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

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*

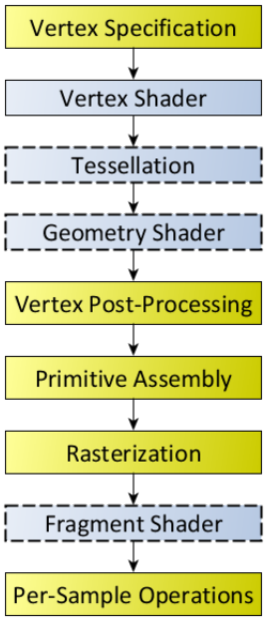

Figura 1: *Rendering pipeline* OpenGL (Fonte: OpenGL, 2016)

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 na 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*.

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*.

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 1, 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).

Referências

[1] 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.

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

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

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

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

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

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

[8] 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_specification_to_khronos_group. Acesso em: 13 de junho de 2016.

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

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

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

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

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

[14] 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.

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