Visual Computing: Geometry Shader

Mariana Andrade marianaandrade@ua.pt

***Abstract*—Este relatório explora o papel dos Geometry Shaders em computação visual, destacando sua contribuição para a evolução da computação gráfica. Apresenta um histórico dos shaders, com foco nos Geometry Shaders e nas suas capacidades de manipulação geométrica em tempo real. Através de uma implementação prática em OpenGL, demonstra 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 repleta de avanços significativos que transformam a nossa capacidade de criar e interagir com ambientes virtuais. Desde os primeiros gráficos *raster* até aos modelos 3D complexos e altamente detalhados de hoje, cada avanço abriu novos horizontes para a expressão digital. Em maio de 1988[12], um marco importante foi alcançado, pela *Pixar* com a introdução do conceito de ***shader*** na versão *3.0* da sua *interface RenderMan*, inaugurando o início de uma nova era de possibilidades 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.

Com o aperfeiçoamento dos *shaders* e a inovação das tecnologias, especialmente no desenvolvimento das unidades de processamento gráfico (*GPU*), foi possível exercer um controlo mais refinado sobre o processamento, permitindo a criação de imagens com um nível de detalhe e realismo anteriormente atingíveis. Transitamos de simples funções de manipulação de luz e cor para técnicas mais complexas e sofisticadas, como os *Vertex Shaders* e os *Fragment Shaders*.[2]

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 permitiu aos programadores manipular dinamicamente vértices dentro do programa de *shader*. Isso não só superou as limitações dos pipelines fixos dos primeiros gráficos 3D mas também abriu caminho para efeitos visuais revolucionários e otimizações de desempenho.

Este artigo tem como objetivo aprofundar a utilidade e a aplicação de *geometry shaders* no domínio da computação visual. Ao explorar o contexto histórico, aspetos técnicos e possíveis implementações práticas, pretende-se informar e ainda inspirar a exploração desta ferramenta em projetos. Através de exemplos e demonstrações práticas, espera-se destacar como os *geometry shaders* podem enriquecer técnicas de renderização, criar efeitos visuais estonteantes e otimizar o desempenho, esclarecendo os seus desafios e limitações, bem como possíveis soluções.

II. Shaders

Os *Shaders* são programas pequenos, porém robustos, que operam nas *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*[11], um sistema que restringia severamente a criatividade dos programadores devido ao seu conjunto limitado de funcionalidades para processamento gráfico. Este sistema permitia apenas operações básicas de renderização, como a aplicação de texturas e iluminação simplificada,deixando pouco espaço para personalização ou inovação.

No entanto, a transição para o *programmable pipeline*, marcada pela adoção generalizada dos *shaders*, permitiu aos programadores escrever os seus próprios programas de *shader* em linguagens de alto nível como *GLSL* (OpenGL Shading Language) e *HLSL* (High-Level Shading Language)[10], sendo executados diretamente pela *GPU*. Essas linguagens possibilitaram a implementação de uma variedade quase ilimitada de efeitos gráficos complexos, desde o cálculo da iluminação e da cor até efeitos sofisticados de pós-processamento e manipulação de texturas.

A figura abaixo ilustra as etapas da pipeline gráfica, destacando as fases que podem ser personalizadas através do uso de *shaders*:

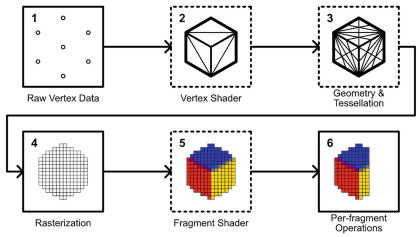

Fig. 1. [6] As etapas do pipeline gráfico. As etapas rodeadas por uma linha tracejada indicam partes do pipeline que podemos personalizar com *shaders*.

Entre os diferentes tipos de *shaders*, destacam-se:

- ***Vertex Shaders***: Processam cada vértice individualmente, determinando as suas propriedades, como posição e cor. São fundamentais para a animação de personagens e a implementação de técnicas de iluminação dinâmica, adaptando-se às mudanças de perspetiva e luz na cena.
- ***Geometry Shaders***: Oferecem um controle mais granular sobre a geometria e permitem a criação e manipulação de formas em tempo real, abrindo portas para a geração de geometrias dinâmicas complexas e efeitos visuais avançados, como a tesselação adaptativa de superfícies.

- ***Fragment Shaders (ou Pixel Shaders)***: Responsáveis pelos detalhes finais de cada pixel, incluindo cor, iluminação e textura. São cruciais para criar efeitos realistas de superfície, como reflexos, refrações e sombras suaves, definindo a qualidade visual dos objetos renderizados.

Enquanto os *vertex* e *fragment shaders* desempenham papéis fundamentais na definição das propriedades geométricas básicas e nos detalhes visuais dos *pixels*, respetivamente, os *geometry shaders* introduzem uma camada adicional de flexibilidade e poder ao processo de renderização.

III. Geometry Shaders

Os ***geometry shader*[9]** são um elemento inovador na renderização gráfica, estrategicamente posicionados no pipelina gráfico entre a etapa de *primitive assembly* e o *fragment shader*. Embora sua utilização seja opcional, a sua contribuição é extraordinária, caracterizada pela capacidade única de manipular múltiplos vértices e gerar novas geometrias a partir de primitivas completas como vértices organizados em *arrays*. Esta etapa é realizada após o *Vertex Shader* ou, em casos em que a pipelines inclui a *tessellation*, após o *Tessellation Evaluation Shader*.

Ao contrário dos *vertex shaders*, que processam cada vértices individualmente, *geometry shaders* processam primitivas inteiras. Eles têm a capacidade de criar novos conjuntos de vértices e formar novas primitivas, o que permite a ampliação da geometria. Este poder de transformação abre um leque de possibilidades para a criação de geometrias complexas e a implementação de efeitos dinâmicos, inclusive efeito de partículas complexas, como a fumaça, fogo, pelos ou relva (que é enriquecido pelo nível de detalhe e dinamismo que este *shader* proporciona), e a capacidade de transformação de primitivas como pontos em triângulos, que permitem a implementação de efeitos visuais e otimizações no processo de renderização.

Outra funcionalidade importante dos *Geometry Shaders* é a sua utilização com *feedback* de transformação, permitindo a divisão de um fluxo de entrada de dados de vértices em múltiplos subfluxos. Esta característica é particularmente valiosa para a criação de técnicas e algoritmos diversificados diretamente na GPU, otimizando a manipulação e o processamento de dados geométricos.

IV. Implementação Geometry Sharders em OpenGL

Após a exploração dos conceitos teóricos e das funcionalidades dos *geometry shaders*, podemos aplicar esses conceitos num cenário prático através da implementação de um exemplo introdutório com *OpenGL*. Este exercício prático tem como objetivo não apenas consolidar nossa compreensão teórica, mas também proporcionar uma oportunidade valiosa para explorar a criação de geometrias dinâmicas e o desenvolvimento de efeitos visuais mais elaborados.

A. Configurações do Ambiente

Para facilitar o processo de configuração, é disponibilizado um projeto de exemplo no GitHub que serve como ponto de partida: **GitHub - Implementação de Geometry Shader com OpenGL**. Este repositório contém um código-base necessário e instruções detalhadas de configuração. Siga as instruções descritas para configurar o ambiente de desenvolvimento.

B. Criar um Geometry Shader

Os *geometry shaders* são criados seguindo o padrão de desenvolvimento comum a que qualquer outro tipo de shader no *OpenGl*. O processo inicia com ***'glCreateShader(GL_GEOMETRY_SHADER)'*** para gerar o shader, seguido pela compilação do seu código-fonte usando o ***glCompileShader()***. No entanto, antes deles serem vinculados, deve ser especificado:

- O **tipo de primitiva de entrada** que o *shader* espera receber.
- O **tipo de primitiva de saída** que o *shader* irá gerar.
- O **número máximo de vértices** que o *shader* pode produzir.

De tal forma, vamos começar por analisar um exemplo básico de *geometry shader*.

Listing 1. Geometry Shader Basic

```
#version 330 core
layout(points) in;
layout(triangle_strip, max_vertices = 3) out;
void main()
{
    for(int i = 0; i < 3; i++) {
        gl_Position = gl_in[0].gl_Position +
            vec4(i == 0 ? -0.1 : 0.1,
            i == 2 ? -0.1 : 0.1, 0.0, 0.0);
        EmitVertex();
    }
    EndPrimitive();
}
```

Este *geometry shader* simplesmente amplia cada vértice de entrada em um novo triângulo. Antes de estudarmos o funcionamento do código, é pertinente percebermos as características deste exemplo que são únicas para este tipo de *shaders*. Primeiro, no topo do *shader*, temos um par de qualificadores de *layout* com a declaração dos tipos primitivos de entrada e saída e o número máximo de vértices. Neste caso em especifico, o shader será executado uma vez para cada ponto renderizado e vai gerar um triângulo como saída. Para visualizar a renderização, podemos executar o ficheiro *create_geometry_shader.py* de maneira a obter o mesmo resultado do lado esquerdo da Figura 2.

A seguir, temos uma listagem dos tipos primitivos aceites como entradas pelo *geometry shader* e dos tipos primitivos correspondentes que podem ser usados em comandos de desenho.

Ainda no trecho de código, podemos analisar funções 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

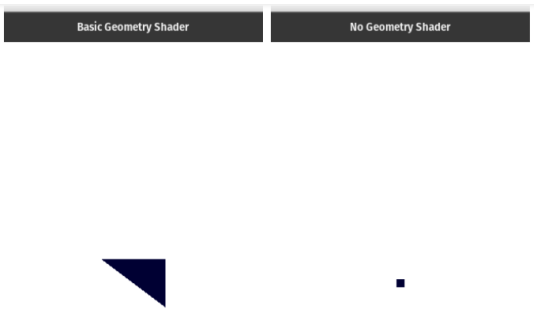

Fig. 2. Resultado do programa com e sem geometry shader

Geometry Shader Primitive Type	Accepted Drawing Command Modes
`points`	GL_POINTS, GL_PATCHES[1]
`lines`	GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_PATCHES[1]
`triangles`	GL_TRIANGLES, GL_TRIANGLE_STRIP GL_TRIANGLE_FAN, GL_PATCHES[1]
`lines_adjacency`[2]	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY
`triangles_adjacency`[2]	GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY

Fig. 3. Geometry shader primitive types and Accepted Drawing Modes

caso do tipo primitivo de saída ser *points*, ela apenas se limita a cria um novo ponto independente.

2) EndPrimitive(): Indica o fim da primitiva atual e sinaliza ao *OpenGl* que, na próxima vez que *EmintVertex*() for invocada, recomeça uma nova primitiva. 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.

O shader apresentado é um exemplo de *geometry shader* simples. No entanto, existem os shaders *culling*, que não fazem absolutamente nada, e os de sharders de passagem, que não alterar a geometria inicial. Podemos visualizar estes exemplos nos ficheiros *culling.glsl*, *geometry_shader_passagem.glsl* e *geometry_shader_points.glsl*, e substitui-los no código do ficheiro *test_geometry_shader.py*.

C. Implementação de Geometry Shaders[8]

Depois de aprendermos como os *geometry shaders* funcionam, passa a ser mais intuitivo a sua utilização. Agora podemos transformar pontos em linhas paralelas - usando *line_strip* - e ainda evoluir para a geração de estrelas dinâmicas variando em cor e número de lados.

Aqui começamos por variar as cores das linhas que estão ser desenhadas, permitindo que cada uma tenha uma cor única. Para isso, temos os seguintes passos:

1) No Vertex Shader, adicionamos um atributo de cor de entrada e passamos essa cor para a saída, que será recebida pelo *Geometry Shader*. O *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 gerar estrelas a partir de um ponto. Cada ponto atuará como o centro de uma estrela, com a possibilidade de ajustar o número de lados para criar diferentes formas. Para isso,

2) Modificamos o *Geometry Shader* para gerar estrelas com um número especifico de pontas a partir de um único ponto de entrada. Para isso, precisamos de definir constantes: o número de pontas da estrela, e os raios externo e interno da estrela. Estas variáveis serão usadas para calcular a posição dos vértices da estrela, como podemos ver no exemplo seguinte:

```
#version 330 core
layout(points) in;
layout(line_strip, max_vertices = 22) out;

in vec3 vColor[];
out vec3 fColor;

const float PI = 3.1415926;
const int numPoints = 10;
const float outerRadius = 0.2;
const float innerRadius = 0.1;

void main() {
 fColor = vColor[0];

 for(int i = 0; i <= numPoints * 2; i++) {
   float angle = 2.0 * PI *
      float(i) / float(numPoints);
   float radius = i % 2 == 0 ?
     outerRadius : innerRadius;

    vec4 offset = vec4(radius *
      cos(angle), radius * sin(angle),
      0.0, 0.0);
    gl_Position = gl_in[0].gl_Position
        + offset;
    EmitVertex();
 }

 EndPrimitive();
}
```

3) Aqui atualizamos o *Vertex Shader* para passar apenas a posição do centro do círculo, para 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);
}
```

4) Finalmente, compilamos e executamos o programa. Agora, devemos ver estrelas coloridas renderizadas no ecrã, cada um com uma cor específica, como na figura 4. Ainda podemos ajustar o valor de **'numPoints'** para alterar o número de lados das estrelas e variar os valores dos raios.

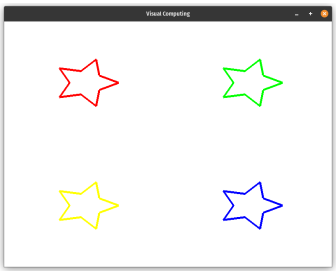

Fig. 4. Estrelas coloridas geradas com Geometry Shader.

D. Exploração e Experimentação

Caso tenhas dificuldade em realizar a implementação, existe uma solução alternativa disponível. Ao aceder à pasta "*solution*" no projeto fornecido, podes 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 em cada passo, podes analisar minuciosamente os arquivos de *shaders* na pasta "*shaders*", que foram utilizados na implementação desta solução.

Se experimentaste ajustar os raios das estrelas, percebeste que a aproximação dos raios afeta diretamente a perceção da forma gerada pelo *shader*. Ou seja, quanto mais próximo os valores dos raios forem, menos pronunciadas serão as pontas da estrelas, resultando em formas mais parecidas com flores ou objetos com bordas levemente onduladas, invés de um estrela com pontas distintas. Poderás ainda concluir, que quando estes valores são iguais, a distinção entre as pontas e os vales da estrela desaparece, aproximando-se o resultado de um círculo perfeito ou de um polígono regular (dependendo do número de pontos definidos), uma vez que todos os vértices da "estrela" passam a estar à mesma distância do centro.

Para ires mais além, podes tentar que as estrelas resultantes sejam diferentes para cada ponto inicial, como podes ver na Figura 5.

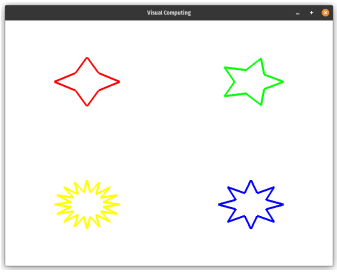

Fig. 5. Estrelas coloridas com diferentes número de lados

V. Aplicações de Geometry shaders

Após compreendermos os *geometry shaders*, podemos apreciar sua vasta aplicabilidade em cenários do mundo real. Estes *shaders* são notáveis pela versatilidade e eficiência em variados contextos digitais. Aqui, exploramos algumas aplicações práticas que destacam o potencial dos *Geometry Shaders*.

A. Renderização de Terrenos em 3D

Os *Geometry Shaders* desempenham um papel crucial na criação de paisagens tridimensionais detalhadas e realistas, permitindo ajustes dinâmicos na geometria do terreno. Essa capacidade é particularmente valiosa em jogos e simulações virtuais, onde a autenticidade do ambiente pode imensamente enriquecer a experiência do utilizador.

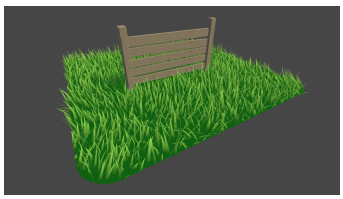

Fig. 6. Aplicação de geometry shader na renderização de Relva em 3D

Na Figura 6, é apresentado um terreno coberto de relva, cuja riqueza de detalhes é alcançada através dos *Geometry Shaders*, ilustrando a robustez dessa tecnologia na simulação de elementos naturais complexos.

B. Efeitos de Partículas

A geração de efeitos de partículas, como fumaça, fogo e neblina, é outra aplicação notável dos *Geometry Shaders*. Eles oferecem um controle granular sobre cada partícula, permitindo a criação de efeitos visuais dinâmicos e realistas. Esse controle aprimorado facilita a simulação de fenómenos naturais em ambientes virtuais, adicionando uma camada de imersão e realismo às cenas.

C. Detalhe em Personagens Digitais

Assim como na renderização de terrenos, os *Geometry Shaders* são essenciais para adicionar detalhes a personagens digitais, especialmente para simular cabelos e peles. Eles possibilitam a criação de texturas dinâmicas que aumentam significativamente a qualidade visual das personagens, conferindo-lhes uma aparência mais natural e convincente.

VI. Implementação Avançada de Geometry Shaders: Simulação do campo estrelar

Após uma exploração de aplicações dos *geometry shader*, avançamos para uma implementação de um campo estrelar baseado em simulações de *N-Body*. Esta simulação, disponível na pasta '*simulation*', não só destaca nossa habilidade técnica, mas também demonstra o potencial destes shaders para criar visualizações astronómicas complexas e envolventes.

A. Conceitos Importantes

Para a implementação do campo estrelar, é essencial compreender:

- **Modelagem Estelar como pontos de dados:** Representamos inicialmente cada estrela como um ponto no espaço virtual. Este método eficiente serve de base para a subsequente expansão geométrica, transformando pontos em representações visuais detalhadas de estrelas.
- **Dinâmica de N-Body para Movimento Estelar:** Utilizamos simulações de *N-Body* para modelar as interações gravitacionais entre corpos celestes, permitindo que cada estrela influencie o movimento das outras. Esta abordagem cria um sistema dinâmico que mimetiza a complexidade do universo. Processar estas simulações em *Geometry Shaders* aproveita a paralelização da GPU, facilitando cálculos eficientes das forças e movimentos de um vasto número de estrelas.

B. Implementação do Shader

Nossa implementação tira proveito dos gráficos computacionais modernos, particularmente o poder dos *Compute Shaders*, para realizar simulações na GPU. Esta abordagem acelera significativamente o processamento em comparação com a CPU. Os *Geometry Shaders* são essenciais para renderizar os corpos celestes como objetos geométricos tridimensionais visualmente impactantes.

No código desenvolvido, tem-se quatro componentes chave:

1) **Vertex Shader (vertex_shader.glsl):** Prepara os dados de vértice para renderização, incluindo informações de posição dos corpos celestes.
2) **Geometry Shader (geometry_shader.glsl):** Recebe pontos e os transforma em quadrados.
3) **Fragment Shader (fragment_shader.glsl):** Gera o efeito de brilho suave das estrelas, suavizando as bordas das formas para criar um efeito de círculo.
4) **Compute Shader (compute_shader.glsl):** Atualiza as posições e velocidades dos corpos celestes, fundamentando a simulação de *N-Body*.

Os resultados da simulação de campo estrelar são visualmente impactantes. Ao executar o programa na pasta '*simulation*', você pode testemunhar uma dança celeste de estrelas que se movimentam e interagem em tempo real. As estrelas não são meros pontos estáticos; elas brilham e oscilam, imitando o cintilar natural que observamos no céu noturno. Este efeito é conseguido pela combinação inteligente de *shaders* que trabalham em uníssono para simular as propriedades físicas e visuais das estrelas.

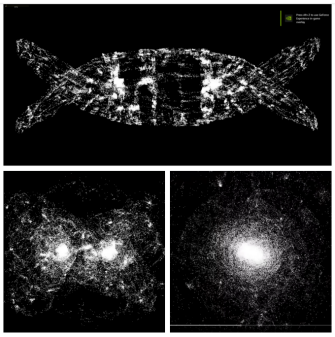

Fig. 7. Resultados da simulação do campo estrelar, em diferentes tempos

Esta simulação transcende a mera visualização estática, ilustrando como uma integração cuidadosa de diferentes *shaders* pode abrir novos horizontes na visualização astronómica e científica.

VII. 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 à necessidade de ter um entendimento mais profundo, não apenas da linguagem, mas também da sua 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 de ser executada numa variedade de dispositivos, pode ser necessário fornecer alternativas para dispositivos que não suportam *geometry shaders*.
3) **Rendimento** [3]: Embora os *geometry shaders* possam melhorar o desempenho em alguns casos, também podem diminuir se forem 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.

VIII. Recursos Adicionais para Aprendizagem

Para aprofundar o conhecimento em *geometry shaders*, recomendamos a seguinte 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

IX. Conclusão

Em suma, os geometry shaders destacam-se com a sua inovação e criatividade no campo da computação gráfica. Eles fornecem aos programadores uma ferramenta versátil para conceber experiências visuais ricamente detalhadas e complexas, como a simulação do campo estrelar. Mais do que isso, ilustram o poder transformador da computação gráfica, abrindo novos caminhos para explorarmos e interagirmos com o universo 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.

[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.