Insper

Computação Gráfica

Aula 20: Programmable Shaders

Ray Casting, Ray Marching, Ray Tracing

- **Ray Casting**: Uma forma simples de lançamento de raios. Essa técnica define o um único raio que detecta interseções com objetos. Essa técnica é usada muitas vezes para interações de raios com objetos.
- **Ray Marching**: Um método de projeção de raios que usa SDFs (Signed Distance Fields) e rastreia objetos marchando em direção a eles.
- **Ray Tracing**: uma versão mais sofisticada de ray casting que dispara vários raios, calcula interseções e e cria recursivamente novos raios a cada reflexão.
    - **Path Tracing**: um tipo de algoritmo de rastreamento de raios que dispara diversos raios por pixel em vez de apenas um. Os raios são disparados em direções aleatórias usando o método Monte Carlo.

Shaders

Linguagens para shaders mais populares são:

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

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/

Tipos de Dados (GLSL)

Os tipos básicos de variáveis são:

`int, float, double, uint e bool`

Vetores podem ter 2, 3 ou 4 compomentes:

`vecn: o vetor padrão com n floats.`

`bvecn: um vetor com n booleanos.`

`ivecn: um vetor com n inteiros.`

`uvecn: um vetor com n inteiros sem sinal.`

`dvecn: um vetor com n doubles.`

Você consegue acessar os valores dos vetores com as extensões: .x .y .z .w, ou também com rgba, ou stqp

Referência: https://learnopengl.com/

Exemplo com Vetores

Um recurso interessante é o **swizzling**, onde você pode combinar e misturar os valores do vetor, por exemplo:

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Na construção de vetores, várias formas de combinação também são viáveis, por exemplo:

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```


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.

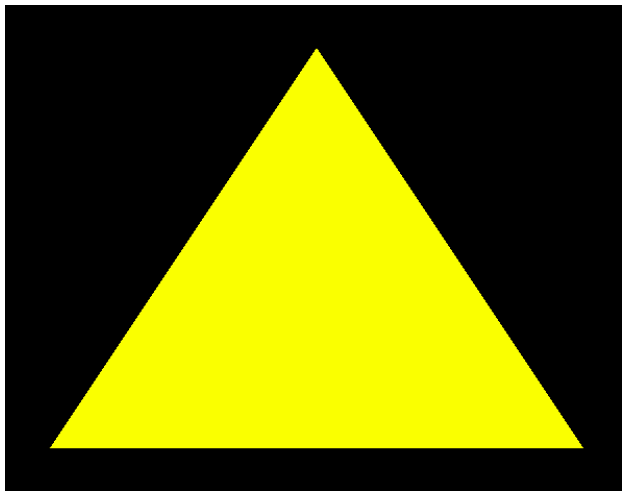
```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;

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 trasnsformações 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 pode exemplo passar o valor de um shader no pipeline para outro.

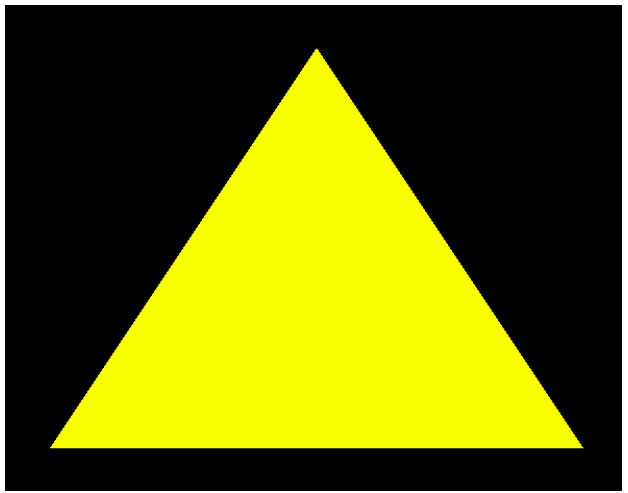
Pode ser usado para receber os dados dos vértices (vertex shader) ou para definir o valor da cor final do pixel (fragmente shader)

```
#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;
}
```

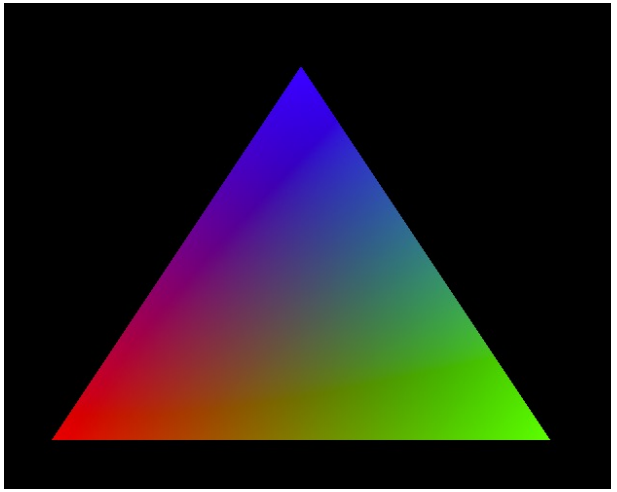


Qual seria o resultado?

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?

Variáveis Built-in

Vertex Shader:

Saída:

- gl_Position (vec4)
- gl_PointSize (int)

Fragment Shader:

Entrada:

- gl_FragCoord (vec4)
- gl_FrontFacing (bool)
- gl_PointCoord (vec2)

Saída:

- gl_FragColor (vec4)

Funções Built-in

O GLSL possui vária funções nativas, a maioria delas funciona para um valor escalar ou um vetor.

Em geral as funções não funcionam para inteiros ou booleanos, assim pode ser necessário converter os valor.

Por exemplo, a função seno funciona com vários tipos:

Name

sin — return the sine of the parameter

Declaration

```
genType sin( genType angle);
```

Parameters

angle

Specify the quantity, in radians, of which to return the sine.

Description

sin returns the trigonometric sine of *angle*.

https://registry.khronos.org/OpenGL-Refpages/gl4/html/sin.xhtml

genType indica que o tipo de dados pode ser float, vec2, vec3 ou vec4.

Algumas das principais funções

Função	Descrição
genType abs(genType α)	Retorna valor absoluto de α, ou seja, -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

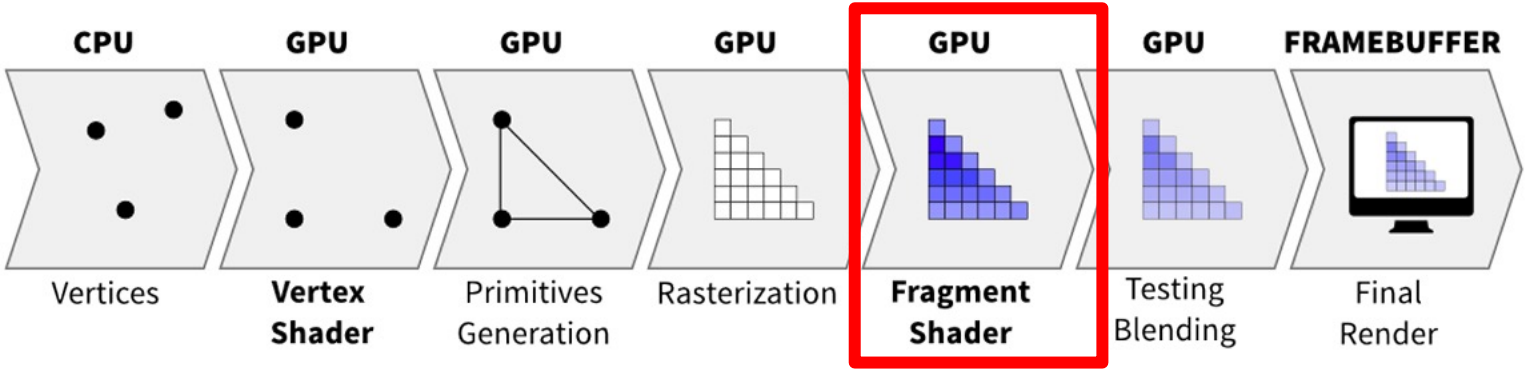
O Shadertoy é uma ferramenta da internet que permite escrever Fragment Shaders direto no navegador.

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)

Shadertoy Uniforms

uniform vec3 iResolution; // Resolução da Janela
uniform float iTime; // Float dos segundos passados
uniform float iTimeDelta;
uniform float iFrame;
uniform float iChannelTime[4];
uniform vec4 iMouse;
uniform vec4 iDate;
uniform float iSampleRate;
uniform vec3 iChannelResolution[4];
uniform samplerXX iChanneli;

Shadertoy

Exemplo quando se cria um novo Shader:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Time varying pixel color
    vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

    // Output to screen
    fragColor = vec4(col,1.0);
}
```

O que isso faz?

Interpreta tipo de vetores

Quando o shader tem um tipo de chamada como a seguinte:

```
uv = fragCoord/iResolution.xy
```

Ele executa as divisões individualmente internamente:

```
uv.x = fragCoord.x/iResolution.x
uv.y = fragCoord.y/iResolution.y
```

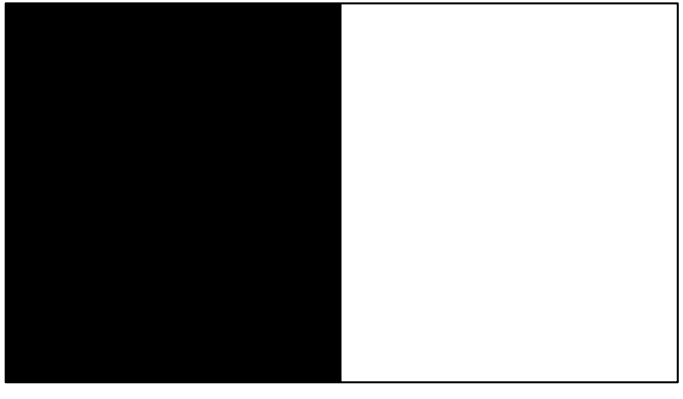
Função step() do GLSL

```
genType step(genType edge, genType x);
```

Essa função retorna 0 se o valor x for menor que edge, senão retorna 1.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord ){
    vec2 uv = fragCoord/iResolution.xy;
    vec3 col = vec3(step(0.5, uv.x));
    fragColor = vec4(col,1.0);
}
```

O que faz esse código?



Referências

Baseado:
https://www.shadertoy.com/

Usando:
https://inspirnathan.com/posts/49-shadertoy-tutorial-part-3

Documentações:
https://iquilezles.org/
https://thebookofshaders.com/

Insper

Computação Gráfica

Luciano Soares
<lpsoares@insper.edu.br>