3
Fundamentos

Este capítulo apresenta uma visão geral do *pipeline* de renderização da versão 4.0 da API OpenGL, tendo como foco os recém-introduzidos estágios de tesselação. Em seguida, é realizada uma abordagem introdutória sobre texturas procedimentais com o objetivo de mostrar alguns aspectos sobre síntese de texturas como alternativa aos tradicionais métodos de obtenção de valores por meio de imagens.

3.1
Pipeline de renderização em OpenGL

Com a introdução de dois novos estágios programáveis chamados *tessellation control shader* e *tessellation evaluation shader* e um estágio fixo chamado *primitive generator*, o *pipeline* de renderização da OpenGL para a atual geração de hardwares gráficos pode ser caracterizado conforme mostra a Figura 2. O processo de tesselação é realizado sobre um novo tipo de primitiva criada especificamente para este fim, intitulada *patch*, a qual consiste em um conjunto de vértices ordenados, mas sem uma topologia definida, já que esta é definida apenas ao final do processo de subdivisão. Cada vértice de um *patch* possui seus atributos próprios, bem como atributos globais associados ao *patch* a que pertencem.

Cada *patch* possui um número fixo de vértices, que deve ser definido pelo programador antes das chamadas aos métodos de renderização que fazem uso de *shaders* de tesselação. Este número não pode exceder a constante especificada pela OpenGL para a máxima quantidade de vértices contidos em um *patch*.

PUC-Rio - Certificação Digital Nº 0921333/CA

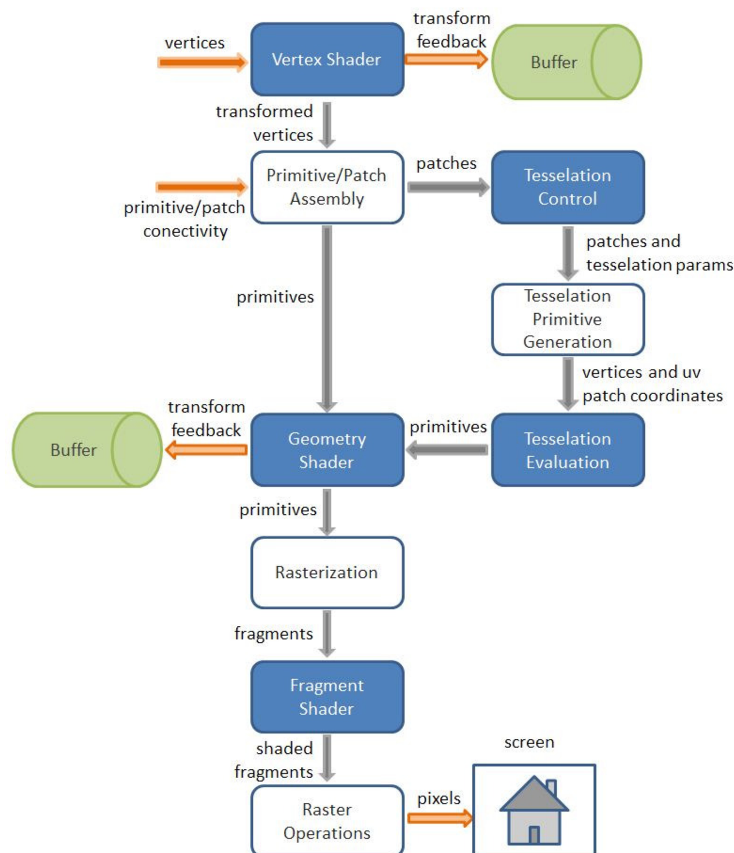

Figura 2: Pipeline de renderização da OpenGL (GL Pipeline, 2011).

3.1.1. Vertex Shader

O *vertex shader* recebe da aplicação o conjunto de vértices que constituem o modelo original e é responsável por transformá-los conforme necessário. Caso exista um *shader* de tesselação ativo no *pipeline*, a operação de transformação da posição do vértice atual para o espaço de projeção por meio do produto com as matrizes *model*, *view* e *projection* se torna desnecessária nesta etapa. Assim, a transformação para o espaço de projeção deve ser computada no *tessellation evaluation shader*, quando todos os vértices da malha tiverem sido gerados.

A forma mais interessante de utilização deste estágio em conjunto com os de tesselação é no caso de aplicações que possuem modelos com animações. Nesse cenário, transformações que realizam deformações na malha (como *skinning* e *morphing*) são realizadas diretamente sobre os pontos de controle do *patch* da malha de baixa resolução no *vertex shader* antes de serem enviados aos estágios de subdivisão (Ni & Castano, 2009). Com isso, as operações de animação

PUC-Rio - Certificação Digital Nº 0921333/CA

são realizadas em uma frequência reduzida e os vértices gerados durante o estágio de tesselação serão interpolados de acordo com as posições dos vértices do *patch* já transformado e estarão em suas posições desejadas para um dado quadro da animação, conforme mostra a Figura 3.

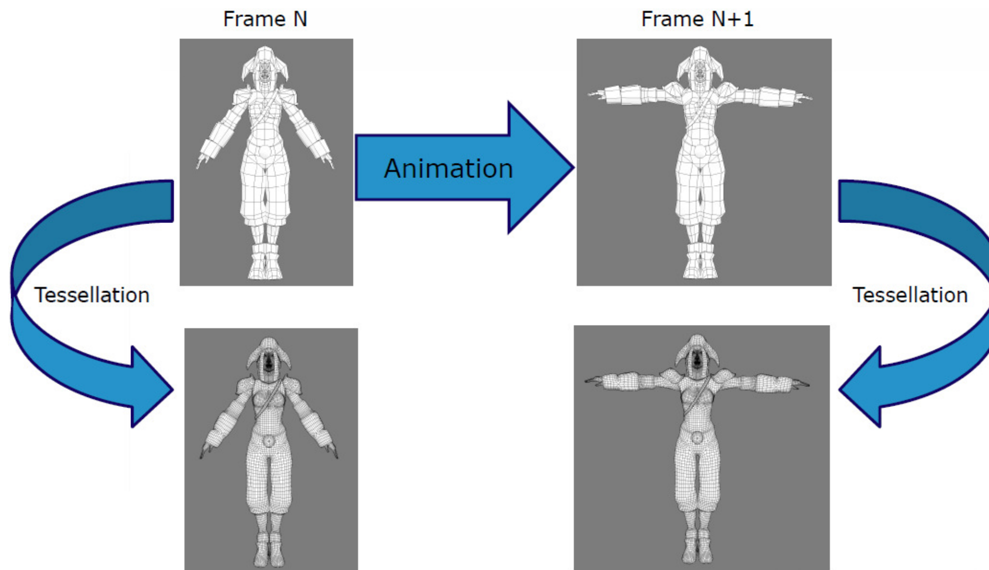

Figura 3: Morphing no vertex shader (Tatarinov, 2008).

3.1.2. Tessellation Control Shader

O *tessellation control shader* recebe um *patch* de entrada e é executado para cada vértice pertencente a ele. Este estágio é responsável pela computação dos atributos do vértice atual e atributos adicionais do *patch* de saída. Entre estes atributos estão os níveis de tesselação interno e externo e o número de vértices de saída.

O nível de tesselação interno é composto por dois valores de ponto flutuante e utilizado pelo *primitive generator* para aproximar o número de segmentos nas arestas internas da primitiva, com o objetivo de oferecer um aspecto homogêneo para a subdivisão. O nível de tesselação externo é composto por até quatro valores de ponto flutuante que definem o número de segmentos de cada aresta externa da primitiva a ser gerada. O número de vértices produzidos pelo *tessellation control shader* deve ser especificado por meio de uma declaração de *layout* para o *patch* de saída. Este valor corresponde ao número de vezes que este *shader* será executado e não pode exceder o valor máximo definido na aplicação para o tamanho do *patch*. O efeito da variação dos níveis de tesselação sobre a primitiva

PUC-Rio - Certificação Digital Nº 0921333/CA

subdividida é ilustrado na Figura 4. Um nível de tesselação externo maior que o interno resulta na primitiva observada na Figura 4(a), enquanto um nível interno maior resulta na observada na Figura 4(b); por fim, níveis de tesselação interno e externo iguais resultam na primitiva exibida na Figura 4(c).

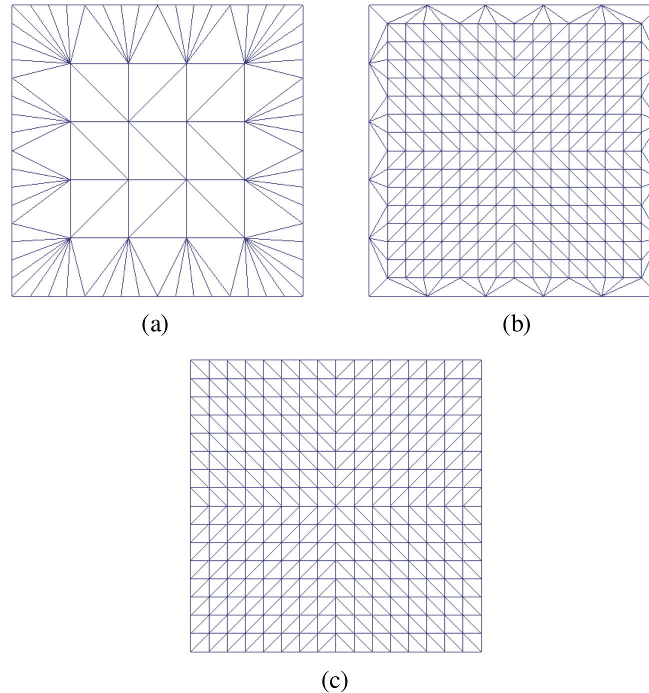

Figura 4: Efeito da variação dos níveis de tesselação interno e externo.

3.1.3.
Primitive Generator

O *tessellation primitive generator* é executado se houver um *tessellation evaluation shader* ativo. Este estágio recebe o número de vértices definido no estágio anterior e realiza a subdivisão do *patch* em um conjunto de pontos, linhas ou triângulos, de acordo com os níveis especificados no *tessellation control shader* e com a declaração do *layout* de entrada do *tessellation evaluation shader*, que será discutida adiante.

A cada vértice da primitiva resultante são associadas coordenadas do tipo (u, v) ou (u, v, w) definidas em um espaço paramétrico. Para triângulos, são as coordenadas baricêntricas (u, v, w) indicando a posição do vértice gerado em relação às coordenadas do triângulo original do *patch* de entrada. Vértices de *isolines* ou *quads*, por outro lado, recebem coordenadas do tipo (u, v) que indicam as posições horizontal e vertical, relativas à primitiva do *patch* original.

PUC-Rio - Certificação Digital Nº 0921333/CA

PUC-Rio - Certificação Digital Nº 0921333/CA

3.1.4. Tessellation Evaluation Shader

O *tessellation evaluation shader* recebe os vértices da primitiva subdividida pelo *primitive generator* e gera cada vértice da primitiva de saída com suas respectivas posições e atributos associados. Este *shader* é executado de forma independente para cada vértice da primitiva de saída, mas permite ao programador o acesso a todos os vértices do *patch* de entrada. Além disso, o programador pode acessar diretamente a posição do vértice atual em relação à primitiva subdividida, isto é, as coordenadas (*u, v*) ou (*u, v, w*) mencionadas anteriormente.

Nesta etapa, cabe ao programador definir o tipo de subdivisão realizada pelo *primitive generator* através de uma declaração de *layout* do *patch* de entrada para o *tessellation evaluation shader*, a qual também estabelece o método usado para derivar o número e o espaçamento entre os segmentos de acordo com o nível de tesselação estabelecido no *tessellation control shader*, além da orientação dos triângulos gerados nos modos *triangles* e *quads*, que pode ser em sentido horário ou anti-horário. Resultados de subdivisões simples utilizando diferentes métodos de segmentação e espaçamento podem ser vistos na Figura 5.

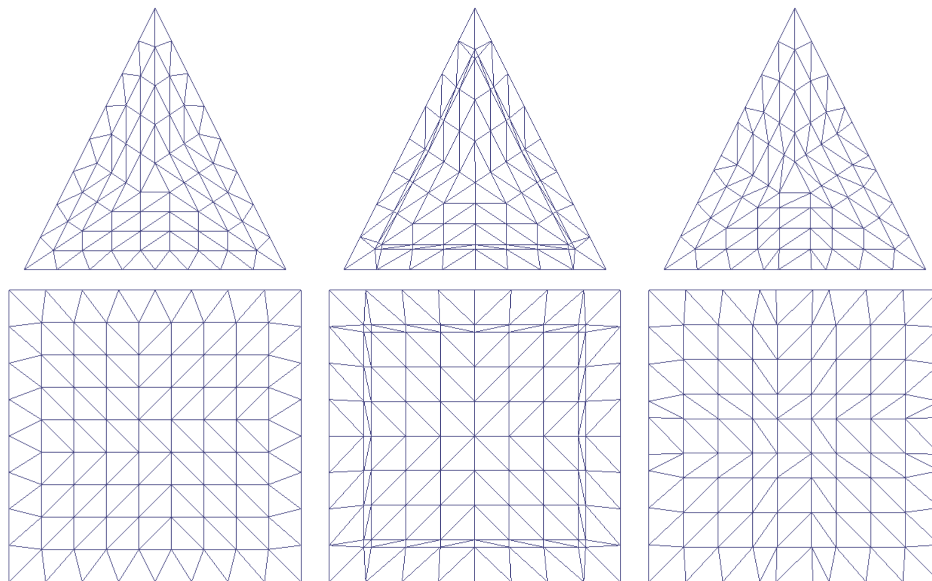

Figura 5: Diferentes modos de espaçamento na subdivisão de triângulos e quads.

3.1.5. Geometry Shader

O *geometry shader*, introduzido com o lançamento das interfaces de programação DirectX 10 e OpenGL 3.2, ofereceu a programadores os primeiros meios de criação de geometria em GPU, apesar de bastante limitados principalmente em termos do tamanho da saída a cada instância desse *shader*. Este estágio fica localizado após o *primitive assembly* e é executado para cada primitiva (cujos vértices são recebidos diretamente do *vertex shader*, caso não existam *shaders* de tesselação ativos), emitindo uma ou mais primitivas de saída do mesmo tipo, descartando, em seguida, a primitiva recebida originalmente.

Neste estágio, são disponibilizadas informações de adjacência da primitiva de entrada, tornando o *geometry shader* adequado para algoritmos que requerem coleta de informações da superfície sendo renderizada. Alguns exemplos de aplicações que utilizam *geometry shaders* são volumes de sombra (Stich, Wächter, & Keller, 2007), simplificação e controle do nível de detalhe de acordo com o ponto de vista do observador (Hu, Sander, & Hoppe, 2010) (DeCoro & Tatarchuk, 2007) e extração de iso-superfícies em GPU (Tatarchuk, Shopf, & DeCoro, 2007).

PUC-Rio - Certificação Digital Nº 0921333/CA

3.1.6. Fragment Shader

O *fragment shader* é responsável pelas operações aplicadas sobre os fragmentos resultantes da rasterização das primitivas e seus dados associados, sendo utilizado fundamentalmente para atualizar o *framebuffer* ou a memória de textura. Neste estágio, é possível ler variáveis de entrada correspondentes aos fragmentos gerados, as quais podem ser internas (como a posição do fragmento), ou definidas pelo usuário nos estágios que antecedem o *fragment shader* (as quais são interpoladas de acordo com a primitiva rasterizada), mas não é possível acessar dados de fragmentos vizinhos.

Neste estágio, também é possível atribuir variáveis de saída, que podem ser utilizadas para operações por fragmento subsequentes, como a escrita de dados em um *framebuffer object* ou o teste de profundidade com o *depth buffer* para determinar se determinado fragmento deve seguir pelo *pipeline* ou ser descartado.

O conjunto de operações e testes realizados por fragmento na etapa anterior à escrita no *framebuffer* não mostrados, em ordem, na Figura 6.

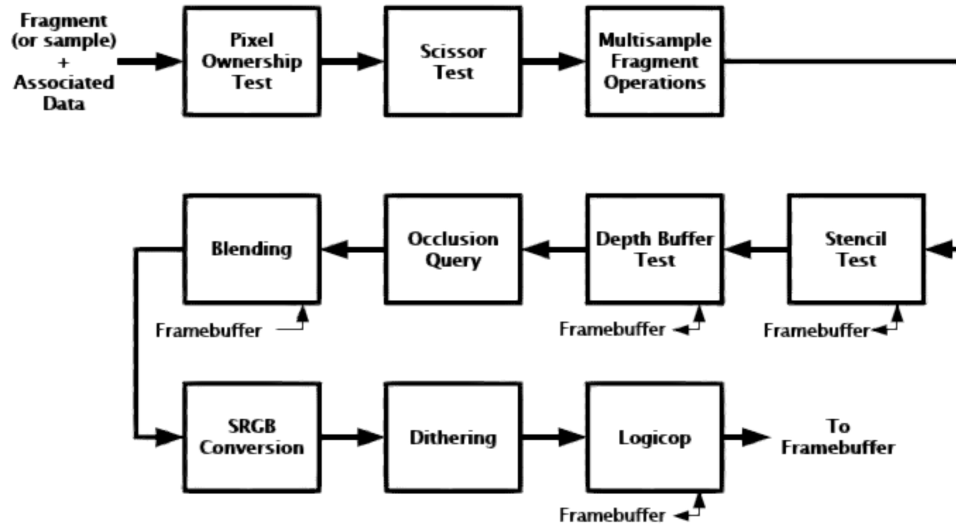

Figura 6: Ordem de operações por fragmento (OpenGL 4.2, 2012).

3.2
Introdução a texturas procedimentais

A geração texturas procedimentais é um campo de grande interesse em computação gráfica em virtude da possibilidade da criação de imagens que representem a natureza aleatória de diferentes tipos de materiais através de funções que codificam efeitos visuais de acordo com um determinado algoritmo, constituindo uma alternativa ao tradicional acesso a texturas codificadas em imagens para obtenção de informações. Não restrita apenas a casos bidimensionais, a síntese procedimental de texturas também é interessante no caso de texturas volumétricas (que possuem um alto custo de armazenamento no caso pré-computado) ou mesmo de geometria (Ebert, et al., 2002), sendo possível gerar uma variedade de formas complexas encontradas na natureza, como a vegetação ilustrada na Figura 7, renderizada através de *ray tracing*. A maioria dos efeitos gerados procedimentalmente são baseados no algoritmo chamado *Perlin Noise*, proposto por Ken Perlin (Perlin, 1985), que posteriormente também apresentou uma versão aperfeiçoada desse algoritmo em termos de desempenho e qualidade do ruído gerado (Perlin, 2002) (Perlin, 2004).

PUC-Rio - Certificação Digital Nº 0921333/CA

Figura 7: Vegetação gerada procedimentalmente (Ebert, et al., 2002).

A síntese de texturas na GPU para aplicações em tempo real possui um conjunto de vantagens e desvantagens que devem ser levadas em consideração ao se avaliar o efeito que se deseja renderizar. É mais comum a utilização balanceada entre texturas armazenadas em imagens e texturas geradas procedimentalmente, em que o primeiro caso cobre a maioria dos objetos existentes no mundo real, como fachadas de prédios, pinturas ou pavimentos, enquanto o segundo cobre elementos que possuem aspectos de cunho mais aleatório, como nuvens, tecidos ou terrenos.

Uma das razões para a efetividade de texturas procedimentais é a incorporação de elementos aleatórios de uma forma controlada. O algoritmo fundamental para geração de ruído consiste em aproximar o que seria obtido por meio do ruído branco e realizar uma filtragem das frequências além de um determinado ponto de corte. Essa aproximação pode ser avaliada para pontos arbitrários sem a necessidade de pré-computação de dados relacionados ao volume que contém esses pontos.

Comparadas a texturas armazenadas em imagens, texturas procedimentais tem um baixo custo de memória, já que são representadas apenas por algoritmos que avaliam determinadas funções, proporcionando uma notável vantagem em comparação a texturas 2D, a qual é ainda maior para texturas 3D. Por serem

PUC-Rio - Certificação Digital Nº 0921333/CA

definidas de forma algorítmica ao invés de amostragem de valores, essas texturas também não apresentam limitações de área ou resolução e podem ser aplicadas sobre objetos de qualquer escala sem que sejam visualizadas replicações, redução de detalhes ou artefatos de amostragem. Além disso, algoritmos para geração de texturas procedimentais são escritos com base em parâmetros que podem ser modificados com facilidade, possibilitando a criação de uma variedade de efeitos a partir do mesmo código.

Por outro lado, dependendo da complexidade do algoritmo ou da superfície, a avaliação da função para cada ponto de uma determinada superfície pode se tornar custosa, resultando em um desempenho menor que o do acesso a uma textura pré-computada. Em geral, texturas procedimentais também requerem algum tipo de filtragem para evitar artefatos causados por *aliasing*, os quais podem ser demasiadamente acentuados, dependendo do efeito implementado. Uma abordagem apresentada por Cook e DeRose (Cook & DeRose, 2005) propõe um método baseado na teoria de *wavelets* (Chui, 1992) para reduzir o efeito de *aliasing* com um pequeno aumento no custo de avaliação da função.

Uma aplicação em particular para texturas procedimentais é a combinação das texturas sintetizadas com a técnica de *bump mapping* para a simulação de detalhamento tanto regular quanto aleatório sobre superfícies. É importante ressaltar que, como as outras técnicas baseadas em imagens, por se tratar apenas de um efeito de iluminação, o *bump mapping* procedimental não realiza modificações sobre a geometria do objeto, sendo, portanto, incapaz de representar corretamente as suas silhuetas. Tal limitação torna este mapeamento adequado apenas para casos em que as irregularidades não são acentuadas em relação à superfície propriamente dita.

PUC-Rio - Certificação Digital Nº 0921333/CA