Visual Computing: Geometry Shader

***Abstract*—Este relatório explora o papel dos Geometry Shaders no campo da computação visual, destacando sua contribuição para a evolução da computação gráfica. Apresenta um histórico dos shaders, com foco nos Geometry Shaders e suas capacidades de manipulação geométrica em tempo real. Através de uma implementação prática em OpenGL, demonstra-se como os Geometry Shaders podem ser aplicados na renderização de gráficos 3D, melhorando significativamente a qualidade e a complexidade dos ambientes virtuais. Além disso, discute as aplicações práticas, desafios e limitações desses shaders.**

***Index Terms*—Geometry Shaders, Sharders, Pipeline Gráfica, Manipulação Geométrica**

I. Introdução

A História da computação gráfica é marcada por avanços significativos que transformaram a maneira como idealizamos e criamos ambientes virtuais. Um marco importante nessa evolução foi a introdução do conceito de ***shaders***, que desempenham um papel crucial na renderização de imagens computacionais. Foi em maio de 1988, durante a apresentação pela *Pixar* da versão *3.0* da sua *interface RenderMan*, que este novo conceito apareceu, marcando consigo o inicio de uma nova era na renderização de imagens. Os *shaders* permitiram a criação de representações visuais mais complexas e realistas, superando as limitações do antigo pipeline de renderização.

Antes disso, a computação gráfica estava limitada a representações simplificadas de objetos e de cores planas. No entanto, com o avanço das unidades de processamento gráfico(*GPU*), o cenário mudou drasticamente. A introdução dos *shaders* inaugurou uma revolução visual. Com eles, veio a capacidade de exercer um controlo mais refinado sobre o processo de renderização, permitindo a criação de imagens com um nível de detalhe e realismo anteriormente inatingíveis. À medida que as GPUs evoluíram em potência e eficiência, o mesmo aconteceu com os shaders. Transitamos de simples funções de manipulação de luz e cor para formas mais complexas e sofisticadas, como os *Vertex Shaders* e os *Fragment Shaders*.

Mais tarde, em 2006, com o lançamento do *Shader Model 4*[1] introduzido no *Direct3D 10* e *OpenGL 3.2*, surgiu um novo tipo de *shader* - o ***Geometry Shader***, que desempenhou um papel crucial na expansão das capacidades de renderização, permitindo a manipulação geométrica em tempo real e abrindo novas possibilidades na criação de ambientes virtuais.

Neste artigo exploramos o funcionamento e a utilidade dos *geometry shader*, com recurso uma pequena implementação em OpenGL.

II. Shaders

Os *Shaders* são programas pequenos, porém robustos, que operam na *GPU* e desempenham um papel fundamental na renderização de gráficos 3D em tempo real.

Antes da sua introdução, a computação gráfica era dominada pelo *fixed-function pipeline*, um sistema com capacidades limitadas, onde os programadores estavam restringidos a um conjunto limitado de funções predefinidas, como a simples adição de texturas. No entanto, com a transição para o *programmable pipeline*, os *shaders* tornaram-se uma ferramenta poderosa, oferecendo aos programadores um controlo completo do comportamento de determinadas partes do processo de renderização.

O desenvolvimento de *shaders* é realizado em linguagens de programação de alto nível como *GLSL* (OpenGL Shading Language) ou *HLSL* (High-Level Shading Language), que permitem aos programadores instruir a GPU a executar uma gama quase ilimitada de efeitos gráficos, desde o cálculo da luz e da cor até à manipulação de texturas e transformações geométricas.

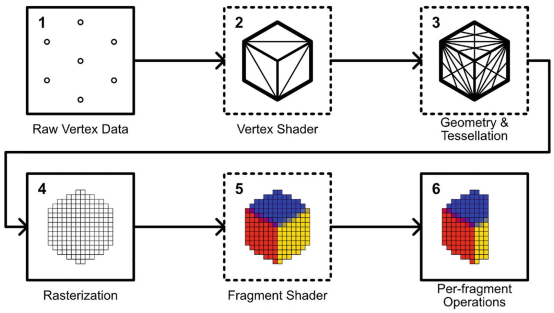

Fig. 1. [6]As etapas do pipeline gráfico. Os estágios rodeados por uma linha tracejada indicam partes do pipeline que podemos personalizar usando *shaders*.

Pela visualização da figura 1, é possível verificar diferentes tipos de *shaders*, cada um com um propósito específico na pipeline gráfica.

- ***Vertex Shaders***: Processa cada vértice individualmente, determinando as propriedades dos vértices, como posição e cor.
- ***Geometry Shaders***: Oferecem um controle sobre a geometria, permitindo a criação e manipulação das formas em tempo real.
- ***Fragment Shaders (ou Pixel Shaders)***: ocupam-se com os detalhes de pixel, como cor e profundidades.

III. Geometry Shaders

Os ***Geometry shaders*** são uma parte importante do pipeline de renderização gráfica, embora opcional, sendo executados antes do *primitive assembly* e do *Fragment Shader*. Este tipo de shaders recebe entradas primitivas completas, tais como coleções de vértices, representadas como *arrays*. Na maioria do casos estas entradas são fornecidas pelo *Vertex Shader*, no entanto, quando o pipeline contém *tessellation*, a entrada será fornecida pelo *tessellation evaluation shader*.

Ao contrário dos *Vertex Shaders*, que processam vértices individualmente, os *Geometry Shaders* têm o poder de processar conjuntos inteiros de vértices, permitindo a formação de novas primitivas e abrindo um leque de possibilidades para a criação de geometrias complexas e efeitos dinâmicos, como por exemplo, a geração de um grupo de partículas a partir de um único ponto. Esta flexibilidade torna-os ferramentas indispensáveis para a amplificação de objetos e para a criação de efeitos de partículas sofisticados, como pelos ou relva. Outro aspeto notável dos *Geometry Shaders* é a sua capacidade de produzir, como output, tipos primitivos diferentes daqueles que foram recebidos por input. Por exemplo, pontos podem ser usados como parâmetros de entradas para gerar triângulos.

IV. Implementação Geometry Sharders em OpenGL

Nesta secção, iremos aplicar os conceitos e capacidades dos *Geometry Shaders* que foram abordados anteriormente. Para isso, vamos desenvolver uma implementação básica de um *Geometry Shader*, com recurso ao OpenGL, com o objetivo de demostrar de forma prática o funcionamento dos *Geometry Shaders* na pipeline de renderização gráfica.

A. Configurações do Ambiente

Para facilitar o processo de configuração, fornecemos um código-base completo e funcional no *GitHub*.

O código-fonte e as instruções de configuração podem ser encontradas no nosso repositório, GitHub - Implementação de Geometry Shader com OpenGL. Para configurar o ambiente de desenvolvimento, siga as instruções dadas no repositório.

B. Criar um Geometry Shader

O Geometry Shader é uma fase opcional no OpenGL, como visto anteriormente, ele é capaz de alterar o tipo e o número de primitivas que passam pelo pipeline. No entanto, antes deles serem vinculados, deve ser especificado o tipo primitivo de entrada e saída, tal como, o número máximo de vértices que ele pode produzir. De tal forma, analisaremos o seguinte exemplo básico de um geometry shader.

Listing 1. Geometry Shader Basic

```
#version 150 core
layout(points) in;
layout(line_strip, max_vertices = 2) out;
void main()
{
    gl_Position = gl_in[0].gl_Position +
vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();
    gl_Position = gl_in[0].gl_Position +
vec4(0.1, 0.0, 0.0, 0.0);
    EmitVertex();
    EndPrimitive();
}
```

Este *shader* simplesmente recebe um ponto e gera uma linha como saída. No entanto, não te preocupes em entender como isto funciona por agora, foca em perceber os recursos

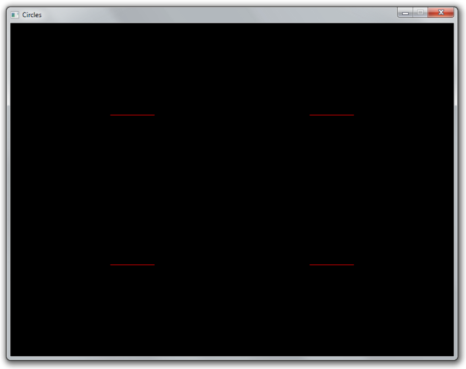

Fig. 2. Resultado da implementação do Geometry Shader Basic

deste exemplo. Primeiro, no topo do *shader*, temos um par de qualificadores de *layout* contendo a declaração dos tipos primitivos de entrada e saída e o número máximo de vértices que serão produzidos. Em anexo, na tabela I, temos ainda uma listagem dos tipos primitivos aceites como entrada pelo geometry shader e os tipos primitivos correspondentes que podem ser usados em comandos de desenho.

Ainda neste trecho de código, podemos analisar funções especiais integradas do GLSL, ***EmitVertex()*** e o ***EndPrimitive()***.

1) EmitVertex(): gera um novo vértice como saída do geometry shader, ou seja, cada vez que a função é chamada um novo vértice é adicionado à primitiva atual, isto caso o tipo primitivo de saída seja *line_strip* ou *triangle_strip*. No caso do tipo primitivo de saída ser *points*, ela apenas se limita a cria um novo ponto independente.

2) EndPrimitive(): indica o fim de uma primitiva atual e sinaliza ao OpenGl que, na próxima vez que *EmintVertex*() for invocada, recomeça uma nova primitiva. Ainda, é importante realçar que quando um geometry shader termina, a primitiva atual termina implicitamente, sem necessidade de chamar esta função. No entanto, quaisquer primitivos incompletos serão descartados. Por exemplo, se o shader produzir uma faixa triangular com apenas dois vértices ou se produzir uma faixa linear com apenas um vértice os vértices extras que compõem a faixa parcial serão descartados.

C. Implementação de Geometry Shaders[8]

Depois de aprender como os geometry shaders funcionam, a funcionalidade do geometry shader, apresentado anteriormente, passa a ser mais intuitiva. Para visualizar a renderização, podemos executar o ficheiro *basic_geometry_shader.py*, obtendo o mesmo resultado que a figura 2.

Agora, vamos tentar criar algo um pouco diferente e tentar gerar retângulos usando *triangle_strip*. Em seguida, vamos tentar criar algum um pouco mais complicado, como criar círculos de diferentes cores e com qualquer quantidade de lados, adicionando pontos. Começa por variar as cores das linhas que estão sendo desenhadas, permitindo que cada uma tenha uma cor única. Para isso segue os seguintes passos:

1) No Vertex Shader, adiciona um atributo de cor de entrada e passa essa cor para a saída, que será recebida pelo

Geometry Shader. No Geometry Shader, recebe a cor do Vertex Shader e passa-a para o Fragment Shader. Por exemplo:

```
in vec3 vColor[];
void main(){
    fcolor = vColor[0]
 ...
}
```

Após adicionar a variação de cores, o próximo passo é criar círculos com qualquer quantidade de lados, utilizando Geometry Shaders. Para tal,

2) Modifica o Geometry Shader para gerar círculos a partir de pontos. Cada ponto será o centro de um círculo. Vais precisar de definir o número de lados (ou segmentos) para o aproximar de um círculo. Varia o número de segmentos. Veja um exemplo simplificado:

```
#version 150 core

layout(points) in;
layout(line_strip,
    max_vertices = 11) out;

in vec3 vColor[];
out vec3 fColor;

const float PI = 3.1415926;

void main()
{
fColor = vColor[0];
for (int i = 0; i <= 10; i++) {
    float ang = PI * 2.0 / 10.0 * i;

    vec4 offset = vec4(cos(ang) *
       0.3, -sin(ang) * 0.4, 0.0, 0.0);

    gl_Position = gl_in[0].gl_Position
        + offset;

    EmitVertex();
}
    EndPrimitive();
}
```

3) Atualiza o Vertex Shader para passar apenas a posição do centro do círculo, além da cor:

```
#version 330 core

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 color;

out vec3 vColor;

void main()
{
    vColor = color;
    gl_Position = vec4(position, 1.0);
}
```

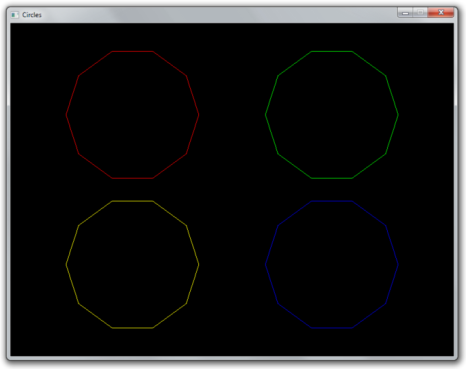

Fig. 3. Resultado do Geometry shader circle

4) Compila e executa o programa. Agora, deverás ver círculos coloridos renderizados no ecrã, cada um com uma cor específica, como na figura 3. Conseguiste perceber o que aconteceu quando varias o número de segmentos do círculo. Torna-se mais suave quanto maior for o número de lados dele.

Caso tenha dificuldade em realizar a implementação, existe uma opção alternativa disponível. Ao aceder a pasta chamada "*solution*" no projeto fornecido e executar a solução já implementada passo a passo. Para isso, basta clicar em cada número sequencial para ver a solução para o próximo passo. O processo começa com apenas quatro pontos no ecrã.

Para uma compreensão mais aprofundada do que está acontecendo em cada passo, pode analisar minuciosamente os arquivos de *shaders* na pasta "*shaders*", que foram utilizados na implementação desta solução. Isso deve ajudar a esclarecer o funcionamento de cada etapa do processo.

Para explorar um pouco mais, tente que as figuras geométricas resultantes sejam diferentes para cada ponto inicial.

V. Aplicações de Geometry shaders

Após aprendizagem básica, é crucial explorar como os *geometry shaders* são aplicados em cenários do mundo real. Seguem-se algumas das aplicações práticas, destacando-se a versatilidade e a eficácia deste tipo de *shaders*, em diferentes contextos, com referências a tutorais e vídeos que pode mais tarde explorar em diferentes softwares.

A. Renderização de Terrenos em 3D

A renderização de Terrenos em 3D é uma das aplicações de *geometry shaders* para a criação de paisagens detalhadas e realistas, ajustando a geometria do terreno em tempo real. Este caso é especialmente útil em jogos e simulações.

No exemplo da Figura 4, temos o exemplo de uma aplicação com recurso a geometry shaders, para a criação da robustez e do detalhes realista de um simples terreno com relva.

B. Efeitos de Partículas

Para a criação de efeitos de partículas complexos, como fumaça, fogo, ou neblina, o recurso ao *geometry shaders*

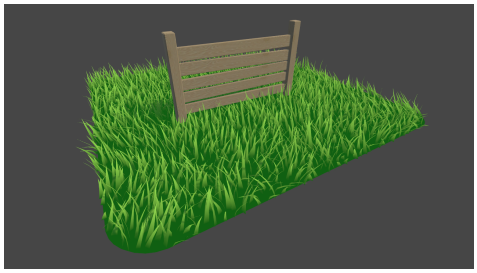

Fig. 4. Aplicação de geometry shader na renderização de Relva em 3D

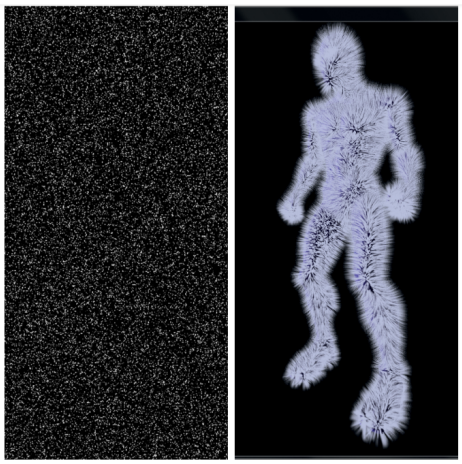

Fig. 5. Textura usada para representar cabelos e saída de um exemplo de renderização de pele

é fundamental. Estes permitem um controlo mais refinado sobre cada partícula, possibilitando a criação de efeitos visuais impressionantes e realistas.

C. Detalhe em Personagens Digitais

Tal como, na renderização de Terrenos em 3D, a adição de detalhes a personagens digitais é extremamente útil, especialmente para cabelos e peles. Com a criação de texturas mais dinâmicas e digitais consegue-se ter um aumento significativo na qualidade visual das personagens.

Na figura 5, tem-se um pequeno exemplo de textura criada para representar cabelos, que ao ser usada num shader adjacente permitiu a renderização de peles de um objeto, neste caso de uma pessoa.

VI. Desafios e Limitações dos Geometry Shaders

Tal como como qualquer tecnologia, os *geometry shaders* também tem as suas próprias limitações e desafios, que devem ser consideradas quando forem usadas.

A. Desafios

1) **Complexidade**: Os geometry shaders acrescentam uma camada adicional de complexidade na programação, comparativamente aos *vertex shader e fragment shader*. Este facto deve-se a necessidade de ter um entendimento mais profundo não apenas da linguagem, mas também da integração na restante pipeline gráfica.
2) **Compatibilidade**: Nem todos os dispositivos de renderização suportam geometry shaders. Portanto, se estivermos a desenvolver uma aplicação que precisa ser executada em uma variedade de dispositivos, pode ser necessário fornecer uma alternativa para dispositivos que não suportam geometry shaders.
3) **Rendimento**[3]: Embora os geometry shaders possam melhorar o desempenho em alguns casos, eles também podem diminuir o desempenho se usados de forma inadequada ou intensiva. Por exemplo, se um geometry shader gerar uma grande quantidade de primitivas, isso pode sobrecarregar o pipeline de renderização e diminuir o desempenho geral.

B. Limitações

1) **Limites de saída**: Seja pelo limite do número máximo de vértices que uma única invocação de um geometry shader pode produzir, ou pelo número total máximo de componentes de saída que uma única invocação de um geometry shader pode produzir. Estes limites estão pendentes pelo hardware e podem variar de acordo com o dispositivo de renderização.

VII. Recursos Adicionais para Aprendizagem

Para aqueles interessados em aprofundar seus conhecimentos em *geoemtry shaders*, tem aqui uma lista de materiais de aprendizagem:

1) Interactive Graphics 17 - Geometry Shaders, um pequeno vídeo com os conceitos teóricos sobre *geometry shaders*, e discusão de implementações.
2) Learn OpenGL - Geometry shaders, um tutorial passo a passo com associação de conteúdos teoricos.
3) Fundamentos do GLSL - Capitulo 5 - Geometry Shaders, um livro com fundamentos base para desenvolver *shaders*, em GSLS, e com implementações mais complexas para explorares.
4) Bifurcation Diagram in OpenGL, implementação de geometry shader em física
5) Na pasta *exploration* do projeto fornecido, podes encontrar uma simulação de um campo estelar com recurso a geometry shader.

VIII. Conclusão

Em suma, os shaders de geometria representam uma junção fascinante de criatividade e inovação tecnológica. Eles não apenas oferecem aos programadores uma ferramenta poderosa para criar experiências visuais impressionantes, mas também demonstram a incrível capacidade da computação gráfica de transformar a maneira como experimentamos o mundo digital.

References

[1] Mike Bailey and Steve Cunningham. *Graphics Shaders, 2nd Edition*. Accessed: 2023-11-23; Online. 2023. URL: https://learning.oreilly.com/library/view/graphics-shaders-2nd/9781439867754/chapter-72.html.

TABLE I
GEOMETRY SHADER PRIMITIVE TYPES AND ACCEPTED DRAWING MODES

Geometry Shader Primitive Type	Accepted Drawing Command Modes
points	GL_POINTS
lines	GL_LINES, GL_LINE_STRIP
triangles	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
lines_adgacency	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY
triangles_adjacency	GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY

[2] Gabriel Soto Bello. *Shaders: Geometry Shader*. Accessed: 2023-11-23. 2011. URL: https://www.tecmundo.com.br/voxel/especiais/182980-shaders-geometry-shader.htm.

[3] NVIDIA Developer. *GPU Gems 3, Chapter 41: Using Geometry Shader for Compact and Variable-Length Output*. Accessed: 2023-11-23. 2023. URL: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-41-using-geometry-shader-compact-and-variable-length.

[4] Khronos Group. *Geometry Shader*. Accessed: 2023-11-23. 2023. URL: https://www.khronos.org/opengl/wiki/Geometry_Shader.

[5] Khronos Group. *Shader*. Accessed: 2023-11-23;Online. 2023. URL: https://www.khronos.org/opengl/wiki/Shader.

[6] Daniel Ilett. *Building Quality Shaders for Unity®: Using Shader Graphs and HLSL Shaders*. Accessed: 2023-11-23; Online. 2023. URL: https://learning.oreilly.com/library/view/building-quality-shaders/9781484286524/html/523243_1_En_1_Chapter.xhtml.

[7] LightningChart. *Introduction to Shaders*. Accessed: 2023-11-23. 2023. URL: https://lightningchart.com/blog/introduction-to-shaders/.

[8] Open.gl. *Geometry*. Accessed: 2023-11-23. 2023. URL: https://open.gl/geometry.

[9] Mark Richards. *OpenGL Programming Guide, Chapter 10: Geometry Shaders*. Accessed: 2023-11-23. 2023. URL: https://learning.oreilly.com/library/view/opengl-programming-guide/9780132748445/ch10.html.

[10] Jacobo Rodríguez. *GLSL Essentials, Chapter 5.3*. Accessed: 2023-11-23. 2023. URL: https://learning.oreilly.com/library/view/glsl-essentials/9781849698009/ch05s03.html.

[11] Wikidot. *Shaders*. Accessed: 2023-11-23; Online. 2023. URL: http://desenvolvimentodejogos.wikidot.com/shaders.

[12] Wikipedia. *Shader*. Online; Accessed: 2023-November-23. 2023. URL: https://en.wikipedia.org/wiki/Shader.

[13] YouTube. *Title of the Video*. Accessed: 2023-11-23. 2023. URL: https://www.youtube.com/watch?v=C8FK9Xn1gUM.