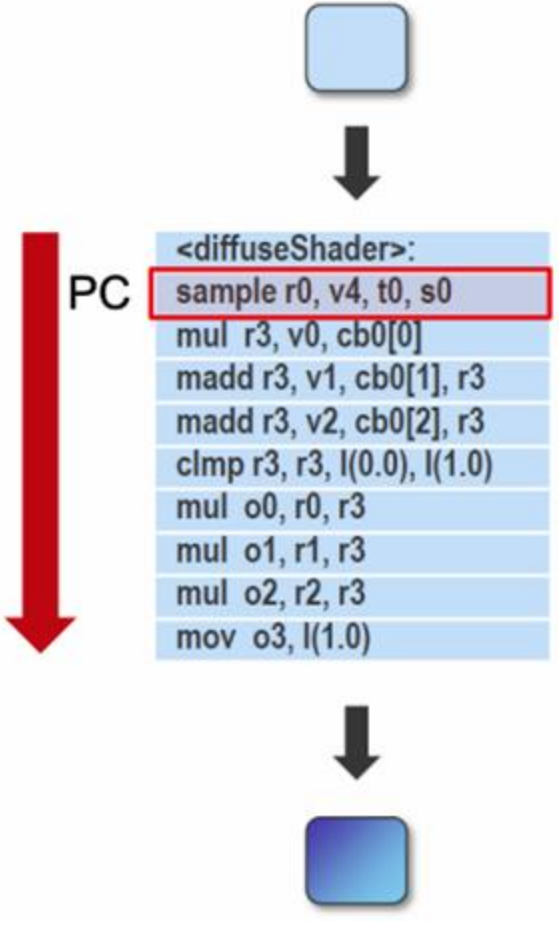
1 fragmento de saída processado

fonte: https://eclass.aueb.gr/modules/document/file.php/INF143

Mapeamento do Shader no HW

Execute o Shader em um único core:

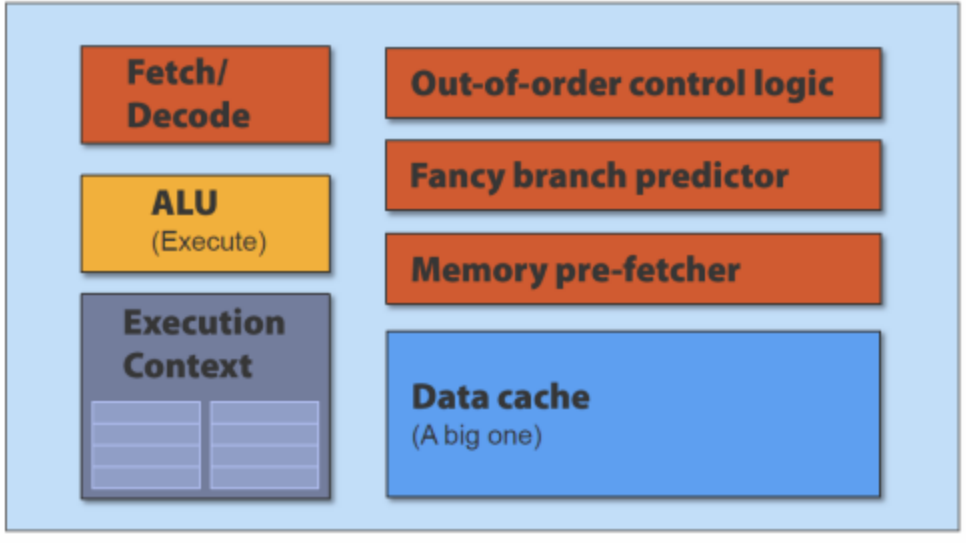




Insper

Mapeamento do Shader no HW

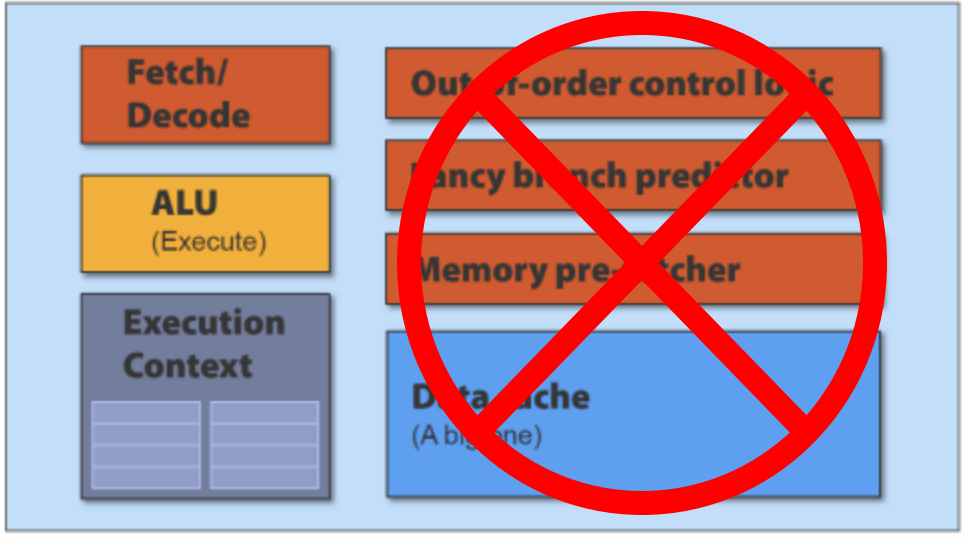
Um *core* de CPU



- Otimizado para acesso de baixa latência aos dados em cache
- Lógica de controle para execução fora de ordem e especulativa
- Grande cache L2

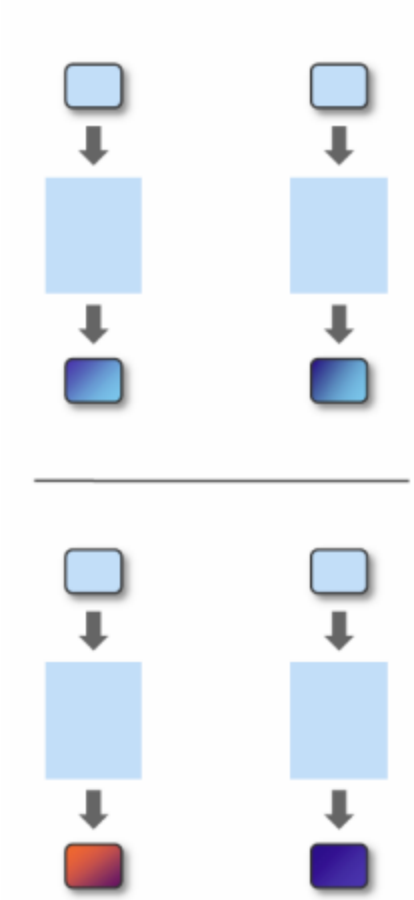
Reduzindo os Cores

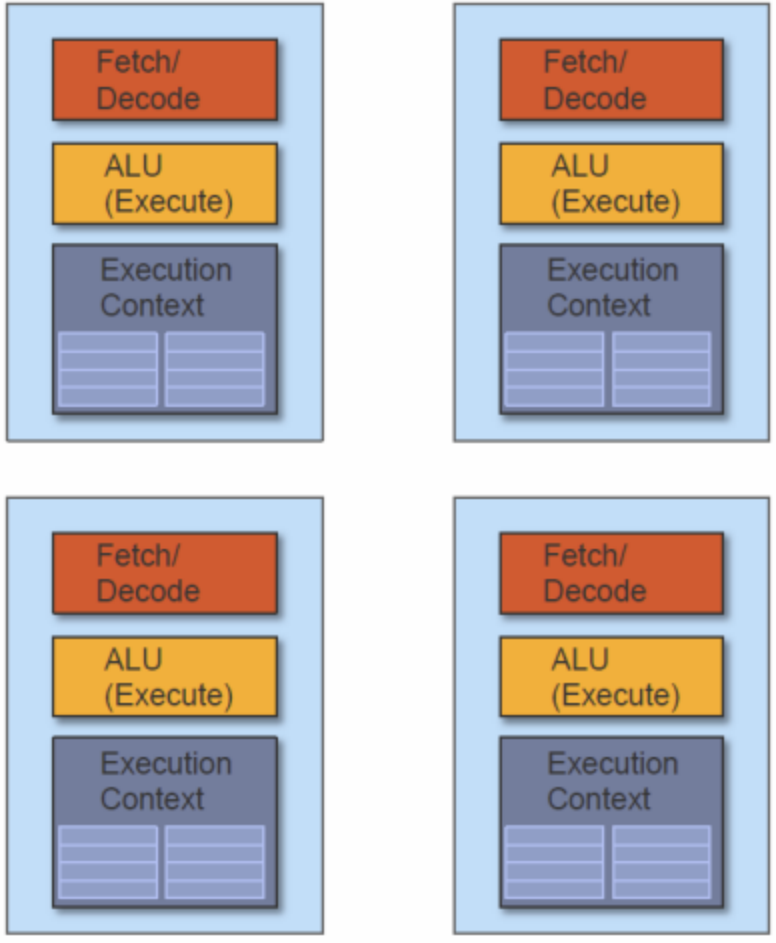
Um *core* de GPU



- Otimizado para computação paralela de dados
- Arquitetura tolerante à latência de memória
- Mais cálculos por mais transistores em ALUs
- Redução dos circuitos principais de controle

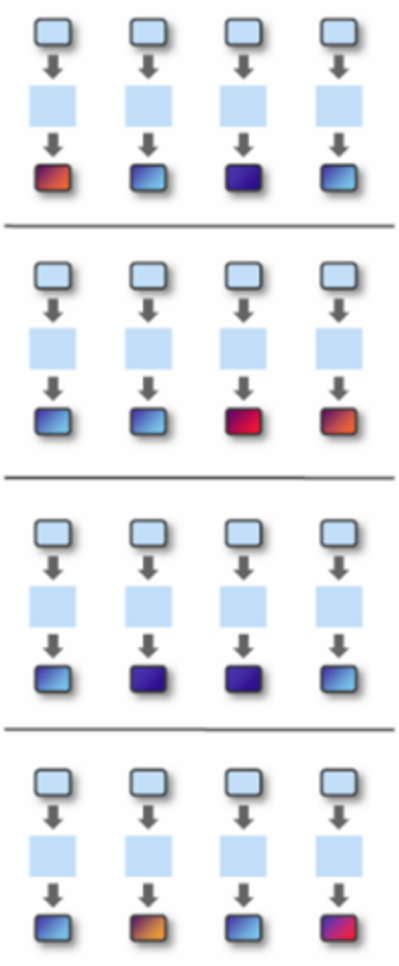
Múltiplas Threads

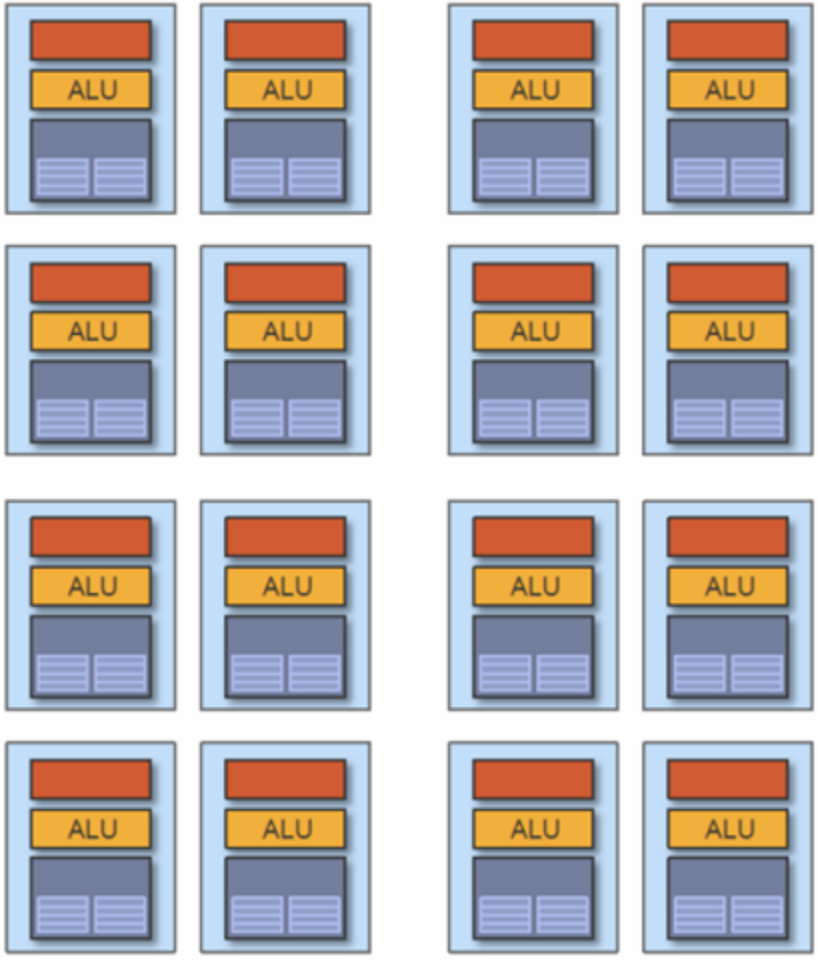




Insper

GPUs com muitos *Cores*

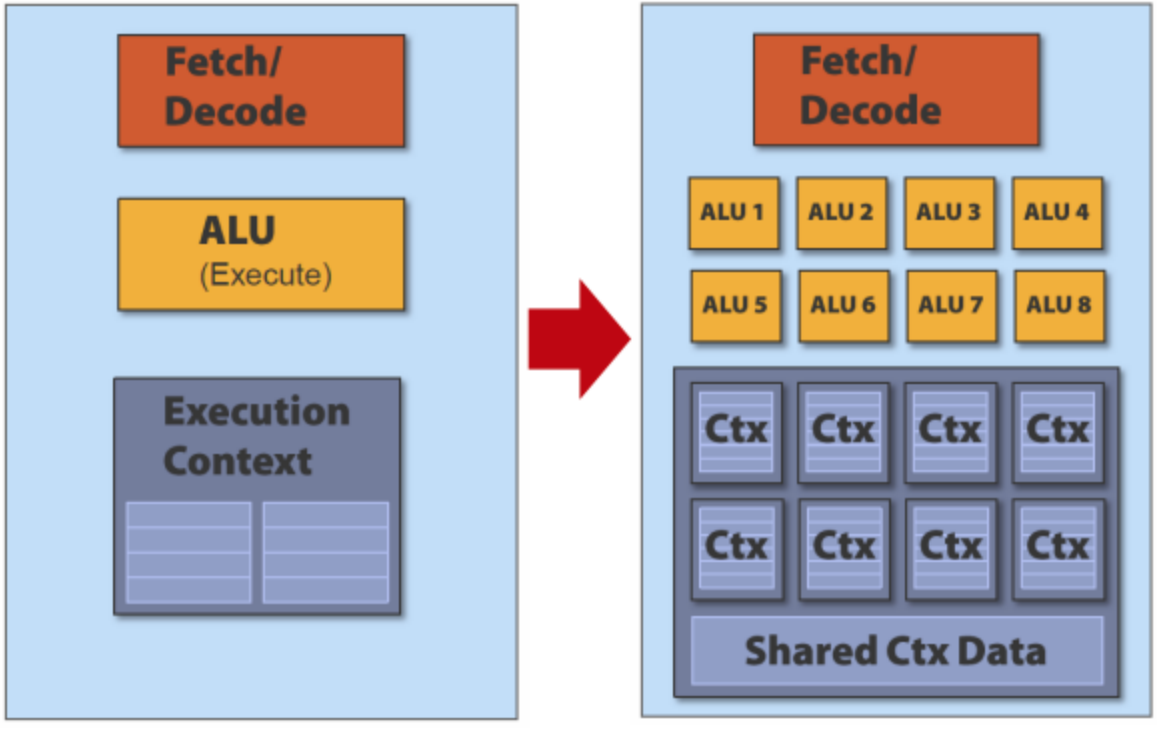




16 cores = 16 fluxos de instruções simultâneos

Múltiplos dados (SIMD)

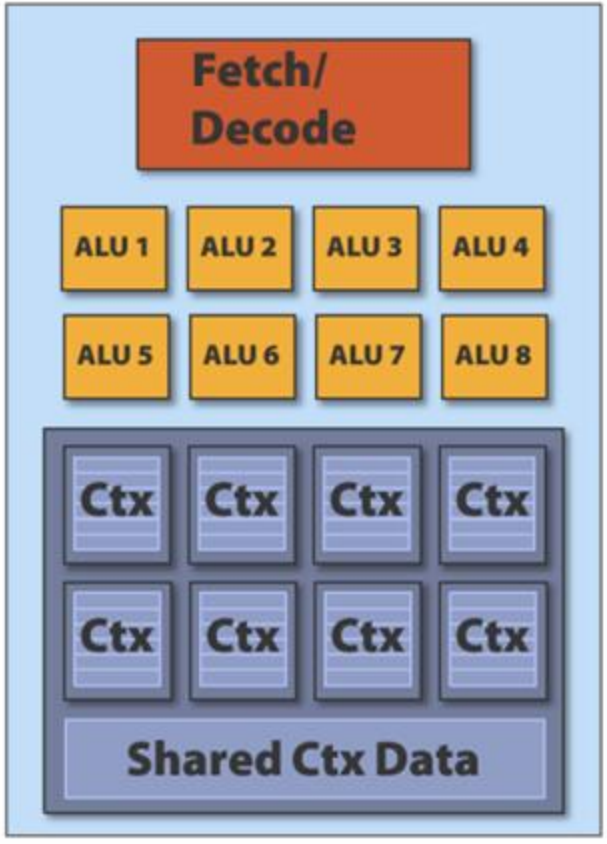
Shaders são inerentemente executados muitas vezes, repetidas vezes em vários registros de seus fluxos de dados de entrada.

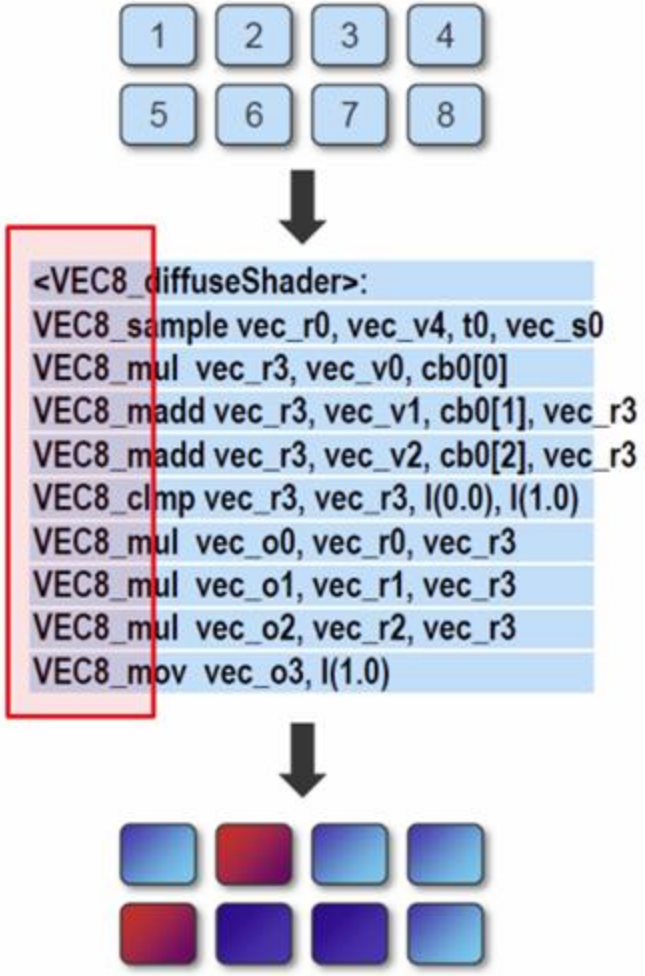


Amortize o custo / complexidade do gerenciamento de instruções para várias ALUs.

Compartilhe a unidade de instrução.

SIMD Cores: Vectorized Instruction Set

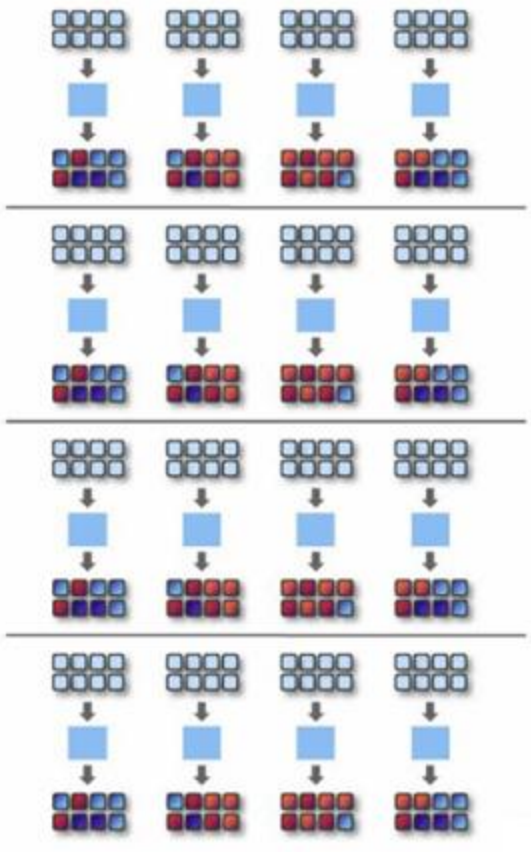


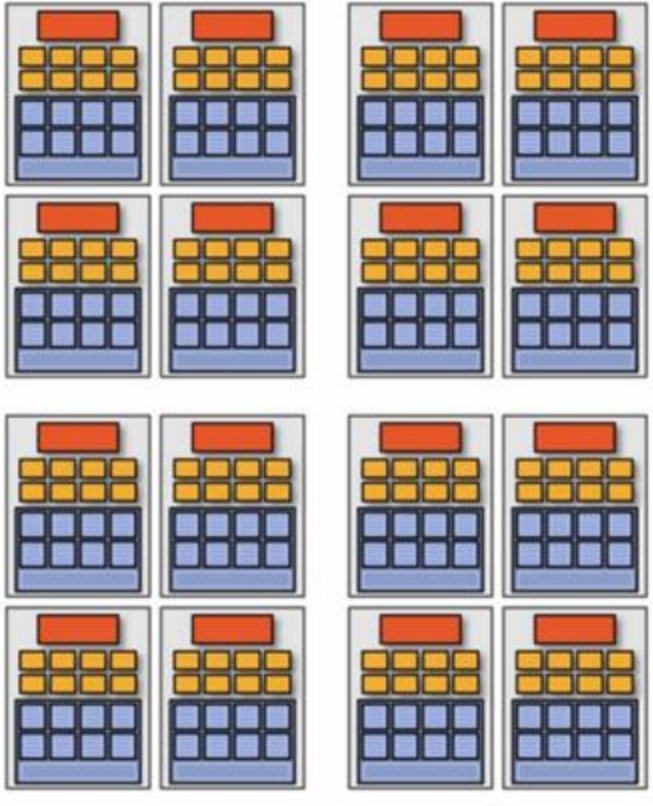


Insper

Adicionando tudo: vários núcleos SIMD

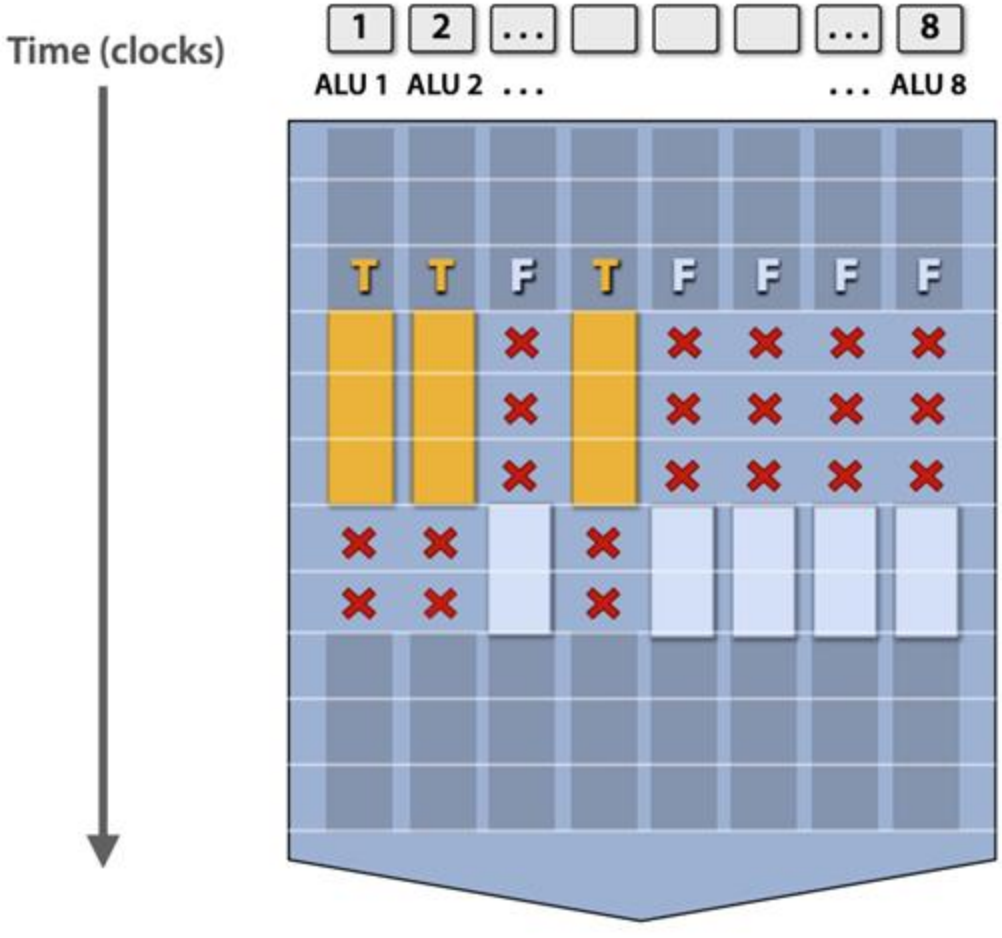
Neste exemplo: 128 dados processados simultaneamente





16 cores = 128 ALUs = 16 fluxos de instruções simultâneos

Condicionais / Branches




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

Introdução a Shaders (GLSL)

Um típico Shader tem a seguinte estrutura:

```
#version numero_da_versão
in type nome_da_variável_de_entrada;
in type nome_da_variável_de_entrada;
out type nome_da_variável_de_saída;

uniform type nome_do_uniform;

void main() { // processa as entrada(s) e faz algo gráfico
    ...
    // pega as coisas processadas e coloca em variáveis de saída
    out_variable_name = weird_stuff_we_processed;
}
```


Referência: https://learnopengl.com/

Uniforms

Uniforms são valores globais que podem ser acessados em qualquer shader do pipeline gráfico. Contudo você tem de declarar ele antes de usar.

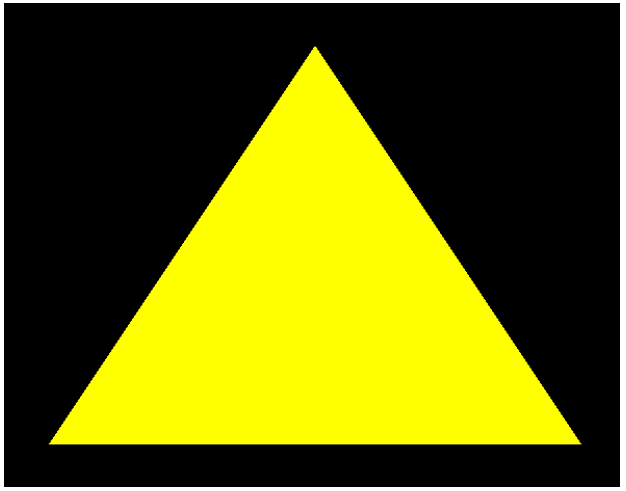
```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;

out vec3 bNormal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    // Invertendo a transformação para normal
    bNormal = mat3(transpose(inverse(model))) * normal;

    // Aplicando transformações em cada vértice
    gl_Position = projection * view * model * vec4(position, 1.0);
}
```

Uniforms

```
...
# Código OpenGL
glUniform3fv(uniforms[”color"], [1.0, 1.0, 0.0])
```

```
#version 330 core
uniform vec3 color;
out vec4 FragColor;
void main() {
    FragColor = vec4(color, 1.0);
}
```

Qual seria o resultado?



In e Outs

Podemos especificar se as variáveis são para a entrada de dados, ou saída de dados. Isso é importante para, por exemplo, passar o valor de um shader no pipeline para outro.

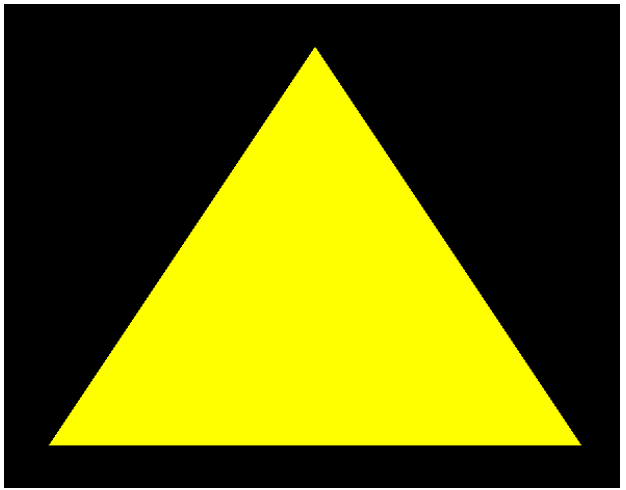
Pode ser usado para receber os dados dos vértices (vertex shader) ou para definir o valor da cor final do pixel (fragmente shader)

```
#version 330 core
in vec4 vertexColor;
out vec4 FragColor;
void main() {
    FragColor = vertexColor;
}
```

In e Outs

```
#version 330 core
layout (location = 0) in vec3 position;
out vec3 bColor;
void main() {
    gl_Position = vec4(position, 1.0);
    bColor = vec3( 1.0, 1.0, 0.0);
}
```

```
#version 330 core
in vec4 bColor;
out vec4 FragColor;
void main() {
    FragColor = bColor;
}
```

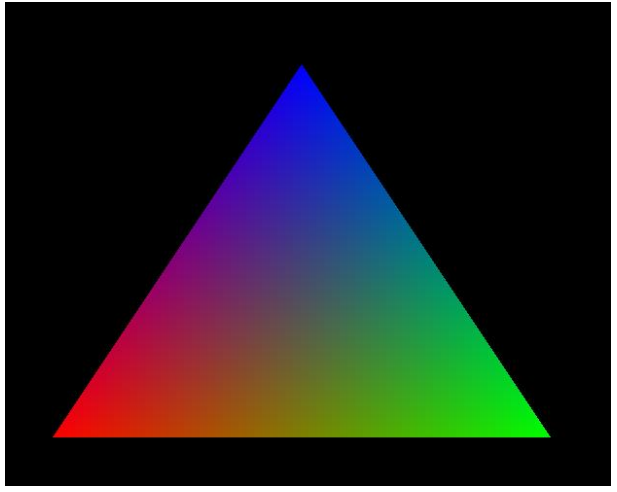


Qual seria o resultado?

In e Outs

```
#version 330 core
layout (location = 0) in vec3 position;
out vec3 bColor;
void main() {
    gl_Position = vec4(position, 1.0);
    if(gl_VertexID == 0) bColor = vec3(1.0, 0.0, 0.0);
    if(gl_VertexID == 1) bColor = vec3(0.0, 1.0, 0.0);
    if(gl_VertexID == 2) bColor = vec3(0.0, 0.0, 1.0);
}
```

```
#version 330 core
in vec4 bColor;
out vec4 FragColor;
void main() {
    FragColor = bColor;
}
```



Qual seria o resultado?

Algumas das principais funções

Função	Descrição
genType abs(genType α)	Retorna valor absoluto de α, ou seja, α ou -α se α < 0;
genType sign(genType α)	Retorna: -1 para α < 0 0 para α = 0 1 para α > 0
genType floor(genType α)	Retorna um inteiro menor ou igual a α
genType ceil(genType α)	Retorna um inteiro maior ou igual a α
genType mod(genType α, float β) genType mod(genType α, genType β)	Retorna o resto da divisão α por β
genType min(genType α, float β) genType min(genType α, genType β)	Retorna α quando α < β Retorna β quando β < α
genType max(genType α, float β) genType max(genType α, genType β)	Retorna α quando α > β Retorna β quando β > α
genType clamp(genType α, genType β, genType δ)	Retorna: α quando β < α < δ β quando α > β δ quando α > δ
genType mix(genType α, genType β, float δ)	Retorna a interpolação linear de α e β, ou seja, α + δ(β - α)
genType step(float limit, genType α) genType step(genType limit, genType α)	Retorna 0 quando α < limit Retorna 1 quando α >= limit
genType smoothstep(float α0, float α1, genType β) genType smoothstep(genType α0, genType α1, genType β)	Retorna 0 quando β < α0 Retorna 1 quando β > α1; Retorna a interpolação de Hermite quando α0 < β < α1

https://relativity.net.au/gaming/glsl/Built-inCommonFunctions.html

Algumas das principais funções

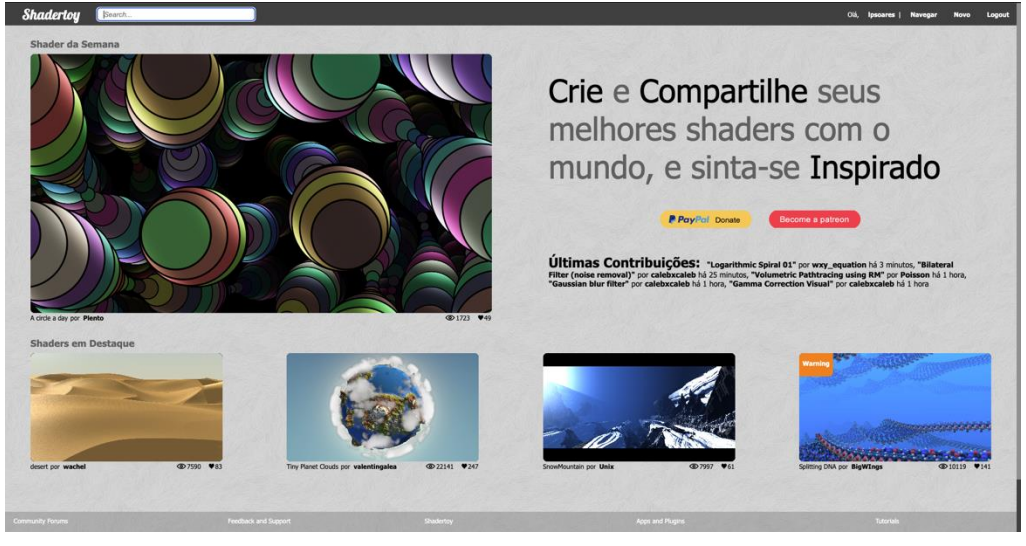
Função	Descrição
genType pow(genType x, genType y)	Retorna valor de x elevado a y
float dot(genType x, genType y)	Retorna o produto escalar dos vetores x e y
vec3 cross(vec3 x, vec3 y)	Retorna o produto vetorial dos vetores x e y
float length(genType x)	Retorna o comprimento (magnitude) do vetor x
genType normalize(genType v);	Retorna o vetor v normalizado

Shadertoy

O Shadertoy é uma ferramenta da internet que permite escrever Fragment Shaders direto no navegador.

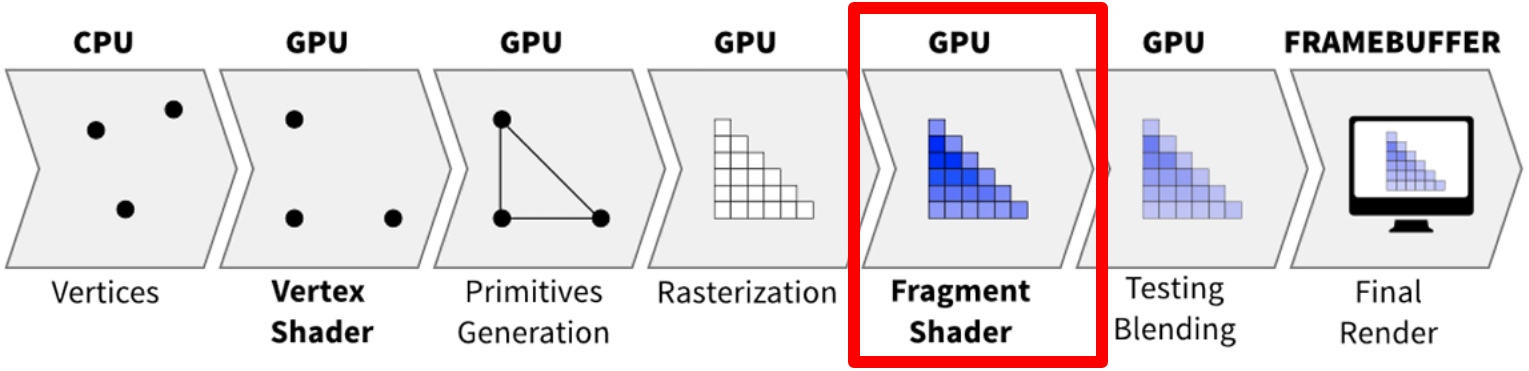
Alguns Uniforms já são automaticamente fornecidos, e todo o processo de compilação é basicamente instantâneo.

O Shadertoy usa alguns padrões para passar os dados, como no caso a chamada do main(), que é mainImage().



Insper

Fragment Shader



O Shadertoy não permite que você escreva vertex shaders e apenas permite que você escreva fragment shaders. Essencialmente, ele fornece um ambiente para experimentar e desenvolver no fragmento shader, tirando todo o proveito do paralelismo de pixels na tela.

Imagem:F.Andreussi(BUW)

Live Coding

Imagem:F.Andreussi(BUW)

Live coding

Introdução: https://www.shadertoy.com/view/MXfBW4

Círculo: https://www.shadertoy.com/view/l3fBWr

Insper

Computação Gráfica

Luciano Soares
<lpsoares@insper.edu.br>

Fabio Orfali
<fabioo1@insper.edu.br>

Gustavo Braga
<gustavobb1@insper.edu.br>