GOVERNO DO ESTADO DO RIO DE JANEIRO
SECRETARIA DE ESTADO DE CIÊNCIA E TECNOLOGIA
FUNDAÇÃO DE APOIO À ESCOLA TÉCNICA
CENTRO DE EDUCAÇÃO PROFISSIONAL EM TECNOLOGIA DA INFORMAÇÃO
FACULDADE DE EDUCAÇÃO TECNOLÓGICA DO ESTADO DO RIO DE JANEIRO
FAETERJ/PETRÓPOLIS

INTEGRAÇÃO OPENGL E BULLET PHYSICS

Gabriel Sussumu Kato

Petrópolis - RJ

Junho, 2016

Gabriel Sussumu Kato

INTEGRAÇÃO OPENGL E BULLET PHYSICS

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Tecnólogo em Tecnologia da Informação e da Comunicação da Faculdade de Educação Tecnológica do Estado do Rio de Janeiro Faeterj/Petrópolis, como requisito parcial para obtenção do título de Tecnólogo em Tecnologia da Informação e da Comunicação.

Orientador:

D.Sc. Sicilia Ferreira Ponce Pasini Judice

Petrópolis - RJ

Junho, 2016

Folha de Aprovação

Trabalho de Conclusão de Curso sob o título *"INTEGRAÇÃO OPENGL E BULLET PHYSICS"*, defendida por Gabriel Sussumu Kato e aprovada em Dia de Mês de Ano, em Petrópolis - RJ, pela banca examinadora constituída pelos professores:

Prof. D.Sc. Sicilia Ferreira Ponce Pasini Judice
Orientador

Prof. Banca Interna
Faculdade de Educação Tecnológica do Estado do Rio de Janeiro Faeterj/Petrópolis

---

Prof. Banca Interna
Faculdade de Educação Tecnológica do Estado do Rio de Janeiro Faeterj/Petrópolis

Declaração de Autor

Declaro, para fins de pesquisa acadêmica, didática e técnico-científica, que o presente Trabalho de Conclusão de Curso pode ser parcial ou totalmente utilizado desde que se faça referência à fonte e aos autores.

Gabriel Sussumu Kato
Petrópolis, em Dia de Mês de Ano

Resumo

Este trabalho demonstra o processo de integração entre a API (*Application Programming Interface*, ou Interface de Programação de Aplicativo) gráfica OpenGL e o motor de física *Bullet Physics*. Nele, foi desenvolvida uma simples demonstração onde é possível observar diversos fenômenos físicos como gravidade, colisão, inércia e fricção. Tal integração foi feita visando tornar mais agradável o aprendizado de qualquer uma das tecnologias envolvidas. A aplicação foi feita utilizando a biblioteca SDL2 (*Simple DirectMedia Layer*), para tratar a criação de janelas e entrada e saída de dados do usuário em múltiplas plataformas, como Windows, MacOS e Linux. Para a criação do ambiente gráfico, foi utilizada a API gráfica multiplataforma OpenGL, escrita na versão 3.3 ou superior, também conhecida como *modern*, o que possibilita utilizar recursos não existentes em versões anteriores. O motor de física Bullet tem suporte a recursos comuns, como gravidade, colisão, corpos rígidos, entre outros. Todas as bibliotecas escolhidas são de código livre. A ferramenta escolhida para o desenvolvimento foi a IDE (*Integrated Development Environment*, ou Ambiente de Desenvolvimento Integrado) Visual Studio Community 2015, para Windows, também gratuita, porém é possível utilizar qualquer outro compilador para obter o mesmo resultado.

Palavras-chave: API gráfica. Motor de física.

Abstract

This work shows the integration process between the graphical API (Application Programming Interface) OpenGL and the physics engine Bullet Physics. It was developed a simple demonstration where it can be observed some physical phenomena like gravity, collision, inertia and friction. Such integration was created in order to make more pleasant learning any of the involved technologies. The application was made using the library SDL2 (Simple Direct Media Layer), to manage windows creation and input and output user data on multiple platforms, like Windows, MacOS and Linux. For the creation of the graphical environment, it was used the multiplatform graphical API OpenGL, written in version 3.3 or superior, also referred to as modern, what makes possible to use features not available on previous versions. The physics engine Bullet has support to common features like gravity, collision, rigid bodies, among others. All chosen libraries are open source. The selected tool for development were the IDE (Integrated Development Environment) Visual Studio Community 2015, for Windows, also free, however it is possible to use any other compiler to achieve the same result.

Key-words: Graphical API. Physics engine.

Lista de Figuras

8 *Rendering pipeline* OpenGL (Fonte: OPENGL, 2016b) . . . . . p. 19

9 *Tesselation* (Fonte: OPENGL, 2016c) . . . . . p. 20

10 *Efeitos de fumaça em Rocket League* (Fonte: ESPN, 2016) . . . . . p. 21

11 *Clipping* (Fonte: WRIGHT, HAEMEL, 2016) . . . . . p. 22

12 Projeção ortográfica . . . . . p. 23

13 Projeção em perspectiva . . . . . p. 23

14 *Determinando a orientação de um triângulo* (Fonte: OPENGL, 2016) . . . . . p. 24

Sumário

1 Introdução

2 OpenGL

Ao longo do final do século XX, a computação gráfica ganhou grande importância na vida de pessoas comuns. Tal tecnologia se tornou presente não apenas em usos militares como SAGE (*Semi-Automatic Ground Environment*), sistema que convertia dados de radares em imagens computadorizadas (MACHOVER, 1978) na década de 50, como também em filmes como Toy Story (1995), primeiro longa-metragem completamente computadorizado (GUHA, 2010).

Na década de 80, boa parte do trabalho gráfico era produzido em *workstations*, que utilizavam a APIs com diferentes implementações, dificultando a produção de programas multiplataforma. Em 1982, a SGI (*Silicon Graphics, Inc*) começou o desenvolvimento de sua API proprietária IRIS GL, atingindo alta popularidade (MARTZ, 2006). A pressão por uma API aberta e unificada aumentou por parte dos desenvolvedores de *software*, culminando em 1991 com a formulação do OpenGL ARB (*Architecture Review Board*), consórcio de empresas para regulamentar o projeto, e o lançamento da versão 1.0 no ano seguinte.

Segundo Wright (2013), o OpenGL é uma camada de abstração entre o *software* e o sistema gráfico, devendo permitir que a aplicação execute independente dos diferentes tipos de *hardware*. Além disso, a API precisa operar em diferentes sistemas operacionais, arquiteturas e resoluções de tela, ao mesmo tempo que expõe as características de cada *hardware* para que o programador faça o melhor uso.

Até o lançamento da versão 2.0 em 2004, todo o processo percorrido pelos vetores até a sua transformação em *pixels* na tela era imutável - chamado *fixed-function pipeline*. Devido a essa padronização, era possível otimizar várias etapas do processo diretamente no *hardware* (BAILEY e CUNNINGHAM, 2012). Por outro lado, alguns efeitos eram difíceis ou não poderiam ser obtidos. Com a evolução tecnológica, tornou-se viável a criação de programas, chamados *shaders*, diretamente para a GPU (*Graphics Processing Unit*).

Através dos *shaders*, pode-se manipular de diferentes formas algumas etapas do processamento gráfico. No OpenGL é usada a linguagem GLSL (OpenGL *Shading Language*),

baseada em ANSI C, de onde foi simplificada e adicionada de alguns elementos constantemente presentes na computação gráfica como vetores e matrizes (OPENGL, 2016a). Atualmente, existem quatro *shaders* disponíveis: *vertex*, *fragment*, *geometry* e *tesselation*, cada qual atuando em uma etapa diferente.

Com o tempo, várias funções foram adicionadas a especificação, o que tornava difícil a compatibilidade entre todas elas. Por esse motivo, em 2008, na versão 3.0, o OpenGL ARB decidiu criar dois perfis: *core* e *compatibility*, separando padrões suportados por arquiteturas modernas e obsoletas, respectivamente. No novo padrão *core*, a *programmable pipeline* substituiu a antiga *fixed-function pipeline*, tornando obrigatório o uso de *shaders*.

Desde 2006, o OpenGL ARB se encontra dentro do Khronos Group, consórcio de empresas "para criação de padrões abertos para computação paralela, gráfica e de mídia dinâmica"(KHRONOS, 2016). Atualmente, a especificação se encontra na versão 4.5, lançada em 2014.

3 *Rendering Pipeline*

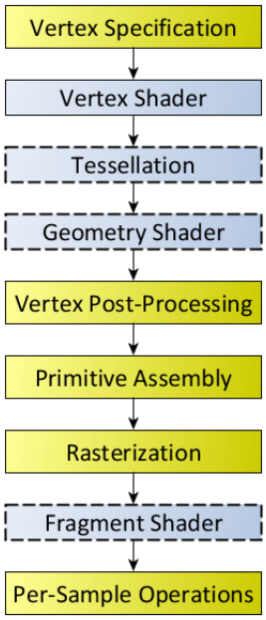


Figura 1: *Rendering pipeline* OpenGL (Fonte: OPENGL, 2016b)

O processo de execução de um programa que exibe resultados em um monitor gráfico requer uma série de etapas, com grande quantidade de cálculos (CLEMENTS, 2014). Assim como a CPU (*Central Processing Unit*, Unidade de Processamento Central), que realiza o processamento geral do computador, a GPU (*Graphics Processing Unit*, Unidade de Processamento Gráfico), também se beneficia de *pipeline*, técnica que consiste em separar as instruções em diversas partes (GOVINDARAJALU, 2004).

Tal processo é definido pela execução paralela de múltiplas etapas, diminuindo a ociosidade do *hardware* e aumentando a taxa de saída de dados (*throuput*) (SHEN e LIPASTI, 2013). Logo que uma primeira instrução é finalizada em uma etapa, uma segunda pode iniciar, desde que não possua outras dependências.

A primeira etapa programável de processamento é o *vertex shader*, sendo também a única obrigatória. Sua função é efetuar cálculos, tendo apenas um vértice como entrada e

saída de dados (OPENGL, 2016b). Para funcionar corretamente, o programador deve especificar a entrada, chamada de *vertex attribute*. Isto é feito através da chamada de funções específicas do OpenGL, utilizando como parâmetros o nome dos atributos.

Código 1: "Exemplo de *vertex shader*"

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

        out vec3 Normal;
        out vec3 FragPos;

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

        void main()
        {
                gl_Position = projection * view *  model * vec4(position,
                   1.0f);
                FragPos = vec3(model * vec4(position, 1.0f));
                Normal = mat3(transpose(inverse(model))) * normal;
        }
}
```

No código 4, podemos notar as entradas definidas por `layout(location = #)` e as saídas por `out`, que serão utilizadas na próxima etapa do *pipeline*. Devido a natureza independente dos vértices, estes podem ser processados paralelamente em alta velocidade, fazendo uso dos múltiplos núcleos da GPU (AKENINE-MÖLLER et al, 2016).

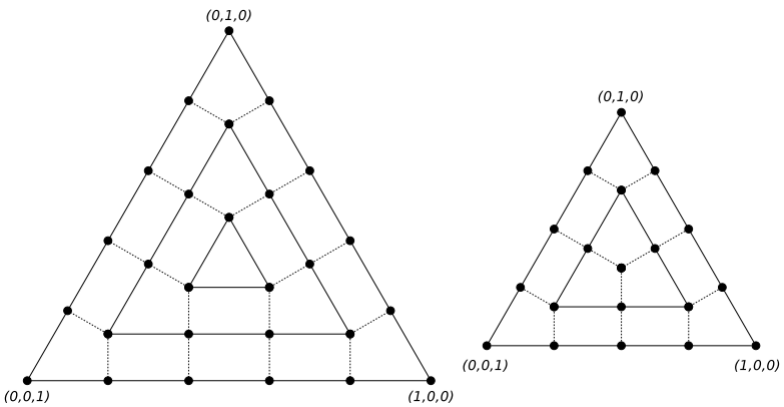


Figura 2: *Tesselation* (Fonte: OPENGL, 2016c)

A próxima etapa de processamento é chamada de *tesselation*, que consiste de duas etapas programáveis opcionais e uma fixa. Este estágio atua sobre *patches*, uma primitiva determinada por uma certa quantidade de vértices especificada pelo programador. Primitivas são tipos de dados processados no OpenGL, representados por sequências de vértices, como pontos, linhas, triângulos e sequências dos dois últimos (OPENGL, 2016c).

Na primeira operação, o *tesselation control shader* recebe um *patch* de entrada e emite outro de saída, podendo determinar as operações a serem efetuadas a cada vértice e a cada *patch*. Através dele é possível aplicar subdivisões de geometria diretamente da GPU (BAILEY e CUNNINGHAM, 2012), por exemplo, para alterar a quantidade de vértices de um objeto. Com uma quantidade maior, o nível de detalhamento da superfície também aumenta. Pode ser usado para criar o efeito de um objeto sendo visto de diferentes distâncias, ao mesmo tempo que economiza o processamento de detalhes desnecessários.

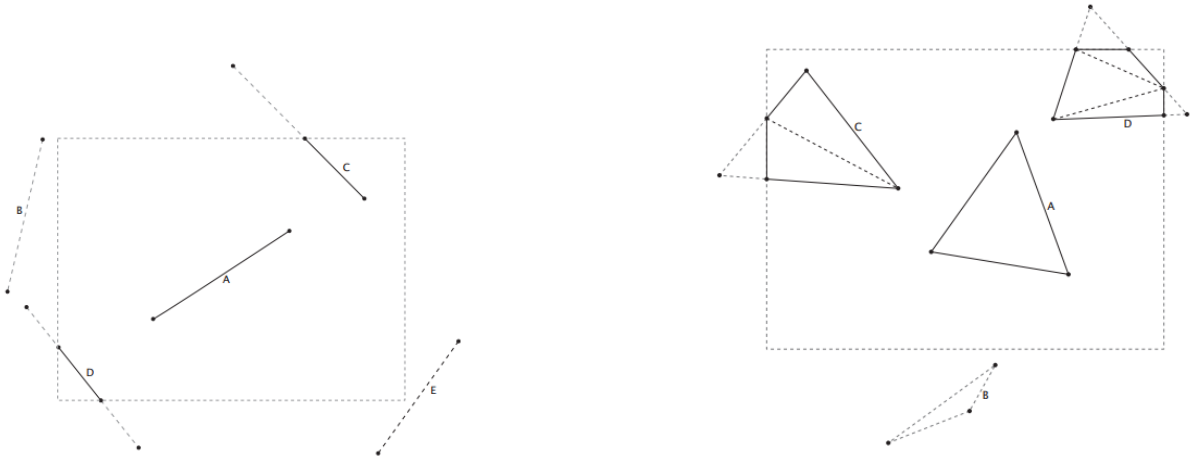
Caso o *tesselation evaluation shader* esteja presente, os dados são transferidos para a etapa fixa *tesselation primitive generation*. Sua função é transformar os *patches* gerados anteriormente em primitivas, de acordo com os valores especificados pelo *shader*. Por último, o *evaluation shader* gera efetivamente os novos vértices e seus atributos.

No próximo estágio opcional, o *geometry shader*, cada primitiva pode ser processada novamente, gerando nenhuma ou mais primitivas. Uma de suas características importantes é a capacidade de executar múltiplas vezes para a mesma primitiva, gerando resultados que podem ser utilizados em diferentes funções. Por exemplo, processar um objeto e extrair sua silhueta para gerar sombras dinamicamente ao mesmo tempo (NVIDIA, 2016).

Figura 3: *Efeitos de fumaça em Rocket League* (Fonte: ESPN, 2016)

Agora, os vértices passam por uma série de processamentos fixos, o primeiro deles sendo o *transform feedback*. As primitivas geradas anteriormente podem ser armazenadas e recuperadas em *buffer objects*, memórias alocadas pelo programa OpenGL. Isso possibilita que a informação em uso não precise transitar entre a memória RAM e a GPU, diminuindo o

tempo de processamento. É útil em sistemas de partículas, por exemplo, onde é necessário simular um grande número de objetos que agem de uma maneira pré-determinada, como a fumaça de uma explosão (figura 10).



(a) *Clipping de linha* (b) *Clipping de triângulo*

Figura 4: *Clipping* (Fonte: WRIGHT, HAEMEL, 2016)

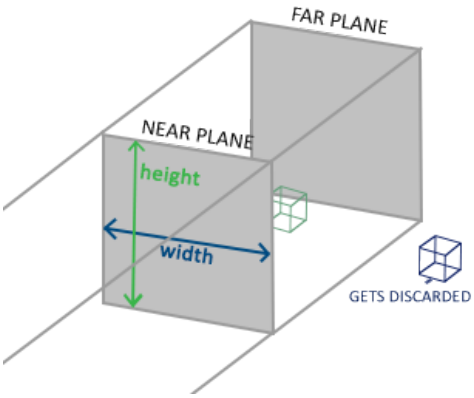
O próximo passo é o *clipping*, onde são determinadas quais primitivas irão aparecer ou não na tela do programa. Pontos isolados fora das coordenadas especificadas são descartados automaticamente. Nas figuras 11a e 11b, podemos notar que as linhas e os triângulos C e D seriam parcialmente inclusos na visualização. Nesse caso são geradas novas primitivas que comportam somente as partes que aparecerão no resultado final (OPENGL, 2016d).

$$x = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ \frac{w}{w} \end{pmatrix} \tag{1}$$

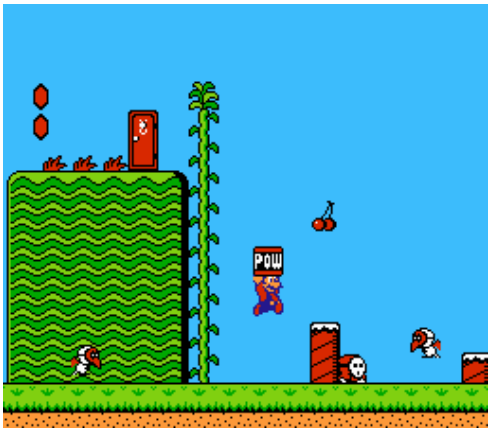
Até o momento, cada coordenada é especificada em vetores de 4 componentes (coordenada homogênea), $v = (x, y, z, w)$, uma vez que torna possível o processo de apresentar objetos de três dimensões em duas dimensões da tela do programa (HAEMEL, SELLERS, 2013). Ocorre agora o processo chamado *perspective division* (equação 2), onde todos os componentes são divididos pelo componente w. Dessa forma é feita a transição do espaço homogêneo para o espaço cartesiano, que representa pontos através de n-coordenadas em um espaço n-dimensional (WOLFRAM, 2016). Após a operação, as coordenadas se encontram normalizadas (*Normalized Device Coordinates*), com valores entre -1 e 1.

Os dois tipos de projeção mais utilizados são a ortográfica e a em perspectiva. Ambas

criam volumes no espaço em formato de tronco de paralelepípedo e pirâmide, respectivamente.



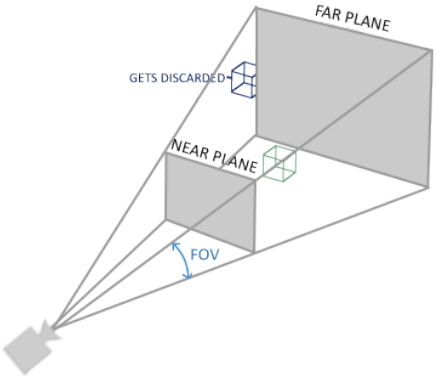
(a) *Projeção ortográfica* (Fonte: Learn OpenGL, 2016)



(b) *Jogo Super Mario Bros 2, projeção ortográfica* (Fonte: Wikipédia, 2016)

Figura 5: Projeção ortográfica

Na projeção ortográfica (figura 12a) definimos largura, altura e comprimento da projeção. É útil na visualização de elementos 2D, como desenhos técnicos e jogos de plataforma, uma vez que as coordenadas são mapeadas diretamente na tela, não havendo distorção. Nesse caso, o valor do componente *w* é constante para todas as coordenadas.



(a) *Projeção em perspectiva* (Fonte: Learn OpenGL, 2016)

(b) *Jogo Life is Strange, projeção em perspectiva* (Fonte: Kotaku, 2016)

Figura 6: Projeção em perspectiva

Alterando o valor da coordenada *w* de acordo com a distância do observador, temos a projeção em perspectiva, onde é criada a ilusão de distância, onde objetos mais distantes são renderizados menores que objetos mais próximos. Na figura 13b, podemos observar que os trilhos do trem convergem para o centro da imagem a medida que se distancia do observador.

Código 2: "Função glViewport"

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Agora as coordenadas serão transferidas do espaço normalizado para o espaço da janela (*viewport*). A transformação é definida pela função 5, onde `x` e `y` são as coordenadas do canto inferior esquerdo da janela e `width` e `height`, a largura e altura, respectivamente.

No *primitive assembly*, os vértices advindos da última etapa são convertidos em uma sequência de primitivas - pontos, linhas ou triângulos. A renderização também pode ser abortada nessa etapa, caso não seja necessário visualizar o resultado dos processamentos anteriores, como na utilização do *transform feedback*.

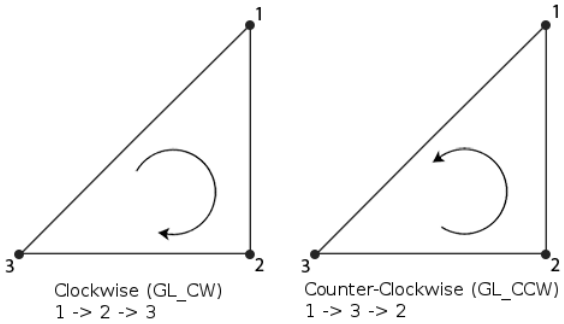


Figura 7: *Determinando a orientação de um triângulo* (Fonte: OPENGL, 2016)

Ocorre agora uma importante etapa de otimização, o *face culling*, onde os triângulos voltados para o lado oposto ao do observador são descartados da renderização. Através da ordem de cada vértice do triângulo é possível obter a sua orientação (figura 14). Por padrão, triângulos com os vértices no sentido anti-horário são considerados como de frente.

As primitivas que não foram descartadas nos estágios anteriores são transformadas em fragmentos em um processo chamado *rasterization*. Cada fragmento possui uma série de informações associadas, que serão utilizadas para determinar sua cor através do *fragment shader* (WRIGHT, HAEMEL, 2016). Entre essas informações estão sua posição na tela, profundidade e outros dados relevantes, como variáveis para cálculo de iluminação, mostradas no código 6.

Ocorre agora uma série de otimizações para descartar fragmentos antes da execução do *fragment shader*. Três operações são efetuadas obrigatoriamente em cada fragmento: *pixel ownership test*, *scissor test* e *multisample fragment operation*. Caso tenham sido ativadas, também são efetuadas *stencil test*, *depth buffer test* e *occlusion query sample counting* (OPENGL, 2016c).

No *pixel ownership test*, cada pixel da janela é testado para verificar se o OpenGL possui controle sobre ele. Um caso onde isso não é verdade acontece quando a janela está oculta por outra.

No *scissor test*, as coordenadas são testadas contra um retângulo que pode ser determinado no programa. Dessa forma é possível realizar operações somente em certas partes da tela, sem afetar o todo.

Caso o *multisampling* esteja ativo, o *multisample fragment operation* é executado. *Multisampling* é uma técnica utilizada para remover o *aliasing*, fenômeno que acontece quando a qualidade da amostra não é suficiente para apresentar satisfatoriamente o conteúdo (WRIGHT, HAEMEL, 2016). Tal defeito é demonstrado através de deformações nas arestas e bordas dos objetos renderizados.

Código 3: "Exemplo de *fragment shader*"

```
#version 330 core

struct Light {
vec3 direction;
vec3 ambient;
vec3 diffuse;
vec3 specular;
};

in vec3 FragPos;
in vec3 Normal;

out vec4 color;

uniform vec3 viewPos;
uniform vec3 objectColor;
uniform Light light;

void main()
{
// Ambient
vec3 ambient = light.ambient;

// Diffuse
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(-light.direction);
float diff = max(dot(norm, lightDir), 0.0);
```

```
    vec3 diffuse = light.diffuse * diff;

    // Specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 20);
    vec3 specular = light.specular * spec * 0.3;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    color = vec4(result, 1.0f);
    }
```

O *fragment shader* (código 6) é uma etapa programável e opcional onde podem ser definidas informações de iluminação, cor e textura. No código citado, definimos uma `struct` que será responsável por armazenar os dados de iluminação para cada fragmento. Declarando-a como `uniform`, podemos acessar seu valor em qualquer ponto de nossa aplicação para definir seus valores. Isso também é feito com as variáveis `viewPos` e `objectColor`, onde iremos armazenar a posição do observador e a cor do fragmento, respectivamente. Note que as variáveis de entrada `FragPos` e `Normal` são as mesmas que definimos como a saída do *vertex shader* do código 4.

4 *Rendering Pipeline*

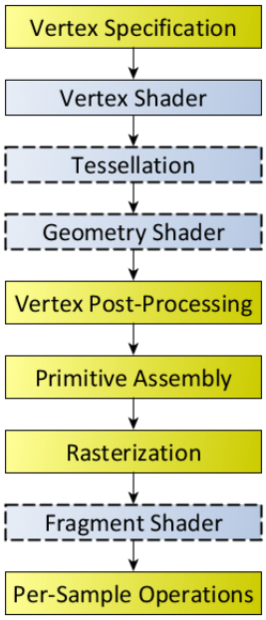


Figura 8: *Rendering pipeline* OpenGL (Fonte: OPENGL, 2016b)

O processo de execução de um programa que exibe resultados em um monitor gráfico requer uma série de etapas, com grande quantidade de cálculos (CLEMENTS, 2014). Assim como a CPU (*Central Processing Unit*, Unidade de Processamento Central), que realiza o processamento geral do computador, a GPU (*Graphics Processing Unit*, Unidade de Processamento Gráfico), também se beneficia de *pipeline*, técnica que consiste em separar as instruções em diversas partes (GOVINDARAJALU, 2004).

Tal processo é definido pela execução paralela de múltiplas etapas, diminuindo a ociosidade do *hardware* e aumentando a taxa de saída de dados (*throuput*) (SHEN e LIPASTI, 2013). Logo que uma primeira instrução é finalizada em uma etapa, uma segunda pode iniciar, desde que não possua outras dependências.

A primeira etapa programável de processamento é o *vertex shader*, sendo também a única obrigatória. Sua função é efetuar cálculos, tendo apenas um vértice como entrada e

saída de dados (OPENGL, 2016b). Para funcionar corretamente, o programador deve especificar a entrada, chamada de *vertex attribute*. Isto é feito através da chamada de funções específicas do OpenGL, utilizando como parâmetros o nome dos atributos.

Código 4: "Exemplo de *vertex shader*"

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

        out vec3 Normal;
        out vec3 FragPos;

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

        void main()
        {
                gl_Position = projection * view *  model * vec4(position,
                    1.0f);
                FragPos = vec3(model * vec4(position, 1.0f));
                Normal = mat3(transpose(inverse(model))) * normal;
        }
}
```

No código 4, podemos notar as entradas definidas por `layout(location = #)` e as saídas por `out`, que serão utilizadas na próxima etapa do *pipeline*. Devido a natureza independente dos vértices, estes podem ser processados paralelamente em alta velocidade, fazendo uso dos múltiplos núcleos da GPU (AKENINE-MÖLLER et al, 2016).

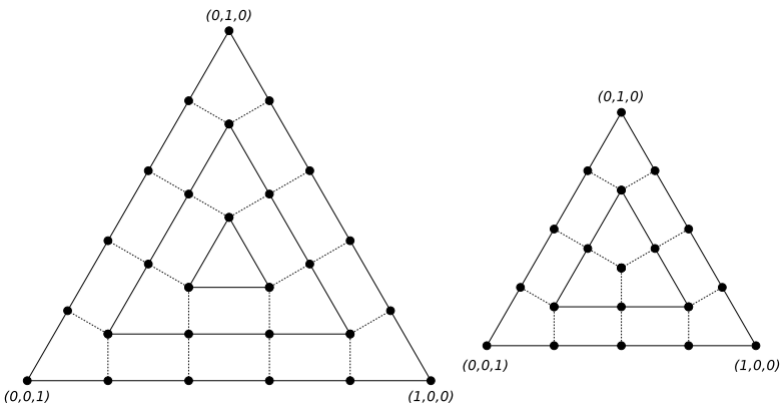


Figura 9: *Tesselation* (Fonte: OPENGL, 2016c)

A próxima etapa de processamento é chamada de *tesselation*, que consiste de duas etapas programáveis opcionais e uma fixa. Este estágio atua sobre *patches*, uma primitiva determinada por uma certa quantidade de vértices especificada pelo programador. Primitivas são tipos de dados processados no OpenGL, representados por sequências de vértices, como pontos, linhas, triângulos e sequências dos dois últimos (OPENGL, 2016c).

Na primeira operação, o *tesselation control shader* recebe um *patch* de entrada e emite outro de saída, podendo determinar as operações a serem efetuadas a cada vértice e a cada *patch*. Através dele é possível aplicar subdivisões de geometria diretamente da GPU (BAILEY e CUNNINGHAM, 2012), por exemplo, para alterar a quantidade de vértices de um objeto. Com uma quantidade maior, o nível de detalhamento da superfície também aumenta. Pode ser usado para criar o efeito de um objeto sendo visto de diferentes distâncias, ao mesmo tempo que economiza o processamento de detalhes desnecessários.

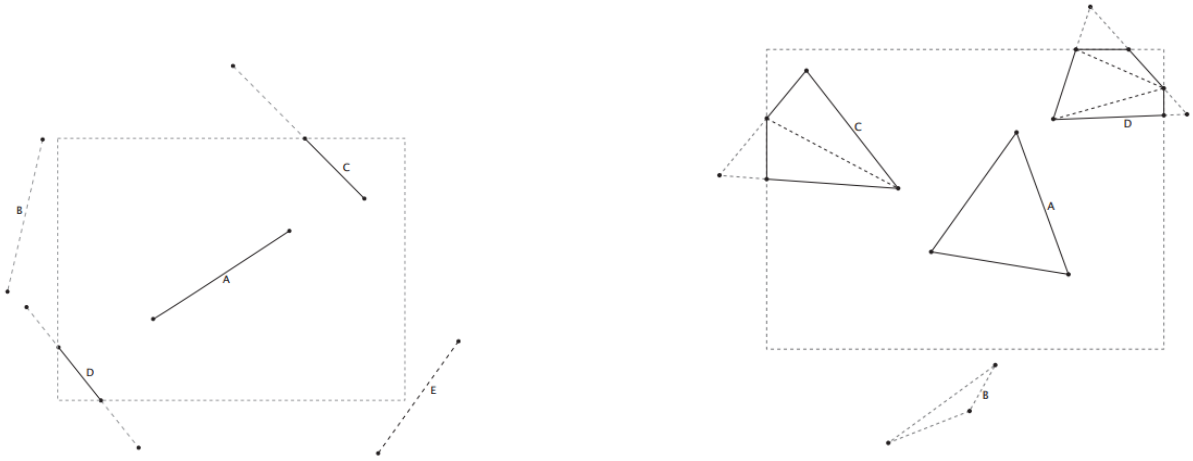
Caso o *tesselation evaluation shader* esteja presente, os dados são transferidos para a etapa fixa *tesselation primitive generation*. Sua função é transformar os *patches* gerados anteriormente em primitivas, de acordo com os valores especificados pelo *shader*. Por último, o *evaluation shader* gera efetivamente os novos vértices e seus atributos.

No próximo estágio opcional, o *geometry shader*, cada primitiva pode ser processada novamente, gerando nenhuma ou mais primitivas. Uma de suas características importantes é a capacidade de executar múltiplas vezes para a mesma primitiva, gerando resultados que podem ser utilizados em diferentes funções. Por exemplo, processar um objeto e extrair sua silhueta para gerar sombras dinamicamente ao mesmo tempo (NVIDIA, 2016).

Figura 10: *Efeitos de fumaça em Rocket League* (Fonte: ESPN, 2016)

Agora, os vértices passam por uma série de processamentos fixos, o primeiro deles sendo o *transform feedback*. As primitivas geradas anteriormente podem ser armazenadas e recuperadas em *buffer objects*, memórias alocadas pelo programa OpenGL. Isso possibilita que a informação em uso não precise transitar entre a memória RAM e a GPU, diminuindo o

tempo de processamento. É útil em sistemas de partículas, por exemplo, onde é necessário simular um grande número de objetos que agem de uma maneira pré-determinada, como a fumaça de uma explosão (figura 10).



(a) *Clipping de linha* (b) *Clipping de triângulo*

Figura 11: *Clipping* (Fonte: WRIGHT, HAEMEL, 2016)

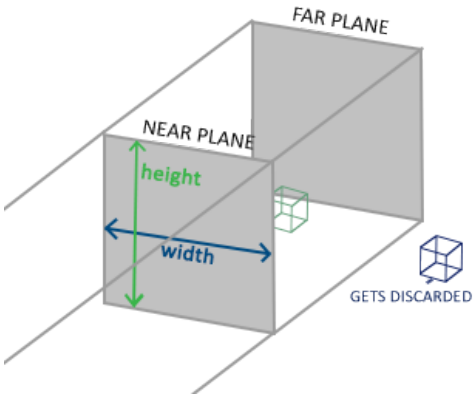
O próximo passo é o *clipping*, onde são determinadas quais primitivas irão aparecer ou não na tela do programa. Pontos isolados fora das coordenadas especificadas são descartados automaticamente. Nas figuras 11a e 11b, podemos notar que as linhas e os triângulos C e D seriam parcialmente inclusos na visualização. Nesse caso são geradas novas primitivas que comportam somente as partes que aparecerão no resultado final (OPENGL, 2016d).

$$x = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ \frac{w}{w} \end{pmatrix} \tag{2}$$

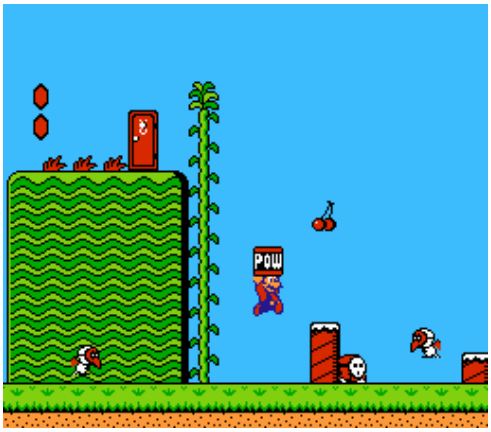
Até o momento, cada coordenada é especificada em vetores de 4 componentes (coordenada homogênea), $v = (x, y, z, w)$, uma vez que torna possível o processo de apresentar objetos de três dimensões em duas dimensões da tela do programa (HAEMEL, SELLERS, 2013). Ocorre agora o processo chamado *perspective division* (equação 2), onde todos os componentes são divididos pelo componente w. Dessa forma é feita a transição do espaço homogêneo para o espaço cartesiano, que representa pontos através de n-coordenadas em um espaço n-dimensional (WOLFRAM, 2016). Após a operação, as coordenadas se encontram normalizadas (*Normalized Device Coordinates*), com valores entre -1 e 1.

Os dois tipos de projeção mais utilizados são a ortográfica e a em perspectiva. Ambas

criam volumes no espaço em formato de tronco de paralelepípedo e pirâmide, respectivamente.



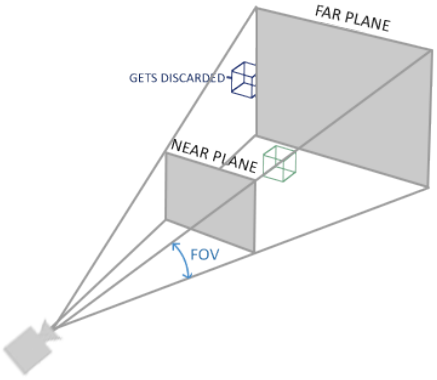
(a) *Projeção ortográfica* (Fonte: Learn OpenGL, 2016)



(b) *Jogo Super Mario Bros 2, projeção ortográfica* (Fonte: Wikipédia, 2016)

Figura 12: Projeção ortográfica

Na projeção ortográfica (figura 12a) definimos largura, altura e comprimento da projeção. É útil na visualização de elementos 2D, como desenhos técnicos e jogos de plataforma, uma vez que as coordenadas são mapeadas diretamente na tela, não havendo distorção. Nesse caso, o valor do componente *w* é constante para todas as coordenadas.



(a) *Projeção em perspectiva* (Fonte: Learn OpenGL, 2016)

(b) *Jogo Life is Strange, projeção em perspectiva* (Fonte: Kotaku, 2016)

Figura 13: Projeção em perspectiva

Alterando o valor da coordenada *w* de acordo com a distância do observador, temos a projeção em perspectiva, onde é criada a ilusão de distância, onde objetos mais distantes são renderizados menores que objetos mais próximos. Na figura 13b, podemos observar que os trilhos do trem convergem para o centro da imagem a medida que se distancia do observador.

Código 5: "Função glViewport"

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Agora as coordenadas serão transferidas do espaço normalizado para o espaço da janela (*viewport*). A transformação é definida pela função 5, onde `x` e `y` são as coordenadas do canto inferior esquerdo da janela e `width` e `height`, a largura e altura, respectivamente.

No *primitive assembly*, os vértices advindos da última etapa são convertidos em uma sequência de primitivas - pontos, linhas ou triângulos. A renderização também pode ser abortada nessa etapa, caso não seja necessário visualizar o resultado dos processamentos anteriores, como na utilização do *transform feedback*.

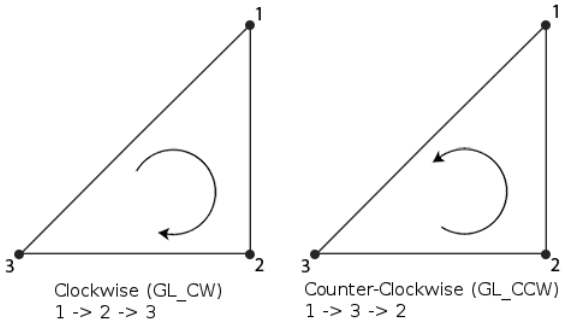


Figura 14: *Determinando a orientação de um triângulo* (Fonte: OPENGL, 2016)

Ocorre agora uma importante etapa de otimização, o *face culling*, onde os triângulos voltados para o lado oposto ao do observador são descartados da renderização. Através da ordem de cada vértice do triângulo é possível obter a sua orientação (figura 14). Por padrão, triângulos com os vértices no sentido anti-horário são considerados como de frente.

As primitivas que não foram descartadas nos estágios anteriores são transformadas em fragmentos em um processo chamado *rasterization*. Cada fragmento possui uma série de informações associadas, que serão utilizadas para determinar sua cor através do *fragment shader* (WRIGHT, HAEMEL, 2016). Entre essas informações estão sua posição na tela, profundidade e outros dados relevantes, como variáveis para cálculo de iluminação, mostradas no código 6.

Ocorre agora uma série de otimizações para descartar fragmentos antes da execução do *fragment shader*. Três operações são efetuadas obrigatoriamente em cada fragmento: *pixel ownership test*, *scissor test* e *multisample fragment operation*. Caso tenham sido ativadas, também são efetuadas *stencil test*, *depth buffer test* e *occlusion query sample counting* (OPENGL, 2016c).

No *pixel ownership test*, cada pixel da janela é testado para verificar se o OpenGL possui controle sobre ele. Um caso onde isso não é verdade acontece quando a janela está oculta por outra.

No *scissor test*, as coordenadas são testadas contra um retângulo que pode ser determinado no programa. Dessa forma é possível realizar operações somente em certas partes da tela, sem afetar o todo.

Caso o *multisampling* esteja ativo, o *multisample fragment operation* é executado. *Multisampling* é uma técnica utilizada para remover o *aliasing*, fenômeno que acontece quando a qualidade da amostra não é suficiente para apresentar satisfatoriamente o conteúdo (WRIGHT, HAEMEL, 2016). Tal defeito é demonstrado através de deformações nas arestas e bordas dos objetos renderizados.

Código 6: "Exemplo de *fragment shader*"

```
#version 330 core

struct Light {
vec3 direction;
vec3 ambient;
vec3 diffuse;
vec3 specular;
};

in vec3 FragPos;
in vec3 Normal;

out vec4 color;

uniform vec3 viewPos;
uniform vec3 objectColor;
uniform Light light;

void main()
{
// Ambient
vec3 ambient = light.ambient;

// Diffuse
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(-light.direction);
float diff = max(dot(norm, lightDir), 0.0);
```

```
    vec3 diffuse = light.diffuse * diff;

    // Specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 20);
    vec3 specular = light.specular * spec * 0.3;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    color = vec4(result, 1.0f);
    }
```

O *fragment shader* (código 6) é uma etapa programável e opcional onde podem ser definidas informações de iluminação, cor e textura. No código citado, definimos uma `struct` que será responsável por armazenar os dados de iluminação para cada fragmento. Declarando-a como `uniform`, podemos acessar seu valor em qualquer ponto de nossa aplicação para definir seus valores. Isso também é feito com as variáveis `viewPos` e `objectColor`, onde iremos armazenar a posição do observador e a cor do fragmento, respectivamente. Note que as variáveis de entrada `FragPos` e `Normal` são as mesmas que definimos como a saída do *vertex shader* do código 4.

5 *Bullet Physics*

Bullet Physics é uma biblioteca livre para uso comercial escrita na linguagem de programação C++. É capaz de detectar e resolver a colisão entre objetos, interações entre articulações, além de simulações de tecidos e fluídos (BULLET, 2015).

A biblioteca é estruturada para ser utilizada como um todo ou apenas em parte. Pode-se utilizar somente a função de detecção de colisão, apenas o componente de *rigid body* ou *soft body* separadamente, ou mesmo somente a biblioteca de cálculos.

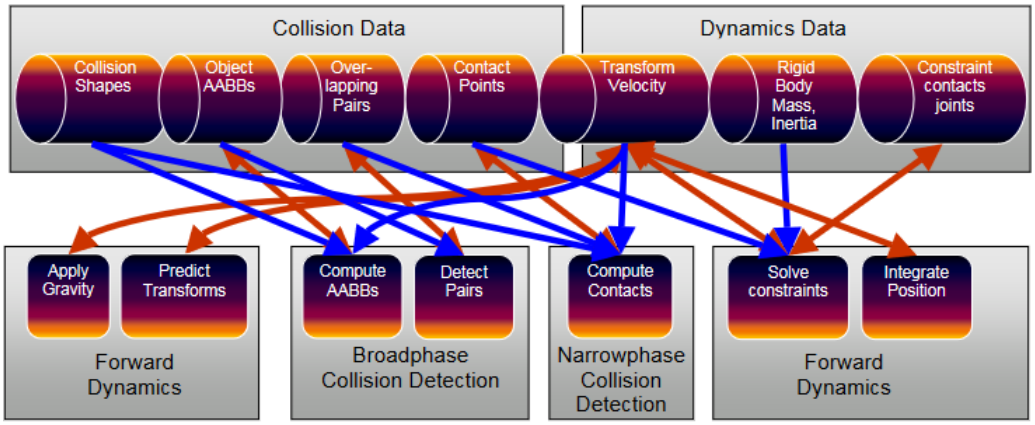


Figura 15: *Pipeline* Bullet Physics (Fonte: BULLET, 2015)

A figura 15 apresenta os passos executados na implementação padrão, chamada de `btDiscreteDynamicsWorld`, e onde são utilizados alguns objetos da biblioteca.

5.1 Detecção de colisão

Um dos recursos da biblioteca é a detecção de colisão. Na *broadphase*, ou fase ampla, um algoritmo simples é executado para separar objetos que podem se colidir ou estão em colisão de objetos que não podem se colidir, seja porque estão parados ou muito distantes entre si. Posteriormente será executado um algoritmo mais complexo para definir as interações entre objetos colidindo.

Tais objetos são definidos por `btCollisionObject`, que contém suas coordenadas e orientação, além de sua forma de colisão. Uma vez que as formas primitivas possuem algoritmos de colisão mais simples, é aconselhável utiliza-las sempre que possível. A figura 16 demonstra a aplicação da esfera, caixa, forma convexa e malha de triângulos, respectivamente. Existem outras formas como cilindro, cone, cápsula, forma côncava e planos.

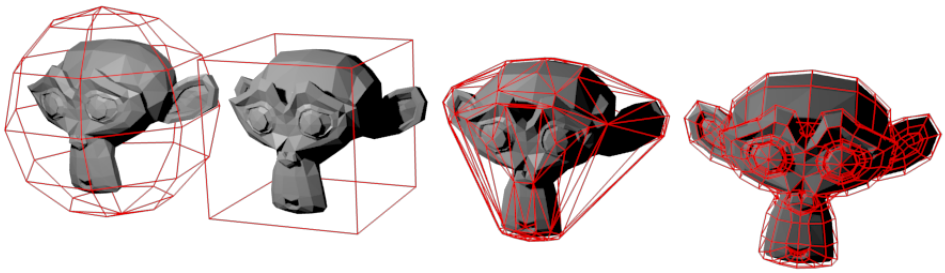


Figura 16: Diferentes formas de colisão aplicadas (Fonte: opengl-tutorial, 2017)

5.2 Dinâmica de corpos rígidos

Corpos rígidos são definidos como um corpo composto de partículas que permanecem a distâncias fixas umas das outras ao longo do tempo. Representam boa parte dos objetos, com exceção corpos que se deformam, como tecidos e líquidos.

Na Bullet Physics é definida por um objeto `btRigidBody`, que contém as propriedades necessárias para simular um corpo rígido, como fricção, restituição, velocidade linear e angular. Tais corpos podem ser estáticos, dinâmicos ou *kinematic*. Objetos estáticos possuem massa infinita, declarada como zero na *engine*, não se movem mas podem colidir. Objetos dinâmicos possuem massa positiva e têm sua posição atualizada a cada passo da simulação. Objetos *kinematic* possuem massa infinita e têm sua posição controlada pelo sistema, não pela Bullet.

Para a fazer a interface entre o programa e o modelo físico pode se utilizar de um *motion state*. Através dele a Bullet comunica as alterações nos objetos para que o programa atualize. No caso dos objetos *kinematic*, o funcionamento é inverso: o programa atualiza a Bullet das atualizações feitas para que a biblioteca possa efetuar os cálculos. O método `getWorldTransform` retorna um `btTransform` com a posição e rotação atualizada do objeto.

Referências

MACHOVER, Carl. *A Brief, Personal History of Computer Graphics*. 1978. p.38. Disponível em: https://www.computer.org/csdl/mags/co/1978/11/01646756.pdf. Acesso em: 07 de junho de 2016.

GUHA, Sumanta. *Computer Graphics Through OpenGL: From Theory to Experiments, Second Edition*. 2014. A K Peters/CRC Press. p.8-9.

MARTZ, Paul. *OpenGL Distilled*. 2016. Addison-Wesley Professional. p.26-8.

WRIGHT, Richard. HAEMEL, Nicholas. SELLERS, Graham. *OpenGL SuperBible: Comprehensive Tutorial and Reference, 6th Edition*. 2013. Addison-Wesley Professional. p.4-9, 38-39, 42.

ROST, Randi. LICEA-KANE, Bill. *OpenGL Shading Language*. 2009. Addison-Wesley Professional. p.26-8.

BAILEY, Mike. CUNNINGHAM, Steve. *Graphic Shaders: Theory and Practice, 2nd Edition*. 2012. CRC Press. p.20-1, 50.

OpenGL. 2016. *OpenGL Shading Language Specification*. Disponível em: https://www.opengl.org/documentation/glsl/. Acesso em: 13 de junho de 2016.

Khronos Group. 2016. *OpenGL ARB to Pass Control of OpenGL Specification to Khronos Group*. Disponível em: https://www.khronos.org/news/press/opengl_arb_to_pass_control_of_opengl_cation_to_khronos_group. Acesso em: 13 de junho de 2016.

OpenGL. 2016a. *History of OpenGL*. Disponível em: https://www.opengl.org/wiki/History_of_OpenGL#Deprecation_Model. Acesso em: 15 de julho de 2016.

Khronos Group. 2016. *About The Khronos Group*. Disponível em: https://www.khronos.org/about/. Acesso em: 13 de junho de 2016.

CLEMENTS, Alan. *Computer Organization & Architecture: Themes and Variations*. 2014. Cengage Learning. p.858-861.

SHEN, John. LIPASTI, Mikko. *Modern Processor Design: Fundamentals of Superscalar Processors*. 2013. Waveland Press, Inc. p.40, 51.

OpenGL. 2016b. *Rendering Pipeline Overview*. Disponível em: https://www.opengl.org/wiki/Rendering_Pipeline_Overview. Acesso em: 18 de julho de 2016.

OpenGL. 2016c. *The OpenGL Graphics System: A Specification*. Disponível em: https://www.opengl.org/registry/doc/glspec45.core.pdf. Acesso em: 19 de julho de 2016.

AKENINE-MÖLLER, Tomas. HAINES, Eric. HOFFMAN, Naty. *Real-Time Rendering, Third Edition*. 2016. CRC Press. p.??????????.

NVIDIA. 2016. *GPU Gems 3, capítulo 11 - Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders*. Disponível em: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch11.html. Acesso em: 14 de agosto de 2016.

ESPN. 2016. *Rocket League*. Disponível em: http://www.espn.com/esports/story/_/id/17356148/rocket-league-rostermania-free-agents. Acesso em: 22 de agosto de 2016.

Learn OpenGL. 2016. *Learn OpenGL*. Disponível em: http://www.learnopengl.com/#!Getting-started/Coordinate-Systems. Acesso em: 11 de setembro 2016.

OpenGL. 2016d. *Clipping*. Disponível em: https://www.opengl.org/wiki/Clipping. Acesso em: 11 de setembro 2016.

Wikipédia. 2016. *Super Mario Bros 2*. Disponível em: https://en.wikipedia.org/wiki/Super_Mario_Bros._2. Acesso em: 11 de setembro 2016.

Kotaku. 2016. *Life is Strange*. Disponível em: http://kotaku.com/i-hope-this-theory-about-life-is-stranges-last-episode-1737361284. Acesso em: 11 de setembro 2016.

Wolfram. 2016. *Cartesian Coordinates*. Disponível em: http://mathworld.wolfram.com/CartesianCoordinates.html. Acesso em: 18 de setembro 2016.

OpenGL. 2016. *Face Culling*. Disponível em: https://www.opengl.org/wiki/Face_Culling. Acesso em: 25 de setembro 2016.

GOVINDARAJALU, Bakthavatsalam. *Computer Architecture and Organization: Design Principles and Applications*. 2004. Tata McGraw-Hill Education. p.556.

Bullet. 2015. *Bullet User Manual*. Disponível em: https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf. Acesso em: 29 de julho de 2017.

opengl-tutorial. 2017. *Picking with a physics library*. Disponível em: http://www.opengl-tutorial.org/ru/miscellaneous/clicking-on-objects/picking-with-a-physics-library/. Acesso em: 17 de setembro 2017.

BOURG, David M. BYWALEC, Bryan. *Physics for Game Developers, 2nd Edition*. 2013. CRC Press. p. XIV.